U.S. DEPARTMENT OF ENERGY

PNNL-22598

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

The Paroscientific Digiquartz 6000 Server

DES-0026

Revision 1

Charlie Hubbard

July 2012

Pacific Northwest
NATIONAL LABORATORY

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

The Paroscientific Digiquartz 6000 Server

Charlie Hubbard

DES-0026
Revision 1
July 2012

Contents

1 Introduction

This document describes a hardware interface server application designed to manage one or more *Paroscientific Digiquartz® 6000* high-precision pressure sensors. Each managed sensor is connected to the host computer (the computer on which the server runs) through a separate RS-232 serial interface. Each managed sensor is assigned a short, human-readable, text name by which the sensor is known to the server and its clients. Serial ports and sensor names are assigned in the server's configuration file (see section 5). Assigned sensor names must be unique across the entire project.

The server is implemented by the *digi6000Server.cpp* source module. The client interface to the server is defined and implemented by *digi6000ClientLib.h* and *digi6000ClientLib.cpp* respectively. A standard text-mode test menu client for the server is implemented by *digi6000Menu.cpp* (see section 7).

The primary documentation for the Digiquartz 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 APIs. If contradictions between this document and the Doxygen-generated documentation are found, the Doxygen-generated documentation should be considered correct.

2 The Digiquartz Hardware

The Paroscientific Digiquartz® series 6000 pressure sensor is a high-precision, temperature compensated, intelligent pressure transducer available in a variety of absolute and gauge pressure ranges. Measurement accuracy is typically better than 0.01% full-scale, with a 1 ppm measurement precision at one-second sampling intervals. The sensor features an integrated microcontroller with RS-232 and RS-485 serial communication interfaces that a control application (in this case, the Digi-6000 server this document describes) can use to configure and read the sensor.

3 Serial Communications

The pressure sensor can communicate using either RS-232 or RS-485 using a variety of baud rates and data formats. From the factory, sensors are configured for RS-232, with an 8-bit data word, 1 stop bit, no parity, operating at 9600 baud. Because RS-485 can support multiple devices on the same serial bus, each pressure sensor must be assigned an address so that individual devices on the same bus can be accessed independently of the others. The factory default address is 01. Although RS-232 does not support multiple sensors sharing the same serial lines, devices still require an address, if for no other reason than to make the messaging protocol consistent between the two serial protocols. In the messaging protocol, the host computer always has address 00.

Sensors communicate via a simple, ASCII protocol with the following format[1]:

```
*SSDDcc<data>CRLF
```

where...

- `*` is the string literal "*" (all messages begin with an asterisk character)
- `SS` is the two-digit address of the sender of the message
- `DD` is the two-digit address of the intended receiver
- `cc` is a two letter command tag. These are case sensitive and must always be capitalized. In reply messages returned from a sensor, the `cc` characters may be replaced with response data.
- `data` is any additional data associated with the command
- `CRLF` is a carriage-return / line-feed pair. All messages are terminated with this sequence.

Message generation and reply handling will be discussed in greater detail in section 6.

4 The Client API

The Digi-6000 server, like all servers written for our client/server architecture[2], 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 Digi-6000 server is called `Digi6000Client` and it is defined and implemented in *digi6000ClientLib.h* and *digi6000ClientLib.cpp* respectively.

Like all servers that use our standard client/server architecture, the Digi-6000 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[3].

[1] Much more information on the messaging protocol can be found in the Paroscientific Digiquartz Programming and Operation manual.

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

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

In addition to these standard messages, the server can accept and respond to three additional *client-specific* messages defined by the server's associated client API. These are described in detail in this section[4].

4.1 DIGI_GET_NUM_SENSORS

This message is implemented by the `GetNumSensors()` client API class method. It simply returns the number of Digiquartz series 6000 pressure sensors the server is managing.

4.2 DIGI_GET_SENSOR

This message is implemented by the `GetSensor()` client API class method. It returns a structure of type `DigiStatusRecord` (defined in *digi6000ClientLib.h*), that contains current information about the specified sensor. This information includes the sensor's most recently measured temperature and pressure values.

4.3 DIGI_GET_ALL_SENSORS

This message is implemented by the `GetAllSensors()` client API class method. It returns an STL map of `DigiStatusRecord` structures containing one entry for each sensor managed by the server. The map is keyed on sensor name.

5 The Server Configuration File

The Digi-6000 server uses a configuration file to assign human-readable text labels to individual sensor units and assign them to specific RS-232 serial ports. The configuration file is a simple text file. Blank lines and lines that begin with a '#' character are ignored. All other lines are sensor definition lines. Sensor definition lines have the following format.

```
    name port
```

where

- *name* is a short, human-readable text name the server and its clients use when referring to this specific sensor. This may be something like "PS101," "INLET," etc.
- *port* is the name of the RS-232 serial port the sensor is attached to. This may be something like "/dev/ttyS0."

[4]The symbolic constants that comprise the following subsection headers come from the file *digi6000ClientLib.h*. Please review the Doxygen documentation for that file for more information.

5.1 Example Configuration File

This section contains a complete example of a typical Digi-6000 server configuration file for a system with two Digiquartz series 6000 pressure sensors.

```
################################################################################
#
# This is and example configuration file for the Digi-6000 server.  The
# server can manage one or more Digi-6000 precision pressure sensors, each
# connected to a separate serial port.
#
################################################################################

INLET /dev/ttyS1
PS600 /dev/ttyMXUSB3

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

6 Server Internals

In this section we'll briefly look at the implementation of the Digi-6000 server code. For a more complete description, the reader is urged to consult the Doxygen-generated documentation for the `Digi6000Server` class, its helper classes, and the server's source code in the file *digi6000Server.cpp*.

6.1 The Need for Data Caching

Data exchange on the sensor's serial interface is comparatively slow. When many clients are requesting current pressure data from a particular sensor at one time (as can be the case if many GUIs are active along with external status reporters, SoH loggers, etc.), the serial interface can become a bottleneck that can slow down the operation of the entire system.

To prevent that from happening, a data-caching mechanism is employed. Specifically, each sensor has a data update thread associated with it that reads the sensor's current values once per second and stores them in an internal cache structure. Clients requests are then satisfied with data from this cache rather than querying the sensor directly. In this way, clients never have to wait for data to arrive on a slow serial connection, and each sensor's serial connection is burdened with a traffic load it can easily handle regardless of the number of requesting clients.

6.2 A Layered Implementation

As with most of our servers that control hardware via serial links, the Digi-6000 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 `Digi6000Unit` class. This class provides all the functionality to configure and operate a single Digiquartz series 6000 pressure sensor. Each `Digi6000Unit` class maintains a dedicated instance of the `SerialProtocol` class, which it uses to communicate with its associated pressure sensor. Finally there is the `Digi6000Server` class. This class is responsible for client/server message handling and high level interaction with the pressure sensors. It maintains one instance of the `Digi6000Unit` 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.

6.2.1 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 Digiquartz series 6000 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 is done through the `SerialProtocol` class.

Because the serial port is one of those resources that can potentially be accessed both by the sensor's data update thread and the main server application at the same time, it requires mutex protection[5]. The `SerialProtocol` class contains an internal mutex member variable (cleverly named '`mutex`') that is used to protect the serial port. 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 by locking the class mutex just before sending the command (or query) and not unlocking it again until the response has been completely received.

[5]Actually, as of this writing, the main line server thread only communicates directly with the sensors during initialization. That always takes place *before* the data update thread is started and then never again. So technically, the serial port doesn't currently *require* mutex protection. However, it was thought better to provide mutex protection in the initial implementation anyway. That way future modifications that do allow the main line thread to communicate with the sensors after initialization won't inadvertently introduce a highly intermittent, and very difficult to track down thread contention bug.

As input, the `SendReceiveCommand()` method takes a string consisting of a valid two-letter command code along with any associated data, and returns (through the parameter list) a stripped down response string. As mentioned in section 3, messages always begin with an asterisk character followed by a four-character addressing prefix, and messages are always terminated with a carriage-return/line-feed pair. These elements are automatically added to the command string by the `SendReceiveCommand()` method and they are automatically stripped off the sensor's returned response. Callers should not include these themselves, nor should they expect them to appear in the response string.

The `SendReceiveCommand()` method also implements a retry-on-fail scheme, whereby a command or query sent to a sensor will automatically be resent (multiple times if necessary) if the sensor fails to respond or responds with an error.

6.2.2 The `Digi6000Unit` Class

The `Digi6000Unit` class provides the high-level interface to a single Digiquartz series 6000 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 storage for the sensor's associated data cache (see section 6.1) and the thread function (a static class method called `CommThread()`) that implements the thread that keeps the sensor's local cache up-to-date.

Each instance of the `Digi6000Unit` class...

- has its own internal `SerialProtocol` object which 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 `CommThread()` 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 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

Finally, the Digi-6000 server's *simulator mode* is also implemented at the `Digi6000Unit` 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 `Digi6000Unit` 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 preprocessor 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[6].

In several places in the source code implementing the `Digi6000Unit` class, one will find the following pattern:

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

   do real stuff
   ...

#else
   // we are in simulator mode

   do simulated stuff
   ...

#endif
```

While in simulator mode, the `Digi6000Unit` class' data query method is unaffected. It continues to return values out of the class' local cache structure as it would in real mode. 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.

6.2.3 The `Digi6000Server` Class

The `Digi6000Server` class, derived from our standard `BaseServer` class[7], is what makes the Digi-6000 server application an actual server.

On initial server start up, one instance of the `Digi6000Server` class is instantiated. It's constructor processes the server's configuration file (see section 5) and instantiates `Digi6000Unit` objects for each sensor defined therein.

The `Digi6000Server` class maintains an STL map, called `nameUnitMap`, that maps the short text labels by which individual sensors are known to the specific `Digi6000Unit` 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 `Digi6000Unit` object.

[6]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.

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

6.2.4 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 4).

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***.

6.2.5 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 Digi-6000 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 two pieces of SoH information information: the sensor's current reported temperature and its current reported pressure. So reimplementing `GetNumSohParams()` is easy. We simply return the total number of sensors being maintained by the server multiplied by two. For the next two functions, we just call the `_GetAllSensors()` method[8] to receive an STL map of `DigiStatusRecord` structures containing the current reported information for all sensors managed by the server. We then traverse this 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.

7 The Test Menu Program

For development and testing purposes, a small text-mode menu client application called *digi6000Menu*, (see *digi6000Menu.cpp*) has been developed for the Digi-6000 server. The program uses the `Digi6000Client` class to provide the client API the server supports. It also maintains a client interface to the system event logger[9], so it can record when it starts up and when it shuts down.

[8]This is the server equivalent of the client message handler by the same name. Like the client version, it returns an STL map of `DigiStatusRecord` structures. The only difference is, the server version doesn't have any of the client/server messaging code.

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

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

Digi-6000 Server Specific Items:
  1 - Get Number of Sensors
  2 - Get Data for One Sensor
  3 - Get Data for All Sensors

  0 - Exit Program

Enter Selection >
```

The first seven menu items correspond to standard messages that all servers can support[10]. Following this are three items that correspond to the three client messages provided by the `Digi6000Client` 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.

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