PNNL-22601

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

Piezocon Binary Gas Analyzer Server

DES-0039

Revision 1

Charlie Hubbard

June 2013

Piezocon Binary Gas Analyzer Server

Charlie Hubbard

DES-0039
Revision 1
June 2013

Contents

1 Introduction

This document describes a hardware interface server application designed to manage one or more *Lorex Piezocon*® high-precision binary gas analyzers with single channel controllers. Each managed Piezocon unit is connected to the host computer (the computer on which the server runs) through a separate RS-232 serial interface. Each unit is assigned a short, human-readable, text name by which the unit is known to the server and its clients. Serial ports, unit names and other configuration information are assigned in the server's configuration file (see section 6). Assigned unit names must be unique across the entire project.

The server is implemented by the *piezoconServer.cpp* source module. The client interface to the server is defined by *piezoconClientLib.h* and implemented by *piezoconClientLib.cpp*. A standard text-mode test menu client for the server is implemented by *piezoconMenu.cpp* (see section 8).

The primary documentation for the Piezocon 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 design of the server and its client APIs. If contradictions between this document and the Doxygen-generated documentation are found, the Doxygen-generated documentation should be considered correct.

2 The Piezocon Hardware

The Lorex Piezocon unit is a high-precision binary gas concentration analyzer. The unit functions by measuring the time-delay between the generation of a short spread-spectrum acoustic pulse and the reception of this pulse once it has propagated through the gas sample. From this time delay information, the speed of sound through the sample can be computed, and, together with the temperature of the gas mixture and various other gas-specific physical attributes (such as molecular weight), this information can be used to compute the concentration of the two gases that make up the sample. The precision of the measurement depends in large part on the difference between the molecular weights of the two constituents in the gas mixture. For this reason, the instrument must be configured specifically for the gases under test.

The Piezocon instrument consists of two components; the sensor head itself (through which the sample gas flows) and a controller unit, which provides the spread-spectrum acoustic pulses for the speed-of-sound measurement, performs the concentration calculations and provides the RS-232 interface used for control and data acquisition by the server.

3 Serial Communications

The Piezocon unit can communicate using a variety of protocols including Devicenet, Profibus, Modbus/TCP, Modbus/RTU on RS-485, or simple text-based commands on RS-232. The server expects units to use the RS-232 text-based protocol using the factory default serial port configurations (38,400 baud, 8 data bits, no parity, 1 stop bit, no hardware handshaking).

The RS-232 text-based communication protocol is very simple. It consists of four single-character text commands (possibly followed by a parameter value). Each command string is terminated by a single carriage-return character (no line-feeds). Each command is responded to with a simple response string also terminated by a single carriage-return character. There are no framing or checksum requirements.

The four commands supported by the Piezocon unit are

- `P` — (An uppercase "P") Read the value in the specified parameter register.
- `p` — (a lowercase "P") Write the specified value into the specified parameter register.
- `r0 3` — ("r" zero space three) Puts the unit into *run* mode. In this mode, it spews analysis records out the serial port at rapid and regular intervals.
- `S` — (An uppercase "S") Takes the unit out of *run* mode.

Complete details of the communication protocol are described in the Lorex document *The Piezocon Interface Software Program User's Guide v2.06*. However, there is one caviat here. The Lorex document describes using the `V` and `v` commands to read and write parameter values. This seems to be outdated. The actual units appear to expect `P` and `p` commands for this, as described above. Other than this one discrepancy, the information in the Lorex document appears to be accurate.

4 Gas Codes

The Piezocon binary gas analyzer determines the concentration of one gas (called the *process gas*) in another (called the *carrier gas*), based on temperature of the gas mixture and the speed of sound through the gas mixture. To do this, the Piezocon unit needs to know several specific physical parameters associated with each of the two gases. A new Piezocon unit comes with the necessary physical parameters for several common gases stored in its internal ROM. Each of these gases is given a numerical code that can be used to indicate that particular gas when configuring the unit for use with a specific carrier and process gas. The 22 pre-defined gas codes are given in Table 1 below.

Table 1: The gas codes pre-defined on new Piezocon units

Code	**Gas**
00	N2
01	CO2
02	Air
03	IA
04	H2
05	TMI
06	TCS
07	SICL4
08	GECL4
09	POCL3
10	DCE
11	HCL
12	AR
13	HE
14	O2
15	CL2
16	TEOS
17	TEB
18	TEPO
19	H2O
20	TMP0
21	TMB

Gas codes 22 through 53 are reserved for additional gases that can be added to the unit by the user. Unfortunately, the process of adding additional gases is not straightforward and is somewhat confusing. The reader will need to contact Lorex for the exact process and tools needed to add gases to the gas code table, but, in broad terms, the process is similar to that described below.

1. From the Lorex web site, use their gas library browser to select which additional gases you want to add to a unit.
2. The Lorex gas library browser creates a file in a Lorex proprietary format containing information on the specific gases you've chosen. Download this file to your computer.
3. Transfer this custom gas file to a Windows computer that can be connected to the target Piezocon unit via an RS-232 connection.
4. Use Lorex's Windows-based custom configuration utility to upload the data from the custom gas file into the target Piezocon unit (this will require an appropriately configured RS-232 connection between the Windows computer and the Piezocon).

Unfortunately, this process does not allow the user to specify which gas codes to associate with a particular gas. Instead, gas codes are assigned in order (22 through 53) to the gases in the custom gas file based on *the order in which the gases appear in the Lorex online database*. This means that specific gases may be assigned different gas codes on different units depending on what other gases were (or were not) in the custom gas file created on the Lorex web site. For this reason, it is recommended that all Piezocon units associated with a specific project be configured using the same custom gas file and that special care be taken to record which additional gas codes are assigned to which gases.

5 The Client API

The Piezocon server, like all servers written for our client/server architecture[1], relies on a client API class for its client interface. Client API classes provide one public method for each message supported by the 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. The client API class for the Piezocon server is called `PiezoconClient` and it is defined in *piezoconClientLib.h* and implemented in *piezoconClientLib.cpp*.

Like all servers that use our standard client/server architecture, the Piezocon 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 server does fully support the standard state-of-health reporting mechanism implemented by the `BaseServer` class[2].

In addition to these standard messages, the server can accept and respond to several additional *client-specific* messages that are defined by the server's associated client API. These are described in detail in this section[3].

5.1 PIEZO_GET_NUM_SENSORS

This message is implemented by the `GetNumSensors()` client API class method. It simply returns the number of Piezocon binary gas analyzers the server is managing.

[1]See design document ***DES-0005*, *The Client/Server Architecture***, for details on our client/server architecture.

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

[3]The symbolic constants that comprise the following subsection headers come from the file *piezocon-ClientLib.h*. Please review the Doxygen documentation for that file for more information.

5.2 PIEZO_GET_PROCESS_GAS

This message is implemented by the `GetProcessGas()` client API class method. It returns the gas code (see section 4) for the *process gas* currently defined on the specified Piezocon unit.

5.3 PIEZO_GET_CARRIER_GAS

This message is implemented by the `GetCarrierGas()` client API class method. It returns the gas code (see section 4) for the *carrier gas* currently defined on the specified Piezocon unit.

5.4 PIEZO_GET_PRESSURE_FACTOR

This message is implemented by the `GetPressureFactor()` client API class method. It returns the current pressure factor in effect for the specified Piezocon unit. For a description of *pressure factor* and other configurable parameters, see the Lorex document *Piezocon Sensor and 1-Channel Controller Istallation and User's Guide v8.09*.

5.5 PIEZO_GET_ANSWER_INDEX

This message is implemented by the `GetAnswerIndex()` client API class method. It returns the current answer index in effect for the specified Piezocon unit. For a description of *answer index* and other configurable parameters, see the Lorex document *Piezocon Sensor and 1-Channel Controller Istallation and User's Guide v8.09*.

5.6 PIEZO_GET_DAC_HIGH

This message is implemented by the `GetDACHigh()` client API class method. It returns the current DAC high value in effect for the specified Piezocon unit. For a description of *DAC high value* and other configurable parameters, see the Lorex document *Piezocon Sensor and 1-Channel Controller Istallation and User's Guide v8.09*.

5.7 PIEZO_GET_DAC_LOW

This message is implemented by the `GetDACLow()` client API class method. It returns the current DAC low value in effect for the specified Piezocon unit. For a description of *DAC low value* and other configurable parameters, see the Lorex document *Piezocon Sensor and 1-Channel Controller Istallation and User's Guide v8.09*.

5.8 PIEZO_GET_MULTIPLIER

This message is implemented by the `GetMultiplier()` client API class method. It returns the current concentration multiplier value in effect for the specified Piezocon unit. This is a number that is multiplied by the concentration to provide the final value reported to the user. For example, use a value of 100.0 to convert the reported concentration to percent. For a description of *multiplier value* and other configurable parameters, see the Lorex document *Piezocon Sensor and 1-Channel Controller Istallation and User's Guide v8.09*.

5.9 PIEZO_GET_GAS_DATA

This message is implemented by the `GetGasData()` client API class method. It returns a `ReturnGasData` structure (defined in *piezoconClientLib.h*) containing the most current analysis information reported from the specified Piezocon unit. Typically this method is only called if the specified Piezocon is *running*. If the unit is not running, all reported values in the returned structure will be set to zero.

5.10 PIEZO_GET_ALL_GAS_DATA

This message is implemented by the `GetAllGasData()` client API class method. It returns an STL map of `ReturnGasData` structures (defined in *piezoconClientLib.h*) containing the most current analysis information reported by every (running) Piezocon unit managed by the server. The map is keyed on unit name. For units not currently in a *running* state, all reported values are zero.

5.11 PIEZO_SET_PROCESS_GAS

This message is implemented by the `SetProcessGas()` client API class method. It allows the client to set the gas code (see section 4) for the gas the specified Piezocon unit should consider its *process gas*.

5.12 PIEZO_SET_CARRIER_GAS

This message is implemented by the `SetCarrierGas()` client API class method. It allows the client to set the gas code (see section 4) for the gas the specified Piezocon unit should consider its *carrier gas*.

5.13 PIEZO_SET_PRESSURE_FACTOR

This message is implemented by the `SetPressureFactor()` client API class method. It allows the client to set the pressure factor to be used by the specified Piezocon unit. For a description of *pressure factor* and other configurable parameters, see the Lorex document *Piezocon Sensor and 1-Channel Controller Istallation and User's Guide v8.09.*

5.14 PIEZO_SET_ANSWER_INDEX

This message is implemented by the `SetAnswerIndex()` client API class method. It allows the client to set the answer index to be used by the specified Piezocon unit. For a description of *answer index* and other configurable parameters, see the Lorex document *Piezocon Sensor and 1-Channel Controller Istallation and User's Guide v8.09.*

5.15 PIEZO_SET_DAC_HIGH

This message is implemented by the `SetDACHigh()` client API class method. It allows the client to set the DAC high value to be used by the specified Piezocon unit. For a description of the *DAC high value* and other configurable parameters, see the Lorex document *Piezocon Sensor and 1-Channel Controller Istallation and User's Guide v8.09.*

5.16 PIEZO_SET_DAC_LOW

This message is implemented by the `SetDACLow()` client API class method. It allows the client to set the DAC low value to be used by the specified Piezocon unit. For a description of the *DAC low value* and other configurable parameters, see the Lorex document *Piezocon Sensor and 1-Channel Controller Istallation and User's Guide v8.09.*

5.17 PIEZO_SET_MULTIPLIER

This message is implemented by the `SetMultiplier()` client API class method. It allows the client to set the concentration multiplier value to be used by the specified Piezocon unit. For a description of the *concentration multiplier value* and other configurable parameters, see the Lorex document *Piezocon Sensor and 1-Channel Controller Istallation and User's Guide v8.09.*

5.18 PIEZO_START

This message is implemented by the `Start()` client API class method. It instructs the server to place the specified Piezocon unit into *running* mode. When running, the unit continually spews the most current analysis information out its serial port. Meaningful analysis information can only be retrieved from a unit (via the `GetGasData()` or `GetAllGasData()` client methods) when the unit is running.

5.19 PIEZO_STOP

This message is implemented by the `Stop()` client API class method. It instructs the server to take the specified Piezocon unit out of *running* mode. If the unit was not running when this message is sent, an error message will be returned.

6 The Server Configuration File

The Piezocon server uses a configuration file to assign human-readable text labels to individual Piezocon units and assign them to specific RS-232 serial ports. It is also here that a unit's initial carrier and process gas configurations are defined (these can be queried and changed through the client API).

The configuration file is a simple text file. Blank lines and lines that begin with a "#" character are ignored. All other lines are device description lines with the following format.

```
    name  serialPort  carrierGasCode  processGasCode
```

where

- *name* is a short, human-readable text name the server and its clients use when referring to this specific Piezocon unit. This may be something like "Piezo1," "conc," etc.
- *serialPort* is the name of the RS-232 serial port the unit is attached to. This may be something like "/dev/ttyS0."
- *carrierGasCode* is the gas identification code (see section 4) of the gas this Piezocon unit will consider as the carrier gas.
- *processGasCode* is the gas identification code (see section 4) of the gas this Piezocon unit will consider as the process gas.

6.1 Example Configuration File

This section contains a complete example of a typical Piezocon server configuration file for a system with two Piezocon units.

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

```
# This is and example configuration file for the Piezocon server.  The
# server can manage one or more Lorex Piezocon precision binary gas
# concentration analyzers, each connected to a separate serial port.
#
############################################################################

# name  serialPort     carrierGasCode  processGasCode
conc1  /dev/ttyS0            0              22
conc2  /dev/ttyS1            0              12

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

7 Server Internals

This section will briefly look at the implementation of the Piezocon server code. For a more complete description, the reader is urged to consult the Doxygen-generated documentation for the `PiezoconServer` class, its helper classes, its client API class (`PiezoconClient`) and the associated source code modules (*piezoconServer.cpp*, *piezoconClientLib.h*, *piezocon-ClientLib.cpp*, and *piezoconMenu.cpp*).

7.1 The Need for Data Caching

Data exchange on the unit's serial interface is comparatively slow because it arrives over an RS-232 connection. Also, data arrives asynchronously—that is, it is not the result of a command/response mechanism. When data reporting is turned on (which is done by placing the Piezocon unit in its *running* mode), new concentration data is continually spewed to the device's serial port without regard to when a client requests it (including when there are no requests for it at all). In actual use, it is unlikely that new data will arrive from the Piezocon unit precisely as a client is requesting that data, and the delay between a client requesting the data and the next data becoming available can be unacceptably long.

To get around this situation, reception of new data from the Piezocon unit is decoupled from client requests for that data through the use of an internal data cache. As new data becomes available, an independent thread associated with the unit reads that data and stores it in the unit's data cache. When clients request data, the request is satisfied by immediately returning the latest values that have been written into the cache.

7.2 Threads and Mutexes

Every time a client asks the server to "start" a specified Piezocon unit (that is, put that unit into its *running* state), a new *pthread library* style thread is started to manage the data acquisition[4]. The new thread marks the unit as *running* and begins to monitor the unit's

[4]This is of course assuming that the specified unit is not already running. Requesting a "start" on a unit that is already running results in an error and no new thread is created.

associated serial port for incoming concentration analysis data. As that data arrives, it is parsed and put into the unit's internal data cache. The thread also monitors an internal state flag that gets set when a client has requested that the associated unit be "stopped". When that occurs, the thread takes steps to take the unit out of *running* mode and then exits.

One complication with threads is that they run simultaneously along with the main application and they share certain data structures in common with the main application. In the case of the Piezocon server, a situation can develop in which a client is requesting the latest cache data at the exact moment the thread is updating that cache data. If allowed to proceed, it is possible the client could receive a corrupted (only partially updated) data record. To guard against this occurrence, pthread library *mutexes*[5] are used to guarantee that the thread is completely updated before the main application can give it to a client, and that, while handing the latest cache data to a client, the thread cannot make any changes to it.

7.3 A Layered Implementation

As with most of our servers that control hardware via serial links, the Piezocon server is implemented in layers based on three primary classes. At the lowest layer, there is the `PiezoconSerialProtocol` class. This class handles the details of low-level serial communication with the Piezocon units. Next there is the `PiezoconUnit` class, which provides all the functionality needed to configure and operate a single Lorex Piezocon precision binary gas concentration analyzer. Each `PiezoconUnit` class maintains a dedicated instance of the `PiezoconSerialProtocol` class, which it uses to communicate with its associated hardware. Finally there is the `PiezconServer` class. This class is responsible for client/server message handling and high-level interaction with the Piezocon units. It maintains one instance of the `PiezconUnit` class for each unit defined in the server's configuration file. These are used to query and manipulate the individual units. In this section, we will look at each of these three components in detail. The reader is also urged to consult the Doxygen-generated documentation for these three classes.

7.3.1 The `PiezoconSerialProtocol` Class

The purpose of the `PiezoconSerialProtocol` class is to handle the low-level details involved in sending and receiving messages to/from individual Piezocon units 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 `PiezoconSerialProtocol` class is responsible for providing access to the serial port associated with a specific Piezocon unit. 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

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

settings. Conversely, when a `PiezoconSerialProtocol` object is destroyed, the serial port is automatically closed. All communication with the server's Piezocon units is ultimately accomplished through the `PiezoconSerialProtocol` class.

Because the server uses threads, and both the main-line code and these threads read and write data on the serial port, typically the serial port would be protected by a mutex to prevent the thread and the main-line code from accessing the serial port at the same time. Implementers studying the server code may wonder why the `PiezoconSerialProtocol` class does not provide a mutex for protecting the serial port. It turns out that, in this particular case, a mutex is unnecessary. That is because the server code is written in such a way that a `PiezoconUnit` object's thread is only active when its corresponding unit is in *running* mode. When in *running* mode, none of the main-line code functions attempt to use the serial port (typically they return an error instead). When the Piezocon unit is *not* in *running* mode, it has no associated thread (the thread function exits on the *running* to *not running* transition), in which case there is no possibility of contention. But it is important to keep this in mind if modifying the source code. If changes are made such that the `PiezoconUnit` methods *can* use the serial port when the thread is active, then mutex protection will have to be added to the serial port.

7.3.2 The `PiezoconUnit` Class

The `PiezoconUnit` class provides the high-level interface to a single Lorex Piezocon precision binary gas concentration analyzer. This class is responsible for initializing its associated unit on initial start up, and it provides methods for carrying out all queries and manipulations on the unit as needed by the server or the server's clients. This class also provides storage for the unit's associated data cache (see section 7.1) and the thread function (a static class method called `CommThread()`) that implements the thread that keeps the unit's local cache up to date (when the unit is in *running* mode).

Each instance of the `PiezoconUnit` class...

- has its own internal `PiezoconSerialProtocol` object that it uses to communicate with its associated Piezocon unit
- provides the storage for the local data cache associated with this unit
- starts a new copy of the `CommThread()` method in its own, detached thread any time a Piezocon unit is placed into *running* mode (this thread updates the data cache while it is active, and goes away when the unit is stopped)
- provides an internal mutex that is used to prevent the data update thread and the main server application code from accessing the local data cache at the same time

7.3.3 The `PiezoconServer` Class

The `PiezoconServer` class, derived from our standard `BaseServer` class[6], is what makes the Piezocon server application an actual server.

On initial server start up, one instance of the `PiezoconServer` class is instantiated. Its constructor processes the server's configuration file (see section 6) and instantiates `PiezoconUnit` objects for each unit defined therein.

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

7.3.4 Client Message Handlers

The server implements a series of client message handling methods—one for each server-specific client message the server can handle (see section 5).

There is not much to be said about these message handlers other than that they exist. They all provide client/server communication in the standard way, as described in design document ***DES-0005*, *The Client/Server Architecture***.

7.3.5 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 Piezocon server has legitimate SoH data to return for each unit 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 unit managed by the server, we return two pieces of SoH information: the unit's current reported gas concentration and its current reported gas temperature. So reimplementing `GetNumSohParams()` is easy. We simply return the total number of units being maintained by the server multiplied by two. For the next two functions, we just call the `_GetAllGasData()` method[7] to receive an STL map of `ReturnGasData` structures containing the current reported information for all units managed by the server. We then traverse this

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

[7]This is the server equivalent of the client message handler by the same name. Like the client version, it returns an STL map of `ReturnGasData` structures. The only difference is that the server version does not have any of the client/server messaging code.

map and build appropriate responses based on what values it contains. The exact fields and format of the response messages are beyond the scope of this document, but they are covered in section 5 of ***DES-0005*, *The Client/Server Architecture***, and in design document ***DES-0006*, *The State-of-Health Server***. Please refer to those documents and the source code for further details.

8 The Test Menu Program

For development and testing purposes, a small text-mode menu client application called *piezoconMenu*, (see *piezoconMenu.cpp*) has been developed for the Piezocon server. The program uses the `PiezoconClient` class to provide the client API the server supports. It also maintains a client interface to the system event logger[8] 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

Piezocon Server Specific Items:
  1 - Get Number of Sensors
  2 - Get Process Gas
  3 - Get Carrier Gas
  4 - Get Pressure Factor
  5 - Get Answer Index
  6 - Get DAC High
  7 - Get Dac Low
  8 - Get Multiplier
  9 - Get Gas Data
 10 - Get All Gas Data

 11 - Set Process Gas
 12 - Set Carrier Gas
 13 - Set Pressure Factor
 14 - Set Answer Index
 15 - Set DAC High
 16 - Set DAC Low
 17 - Set Multiplier
 18 - Start Unit
 19 - Stop Unit

  0 - Exit Program

Enter Selection >
```

[8]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[9]. Following this are nineteen items that correspond to the client messages provided by the `PiezoconClient` client API class. 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 when running on real-world systems.

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