U.S. DEPARTMENT OF ENERGY

PNNL-22597

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

The Opto-22 SNAP PAC Server

DES-0021

Revision 2

Charlie Hubbard

July 2012

Pacific Northwest
NATIONAL LABORATORY

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

The Opto-22 SNAP PAC Server

Charlie Hubbard

DES-0021
Revision 2
July 2012

DISCLAIMER

PACIFIC NORTHWEST NATIONAL LABORATORY
operated by
BATTELLE
for the
UNITED STATES DEPARTMENT OF ENERGY
under
Contract DE-AC05-76RL01830

Printed in the United States of America

Available to DOE and DOE contractors from the Office of Scientific and Technical Information, P.O. Box 62, Oak Ridge, TN 37831-0062
ph: (865) 576-8401
fax: (865) 576-5728
email: reports@adonis.osti.gov

Available to the public from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Rd., Springfield, VA 22161
ph: (800) 553-6847
fax: (703) 605-6900
email: orders@ntis.fedworld.gov
online ordering: http://www.ntis.gov/ordering.htm

Contents

1 Introduction

This document describes the *Opto-22 SNAP PAC* server application. This server amounts to a user-mode device driver capable of managing one or more Opto-22 SNAP PAC brainboards and their associated backplanes and hardware interfacing modules. It is implemented by the *optoServer.cpp* source module. The server actually implements two separate client APIs; one for digital I/O and one for analog I/O. These two APIs are defined by *analogClientLib.h* and *digitalClientLib.h* respectively, and are implemented by *analogClientLib.cpp* and *digitalClientLib.cpp*. A standard text-mode test menu client for the server is implemented by *anadigMenu.cpp*. Like the server itself, the menu client supports both client APIs.

The primary documentation for the Opto-22 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.

The Opto-22 server is not intended to provide an exhaustive interface to all of the Opto-22 SNAP PAC brainboard's capabilities. Instead, only that subset of the controller's functionality that we need for our current projects is supported. At present, that means the ability to directly set and read analog and digital input and output channels. Currently there are no provisions to take advantage of advanced features of the brainboard such as latching (of digitial inputs), averaging (of analog inputs), on-board conversion to/from engineering units, etc., and there is no expectation that these features will be implemented in the future.

2 Opto-22 Hardware

Opto-22's *SNAP PAC* hardware interfacing product[1] consists of a 4, 8, 12 or 16 slot backplane into which various digital and/or analog I/O modules can be installed. Also attached to the backplane is an electronic control module that Opto-22 calls a *brainboard*. The brainboard is the unit that can communicate directly with the I/O modules on the backplane. External applications, including the Opto-22 server, interact with the digital and analog channels indirectly by communicating with the brainboard over an Ethernet connection.

Opto-22 offers a large number of analog and digital I/O modules compatible with the SNAP PAC system. These offer varying numbers of channels per module, and also offer a wide range of electrical options (the ability to directly switch A/C and D/C signals of varying voltages, in both negative and positive logic configurations, voltage input and output modules for varying ranges, bipolar analog input and output modules, support for 4-to-20 mA current loops, thermocouple inputs, etc.). For the most part, the details of dealing with the different

[1] Please see the manuals for the various Opto-22 components at Opto-22's website at http://www.opto22.com/site/snap_pac_system.aspx for full details on the SNAP PAC architecture.

module types are handled by the Opto-22 brainboard. The Opto-22 server is compatible with most or all of the module types.

3 Brainboard Communications

As mentioned previously, the Opto-22 server communicates with the SNAP PAC hardware via Ethernet connection to the brainboard. More specifically, each brainboard is assigned a unique IP address using an Opto-22–provided, Windows-based configuration utility before it is installed in the system. The Opto-22 server then communicates with the brainboards by sending/receiving command/response message pairs bundled into UDP packets directed at UDP port 2001 at the IP address of the specific brainboard involved. This low-level communication between the server and the hardware is actually accomplished by means of an Opto-22–provided software library, provided in source code form (C++). The source files that comprise the Opto-22 communication library are as follows: *O22SIOMM.cpp*, *O22SIOMM.h*, *O22SIOST.cpp*, *O22SIOST.h*, *O22SIOUT.cpp*, *O22SIOUT.h*, *O22STRCT.h*, *Opto22EthernetIO.cpp*, and *Opto22EthernetIO.h*. All aspects of socket management, packet building, transmission, reception and packet parsing are handled by the Opto-22 communication library.

Unfortunately, Opto-22 has not updated their comm library since 2003. In the intervening years, new, higher-channel-count digital and analog modules have been introduced that the library is not equipped to communicate with. For this reason, a number of small enhancements have been made to the existing library source code by PNNL staff to support these newer modules. Changes are limited to the files *O22SIOMM.h* and *O22SIOMM.cpp*, and they are thoroughly described in the source code comments. Most of these comm library changes have to do with new channel-addressing modes that have been added by Opto-22 to handle the higher-channel-count modules. Much more will be said about channel-addressing modes during the discussion of the server's configuration file (see section 6) and server design internals (see section 7.1.2).

4 The Client APIs

The Opto-22 server, like all servers written for our client/server architecture[2], relies on client API classes 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.

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

Typically, client API classes are written specifically for the servers they support. However, for the Opto-22 server, exactly the opposite is true. The server has been specifically written to support two pre-existing client API classes – one that deals with digital signals and one that deals with analog signals.

Perhaps the most important design attribute of these two APIs is that they are *generic*. That is, they make no assumptions about the underlying hardware technology (in this case, Opto-22 SNAP PAC electronics). Because no details of the underlying hardware are visible through the client interface, client writers do not need to concern themselves with the complexities of communicating with Opto-22 electronics, or indeed, even know that Opto-22 electronics are being used. This has important ramifications to the project code, because it means the underlying A/D I/O technology can easily be changed at a later date without affecting any of the clients that require A/D I/O. If the underlying technology changes, all that changes is the server with which the clients interact.

Other important attributes of the client APIs include

- Individual analog and digital signals are referenced by short, human-readable text names. For example, a digital output that controls a valve may be called "V203" or "vent," whereas an analog input signal that reads the output of a pressure sensor may be assigned a name like "PS101" or "vacuum." This mapping between text labels to physical channels in the underlying electronics is made in the Opto-22 server's configuration file (see section 6).
- Digital channels are logic neutral. That is to say, clients have no way of knowing, nor any reason to care whether an underlying digital signal uses negative logic (low voltage means "on") or positive logic (low voltage means "off"). This detail is also handled by the server's configuration file.
- Analog channels take and report values specified in engineering units appropriate to the hardware to which those analog channels are attached. As an example, consider a hardware pressure transducer that outputs a current in the range of 4 to 20 mA, representing a pressure range of 0 to 100 PSIA. A reading of 12 mA then corresponds to a pressure of 50 PSIA. When the client requests the value of this analog channel, the value returned is 50.0 PSIA, not 12 mA. Not only does the client not need to perform the electrical-to-engineering conversion, the client doesn't even need to know or care that the underlying sensor is a 4-20 mA sensor (as opposed to a 0-10V or 0-100mV sensor say). Again, the details for this feature (various conversion coefficients) exist in the server's configuration file.
- And of course, clients have no way of knowing (and no reason to care) which Opto-22 brainboard is actually responsible for what specific analog or digital channel, or what Opto-22 module the channel exists on. This is important because it means that from a wiring perspective, signals can later be moved to different channels if necessary without affecting existing clients in any way. Here is one example of how this might occur during development: say a system initially has 12 valves controlled by three 4-channel Opto-22 digital output modules. Later, a modification to the system requires the addition of two more valves, but there is no room on the backplane to support another digital

module. Instead, the existing three 4-channel modules are removed and replaced with one 16-channel module, and the existing 12 valves are then wired to the new module, and the two new valves are as well. These changes are absolutely invisible to client applications, meaning no code changes to the clients are required; therefore no new bugs are inadvertently introduced.

5 Client-Specific Messages

Like all servers that use our standard client/server architecture, the Opto-22 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 Opto-22 server does fully support the standard state-of-health reporting mechanism implemented by the `BaseServer` class[3].

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 APIs the server supports. The Opto-22 server supports two client APIs (implemented by the `AnalogClient` and `DigitalClient` classes respectively) that together define 13 additional client messages. These are described in detail in this section[4].

5.1 ANA_GET_NUM_ANALOG_IN_CHANNELS

This message, implemented by the `AnalogClient::GetNumInChannels()` method, returns the total number of analog input channels being managed by the server.

5.2 ANA_GET_NUM_ANALOG_OUT_CHANNELS

This message, implemented by the `AnalogClient::GetNumOutChannels()` method, returns the total number of analog output channels being managed by the server.

5.3 ANA_GET_ANALOG

This message, implemented by the `AnalogClient::Get()` method, returns a data record[5] containing the current value of the specified analog input or output channel, along with

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

[4] The symbolic constants that comprise the following subsection headers come from the files *analogClientLib.h* and *digitalClientLib.h* respectively. Please review the Doxygen documentation for those files for more information.

[5] See the Doxygen documentation for the `AnalogChannelRecord` struct for more details on the returned data record.

many other details about the channel. Keep in mind that the returned value is specified in engineering units appropriate to the sensor or control to which the channel is attached and not in the native electrical units (volts or mA, say) of the underlying hardware.

5.4 ANA_GET_ANALOG_ALL

This message, implemented by the `AnalogClient::GetAll()` method, returns an STL map containing data records fore every analog input and output channel being managed by the server.

5.5 ANA_SET_ANALOG

This message, implemented by the `AnalogClient::Set()` method, tells the server to set the specified analog output channel to the specified value. As always, the value is specified in engineering units appropriate to the device the channel is controlling.

5.6 ANA_SET_ANALOG_INPUT

This message, implemented by the `AnalogClient::SetInput()` method, tells the server to report back the specified value when clients query the specified analog input channel. This message is only valid if the server is in simulator mode. Otherwise an appropriate error message is returned. See sections 7.1.4 and 7.2 for more details.

5.7 ANA_SET_GAIN_OFFSET

As has been mentioned before, the analog client API always deals in engineering units appropriate to the device a specific analog channel is connected to. It does not deal in the native units of the underlying electrical signal (volts say, for an analog input module with 0-10VDC inputs). The conversion between native units and engineering units is performed automatically by the server using a *gain* and *offset* value[6] (see sections 6.3.2 and 6.3.3 for a more detailed discussion of converting between native and engineering units).

The conversion is straightforward. When converting from a native value N (in volts say) to engineering units E (say in kilograms), the following equation is used:

E = N * gain + offset.

The gain and offset values to be used for each analog channel are stored in the server's configuration file (see section 6). Normally these values do not need to be changed. However, there are certain times (calibrating sensors for example) when it is convenient to change these values directly through the client interface.

[6]The reader may be more familiar with these terms referred to as a *slope* and an *intercept* value.

This client message, implemented by the `AnalogClient::SetGainOffset()` method, provides a means to do just that. When changes are made to an analog channel's gain and offset values, the changes take effect immediately. However, the server's configuration file is also automatically updated with the changes, so that the next time the server is started, the most recent values will be used.

The reader may notice that there is no explicit client message for retrieving the current gain and offset values set on a channel. These values are returned as part of the `AnalogChannelRecord` structure returned by the client's `Get()` and `GetAll()` methods.

5.8 DIG_GET_NUM_DIGITAL_IN_CHANNELS

This message, implemented by the `DigitalClient::GetNumInChannels()` method, returns the total number of digital input channels being managed by the server.

5.9 DIG_GET_NUM_DIGITAL_OUT_CHANNELS

This message, implemented by the `DigitalClient::GetNumOutChannels()` method, returns the total number of digital output channels being managed by the server.

5.10 DIG_GET_DIGITAL

This message, implemented by the `DigitalClient::Get()` method, returns a data record[7] containing the current state (asserted or not asserted) of the specified digital input or output channel.

5.11 DIG_GET_DIGITAL_ALL

This message, implemented by the `DigitalClient::GetAll()` method, returns an STL map containing data records fore every digital input and output channel being managed by the server.

5.12 DIG_SET_DIGITAL

This message, implemented by the `DigitalClient::Set()` method, tells the server to set the specified digital output channel to the specified state (asserted or not asserted).

[7] See the Doxygen documentation for the `DigitalChannelRecord` struct for more details on the returned data record.

5.13 DIG_SET_DIGITAL_INPUT

This message, implemented by the `DigitalClient::SetInput()` method, tells the server to report back the specified value when clients query the specified digital input channel. This message is only valid if the server is in simulator mode. Otherwise an appropriate error message is returned. See sections 7.1.4 and 7.2 for more details.

6 The Server Configuration File

The Opto-22 server uses a configuration file to map individual digital and analog channels to a specific Opto-22 brainboard, to assign human-readable text labels to individual channels, to set logic sense (negative or positive) on digital channels, to define native to engineering unit-conversion coefficients on analog channels, and so on. When the server initially starts, it parses its configuration file, establishes communication links to the brainboards defined therein, and configures individual I/O channels as necessary.

6.1 Module Types

The Opto-22 server application supports six different kinds of Opto-22 SNAP PAC hardware interfacing modules. They are:

1. Low-density digital input modules – These are modules that contain between one and four digital input channels.
2. High-density digital input modules – These are modules that contain more than four digital input channels. The distinction between these and their low-density counterparts has to do with the way the brainboard manages the modules. High-density modules are newer technology. They are internally addressed differently by the server and, unlike low-density modules, they do not need to be manually configured. The brainboard automatically detects and configures high-density modules.
3. Low-density digital output modules – These are modules that contain between one and four digital output channels.
4. High-density digital output modules – These are modules that contain more than four digital output channels. The distinction between these and their low-density counterparts has to do with the way the brainboard manages the modules. High-density modules are newer technology. They are internally addressed differently by the server and, unlike low-density modules, they do not need to be manually configured. The brainboard automatically detects and configures high-density modules.
5. Analog input modules
6. Analog output modules

6.2 Channel Addressing

The actual physical channel addressing used by the Opto-22 brainboard for the various digital and analog channels on the modules attached to an Opto-22 SNAP PAC backplane is, for historical reasons, somewhat complicated. To the extent possible, these nuances have been hidden from the writers and maintainers of the server's configuration file. In the configuration file, each channel is specified by the module number (that is to say, the backplane slot number) of the module that hosts the channel, and the channel number of the channel relative to its module.

As an example, if a particular digital channel is channel 11 of a 16-channel digital output module installed into backplane slot 5, then that channel's address is fully specified in the configuration file with two parameters: 5 and 11.

This addressing scheme (module number/channel number) is used for *all* channels defined in the configuration file, regardless of how these channels are actually addressed when the server communicates with an Opto-22 brainboard. Hiding the details of the actual addressing modes makes the configuration file much easier to write and maintain. More will be said about the underlying, module-specific addressing modes used internally by the server in section 7.1.2.

6.3 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[8]. In the text file, blank lines and lines that begin with a "#" character are ignored. All other lines are relevant. There are seven valid line types, which are described below.

6.3.1 Brainboard Definition Lines

Brainboard definition lines have the following syntax.

```
@<dotted-decmial-ip>
```

As an example...

```
@192.168.1.100
```

[8] As mentioned previously, the server itself will also update the current configuration file any time a client requests changes to the coefficients an analog channel uses to convert between native units (like volts) to engineering units (like Torr).

The "@" character at the beginning of the line denotes this as a brainboard definition line. The IP address that follows means that all subsequent lines in the file, up to but not including the next brainboard definition line, pertain to signals that are managed by the Opto-22 SNAP PAC brainboard located at this IP address. There will be as many brainboard definition lines in the file as there are brainboards in the system (frequently there is just one).

6.3.2 Analog Input Definition Lines

There is one analog input definition line for every analog input channel in the system that is actually wired to hardware. That is to say, unused analog input channels (extra channels on a module that aren't currently being used) do not have to be specified in the server's configuration file (although doing so is not harmful). Analog input definition lines have the following syntax:

```
<name> ai <modnum> <channum> <pointtype> <lower> <upper> <gain> <offset> <units> <description>
```

- *name* – This is a short, human-readable text name that clients will use to reference this channel. This name should be globally unique across all servers running on the system.
- *ai* – This is the string literal "ai." This is merely a tag that indicates what type of channel is being described by this line (an analog input). It is mostly a convenience to the parser. The string is case-sensitive and must be all lowercase.
- *modnum* – This field contains the backplane slot number that the module hosting this channel is installed into. This is one of the two fields that implement the module number/channel number addressing scheme described in section 6.2.
- *channum* – This is the number of the channel relative to the module that hosts the channel. For example, a 16-channel analog input module has channels that are numbered 0 through 15. This is the other of two fields that implement the module number/channel number addressing scheme described in section 6.2.
- *pointtype* – This is a small (typically two- or three-digit) numerical code defined by Opto-22 that indicates how this channel should be interpreted. Typically all analog input channels on a given module use the same point-type number, but that doesn't have to be the case. One example of where individual channels on a given module might use different point-type numbers is in the case of thermocouple modules. Opto-22 thermocouple modules are capable of supporting a wide variety of thermocouple types. In this case, the exact point-type code specifies what type of thermocouple is connected to a specific channel. Point-type codes can be found in Opto-22 document **1465**, **OptoMMP Protocol Guide**, starting on page 21.
- *lower* – This defines the lower limit the channel can report specified in native units. For example, for an analog input module that supports bipolar inputs with an input range of between -10.0 and 10.0 volts, this field would be -10.0.

- *upper* – This defines the upper limit the channel can report specified in native units. For example, for an analog input module that supports bipolar inputs with an input range of between -10.0 and 10.0 volts, this field would be 10.0.
- *gain* – This is one of two parameters used to convert between a module's native units and appropriate engineering units for the channel (the other parameter being the *offset* parameter defined below). The conversion equation is simple.

  Engineering = Native * gain + offset.

  As an example, consider a 4-20 mA analog input channel that is connected to a pressure sensor that provides a 4-20 mA current output corresponding to a pressure range of 0 to 160 PSI. The gain and offset values are found by solving the following two simultaneous equations:

  0 PSI = 4 mA * gain + offset
  160 PSI = 20 mA * gain + offset.

  This yields the following values:

  gain = 10 PSI/mA
  offset = -40 PSI.

  Let's test these out. Assume that the pressure sensor is reading 12 mA. That's the midway point of the range 4-20 mA, so it should correspond to the midway point of the sensor's pressure range as well (80 PSI). Does it?

  Engineering = 12 * 10 - 40 = 80 PSI.
- *offset* – This is the other of the two parameters used to convert between a module's native units and appropriate engineering units for the channel. See the *gain* field described above for details.
- *units* – This is a short text string that names the engineering units in which the sensor connected to this input channel reports. For the example described above, this field would contain the string "PSI" (without the quotes).
- *description* – This is a free-form text description of the sensor. In our running example, this might be something like "Primary pump outlet pressure" (without the quotes).

6.3.3 Analog Output Definition Lines

There is one analog output definition line for every analog output channel in the system that is actually wired to hardware. That is to say, unused analog output channels (extra channels on a module that aren't currently being used) do not have to be specified in the server's configuration file (although doing so is not harmful). Analog output definition lines have the following syntax.

```
<name> ao <modnum> <channum> <pointtype> <lower> <upper> <gain> <offset> <units> <description>
```

- *name* – This is a short, human-readable text name that clients will use to reference this channel. This name should be globally unique across all servers running on the system.
- *ao* – This is the string literal "ao." This is merely a tag that indicates what type of channel is being described by this line (an analog output). It is mostly a convenience to the parser. The string is case-sensitive and must be all lowercase.
- *modnum* – This field contains the backplane slot number into which the module hosting this channel is installed. This is one of the two fields that implement the module number/channel number addressing scheme described in section 6.2.
- *channum* – This is the number of the channel relative to the module that hosts the channel. For example, a 16 channel analog output module has channels that are numbered 0 through 15. This is the other of the two fields that implement the module number/channel number addressing scheme described in section 6.2.
- *pointtype* – This is a small (typically two- or three-digit) numerical code defined by Opto-22 that indicates how this channel should be interpreted. Typically all analog output channels on a given module use the same point-type number. Point-type codes can be found in Opto-22 document **1465**, **OptoMMP Protocol Guide**, starting on page 21.
- *lower* – This defines the lower limit the channel can output specified in native units. For example, for an analog output module that supports unipolar outputs with a range of between 0.0 and 10.0 volts, this field would be 0.0.
- *upper* – This defines the upper limit the channel can output specified in native units. For example, for an analog output module that supports unipolar outputs with a range of between 0.0 and 10.0 volts, this field would be 10.0.
- *gain* – This is one of two parameters used to convert between a module's native units and the appropriate engineering units for the channel (the other parameter being the *offset* parameter defined below). The conversion equation is simple.

  Engineering = Native * gain + offset.

  As an example, consider a mass flow controller with a range of 0 to 100 ml/min whose set point is programmed with a voltage in the range of 0 to 5 volts (that is, outputting 0 V will result in zero flow, and output 5 V will result in a flow rate of 100 ml/min) The gain and offset values are found by solving the following two simultaneous equations...

  0 ml/min = 0 V * gain + offset
  100 ml/min = 5 V * gain + offset

  This yields the following values:

gain = 20 ml/min/V
offset = 0 ml/min

- *offset* – This is the other of the two parameters used to convert between a module's native units and appropriate engineering units for the channel. See the *gain* field described above for details.
- *units* – This is a short text string that names the engineering units in which the device connected to this output channel reports. For the example described above, this field would contain the string "ml/min" (without the quotes).
- *description* – This is a free-form text description of the sensor. In our running example, this might be something like "Sample inlet flow rate" (without the quotes).

6.3.4 Low-density Digital Input Definition Lines

Opto-22 makes *low-density* digital input modules (simply, modules with four or fewer channels) and *high-density* digital input modules (modules with more than four channels). The difference between the two has to do with how channels are addressed and whether or not modules have to be explicitly configured with a point-type code (low-density modules only). The following describes the format of definition lines for *low-density* digital inputs.

There is one digital input definition line for every digital input channel in the system that is actually wired to hardware. That is to say, unused channels (extra channels on a module that aren't currently being used) do not have to be specified in the server's configuration file (although doing so is not harmful). Low-density digital input definition lines have the following syntax.

```
<name> di <modnum> <channum> <pointtype> <logic> <description>
```

- *name* – This is a short, human-readable text name that clients will use to reference this channel. This name should be globally unique across all servers running on the system.
- *di* – This is the string literal "di." This is merely a tag that indicates what type of channel is being described by this line (a low-density digital input). It is mostly a convenience to the parser. The string is case-sensitive and must be all lowercase.
- *modnum* – This field contains the backplane slot number that the module hosting this channel is installed into. This is one of the two fields that implement the module number/channel number addressing scheme described in section 6.2.
- *channum* – This is the number of the channel relative to the module that hosts the channel. For example, a 4-channel low-density digital input module has channels numbered 0 through 3. This is the other of the two fields that implement the module number/channel number addressing scheme described in section 6.2.

- *pointtype* – This is a small (typically two- or three-digit) numerical code defined by Opto-22 that indicates how this channel should be interpreted. Typically, all digital input channels on a given module use the same point-type number. Point-type codes can be found in Opto-22 document **1465**, **OptoMMP Protocol Guide**, starting on page 21.
- *logic* – This field contains either a "-" character (indicating this is a negative logic channel) or a "+" character (indicating this is a positive logic channel).
- *description* – This is a free-form text description of the sensor. For a digital input channel, this might be something like "Over-temperature trip indication" (without the quotes).

6.3.5 High-density Digital Input Definition Lines

Opto-22 makes *low-density* digital input modules (simply, modules with four or fewer channels) and *high-density* digital input modules (modules with more than four channels). The difference between the two has to do with how channels are addressed and whether modules have to be explicitly configured with a point-type code (low-density modules only). The following describes the format of definition lines for *high-density* digital inputs.

There is one digital input definition line for every digital input channel in the system that is actually wired to hardware. That is to say, unused channels (extra channels on a module that aren't currently being used) do not have to be specified in the server's configuration file (although doing so is not harmful). High-density digital input definition lines have the following syntax:

```
<name> hdi <modnum> <channum> <logic> <description>
```

- *name* – This is a short, human-readable text name that clients will use to reference this channel. This name should be globally unique across all servers running on the system.
- *hdi* – This is the string literal "hdi." This is merely a tag that indicates what type of channel is being described by this line (a high-density digital input). It is mostly a convenience to the parser. The string is case-sensitive and must be all lowercase.
- *modnum* – This field contains the backplane slot number that the module hosting this channel is installed into. This is one of the two fields that implement the module number/channel number addressing scheme described in section 6.2.
- *channum* – This is the number of the channel relative to the module that hosts the channel. For example, a 32-channel high-density digital input module has channels that are numbered 0 through 31. This is the other of the two fields that implement the module number/channel number addressing scheme described in section 6.2.

- *logic* – This field contains either a "-" character (indicating this is a negative logic channel) or a "+" character (indicating this is a positive logic channel).
- *description* – This is a free-form text description of the sensor. For a digital input channel, this might be something like "Over-temperature trip indication" (without the quotes).

6.3.6 Low-density Digital Output Definition Lines

Opto-22 makes *low-density* digital output modules (simply, modules with four or fewer channels) and *high-density* digital output modules (modules with more than four channels). The difference between the two has to do with how channels are addressed and whether or not modules have to be explicitly configured with a point-type code (low density modules only). The following describes the format of definition lines for *low-density* digital outputs.

There is one digital output definition line for every digital output channel in the system that is actually wired to hardware. That is to say, unused channels (extra channels on a module that aren't currently being used) do not have to be specified in the server's configuration file (although doing so is not harmful). Low-density digital output definition lines have the following syntax:

```
<name> do <modnum> <channum> <pointtype> <logic> <initialstate> <description>
```

- *name* – This is a short, human-readable text name that clients will use to reference this channel. This name should be globally unique across all servers running on the system.
- *do* – This is the string literal "do." This is merely a tag that indicates what type of channel is being described by this line (a low-density digital output). It is mostly a convenience to the parser. The string is case-sensitive and must be all lowercase.
- *modnum* – This field contains the backplane slot number in which the module hosting this channel is installed. This is one of the two fields that implement the module number/channel number addressing scheme described in section 6.2.
- *channum* – This is the number of the channel relative to the module that hosts the channel. For example, a 4-channel low-density digital output module has channels numbered 0 through 3. This is the other of the two fields that implement the module number/channel number addressing scheme described in section 6.2.
- *pointtype* – This is a small (typically two- or three-digit) numerical code defined by Opto-22 that indicates how this channel should be interpreted. Typically all digital output channels on a given module use the same point-type number. Point-type codes can be found in Opto-22 document **1465**, **OptoMMP Protocol Guide**, starting on page 21.

- *logic* – This field contains either a "-" character (indicating this is a negative logic channel) or a "+" character (indicating this is a positive logic channel).
- *initialstate* – This is the state the digital output will be placed into when the server is first started. Valid values for this field are the character "0" (not asserted) or "1" (asserted). The initial state field honors the *logic* field described above.
- *description* – This is a free-form text description of the control connected to this channel. For a digital output, this might be something like "Sample inlet bypass valve" (without the quotes).

6.3.7 High-density Digital Output Definition Lines

Opto-22 makes *low-density* digital output modules (simply, modules with four or fewer channels) and *high-density* digital output modules (modules with more than four channels). The difference between the two has to do with how channels are addressed and whether or not modules have to be explicitly configured with a point-type code (low density modules only). The following describes the format of definition lines for *high-density* digital outputs.

There is one digital output definition line for every digital output channel in the system that is actually wired to hardware. That is to say, unused channels (extra channels on a module that aren't currently being used) do not have to be specified in the server's configuration file (although doing so is not harmful). High-density digital output definition lines have the following syntax.

```
<name> hdo <modnum> <channum> <logic> <initialstate> <description>
```

- *name* – This is a short, human-readable text name that clients will use to reference this channel. This name should be globally unique across all servers running on the system.
- *hdo* – This is the string literal "hdo." This is merely a tag that indicates what type of channel is being described by this line (a high-density digital output). It is mostly a convenience to the parser. The string is case-sensitive and must be all lowercase.
- *modnum* – This field contains the backplane slot number that the module hosting this channel is installed into. This is one of the two fields that implement the module number/channel number addressing scheme described in section 6.2.
- *channum* – This is the number of the channel relative to the module that hosts the channel. For example, a 32-channel high-density digital output module has channels that are numbered 0 through 31. This is the other of the two fields that implement the module number/channel number addressing scheme described in section 6.2.
- *logic* – This field contains either a "-" character (indicating this is a negative logic channel) or a "+" character (indicating this is a positive logic channel).

- *initialstate* – This is the state the digital output will be placed in when the server is first started. Valid values for this field are the character "0" (not asserted) or "1" (asserted). The initial state field honors the *logic* field described above.
- *description* – This is a free-form text description of the sensor. For a digital output, this might be something like "Sample inlet bypass valve" (without the quotes).

6.3.8 Example Configuration File

This section contains a complete example of a typical Opto-22 server configuration file.

```
################################################################################
#
# This is the configuration file for the optoServer (the server that
# controls the Opto-22 SNAP PAC I/O electronics).  See optoServer.cpp
#  for details.
#
################################################################################

@192.168.1.100

# Slot 0
#-------------------------------------------------------------------------
# Analog output module, SNAP-AOV-25 is type 165
# <name> ao <modnum> <channum> <pointtype> <lower> <upper> <gain> <offset> <units> <description>
#-------------------------------------------------------------------------
mfc0  ao  0  0  165  0.0  10.0  100.0  0.0  cc/min  Carrier back pressure controller setpoint
mfc1  ao  0  1  165  0.0  10.0  100.0  0.0  cc/min  Sample back pressure controller setpoint

# Slot 1
#-------------------------------------------------------------------------
# Analog input module, SNAP-AIV-4 is type 12
# <name> ai <modnum> <channum> <pointtype> <lower> <upper> <gain> <offset> <units> <description>
#-------------------------------------------------------------------------
ps101  ai  1  0  12  -10.0  10.0  517.1493   0.0  Torr  Sample bottle manifold pressure
ps102  ai  1  1  12  -10.0  10.0  517.1493   0.0  Torr  Vacuum manifold pressure
bpr    ai  1  2  12  -10.0  10.0  159.7000  -3.0  Torr  System back pressure controller readback

# Slot 2
#-------------------------------------------------------------------------
# Digital output module (low-density) SNAP-ODC5SNK is type 384
#<name> do <modnum> <channum> <pointtype> <logic> <initialstate> <description>
#-------------------------------------------------------------------------
V101  do  2  0  384  +  0  Carrier isolation valve
V102  do  2  1  384  +  0  Sample isolation valve
V103  do  2  2  384  +  0  System vacuum valve
V104  do  2  3  384  +  0  Pressure near DPS sensors

# Slot 3
#-------------------------------------------------------------------------
# Digital output module (high-density)
#<name> hdo <modnum> <channum> <logic> <initialstate> <description>
#-------------------------------------------------------------------------
V201  hdo  3  0  +  0  N2 purge isolation valve
V202  hdo  3  1  +  0  Archive access valve
V203  hdo  3  2  +  0  Turbo pump isolation valve
V204  hdo  3  3  +  0  Sample bypass valve

# Slot 4
#-------------------------------------------------------------------------
# Digital input module (low-density)
```

```
#<name> di <modnum> <channum> <pointtype> <logic> <description>
#-------------------------------------------------------------------------
OT1  di  4  0  256  +  Large heater overtemp trip indicator
OT2  di  4  1  256  +  Small heater overtemp trip indicator

# Slot 5
#-------------------------------------------------------------------------
# Digital input module (high-density)
#<name> hdi <modnum> <channum> <logic> <description>
#-------------------------------------------------------------------------
UPLIMIT   hdi  5  0  -  Slide upper limit indicator
LOWLIMIT  hdi  5  1  -  Slide lower limit indicator

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

7 Server Implementation

In this section we'll briefly look at the implementation of the Opto-22 server code. For a more complete description, the reader is urged to consult the Doxygen-generated documentation for the `OPTOServer` class, its helper classes, and the server's source code in the file *optoServer.cpp*.

7.1 The `OPTOServer` Class

The Opto-22 server is implemented by the `OPTOServer` class and various other support classes that will be described later. Because the server needs to fit into our standard client/server architecture, it is necessarily derived from the `BaseServer` class[9]. This `OPTOServer` class performs many functions.

- It reads and processes the server's configuration file, and it re-writes the configuration file any time a client makes changes to the coefficients used to convert between native units and engineering units (see sections 6.3.2 and 6.3.3 for more details on unit-conversion).
- It establishes communication links (via UDP over Ethernet) to every Opto-22 brainboard defined in the server's configuration file, and configures each analog or digital channel associated with the brainboards.
- It receives and responds to all incoming client request messages. This is done with a series of client message handler methods. There is one client message handler for each message the client API class can generate.

[9]The `BaseServer` class provides methods to handle the low-level details of client message receipt and response generation, implements the server's message queue, provides support for software signals, and implements a number of standard message handlers. For full details, please see design document ***DES-0005, The Client/Server Architecture***.

- It provides custom message handler replacements for the client/server standard SoH message handlers. These handlers know how to properly format and report state-of-health information to any requesting clients.

7.1.1 Configuration File Management

At the heart of configuration file management are five *STL maps* which are members of the `OPTOServer` class – `aoMap`, `aiMap`, `doMap`, `diMap`, and `optoMap`. The first four of these are referred to in this document as *I/O maps*. They maintain all the information specified in the configuration file for analog outputs, analog inputs, digital outputs and digital inputs respectively. These maps are keyed on channel name (the small human-readable text labels assigned to each channel in the configuration file).

The remaining map, `optoMap`, we will refer to as the *device map*. It is keyed on IP address (as a dotted decimal string). Each entry in the map contains a pointer to the communication class that is responsible for communicating with the Opto-22 brainboard that lives at the corresponding IP address.

When the server application is first started, the `OPTOServer` class constructor calls the `ReadConfigFile()` method to process the configuration file[10]. The main purpose of this method is to populate these five maps. As brainboard definition lines are encountered, communication classes to communicate with the corresponding brainboards are immediately instantiated, thereby establishing communications with the brainboards. Pointers to these classes are stored in the device map (`optoMap`). For other line types, the information is simply copied into the appropriate I/O map for later use.

Besides initial server start-up, the only other time the server deals with the configuration file is if a client makes changes to the unit-conversion coefficients associated with an analog input or output channel. Client changes to the coefficients take effect immediately, but they are also immediately written to the server's configuration file so that they will continue to be used the next time the server is restarted.

7.1.2 Hardware Addressing

The following two subsections (7.1.3 and 7.1.4) discuss the class methods that directly communicate with the Opto-22 hardware (using UDP over Ethernet via an Opto-22 provided, PNNL modified communication library). Now would be an appropriate time to discuss Opto-22 channel addressing.

As mentioned in section 6.2, for historical reasons, addressing individual I/O channels on the Opto-22 backplane is somewhat complicated. In short, three different addressing modes are employed. Which of these is used depends on the type of module that hosts a given channel.

[10]The `ReadConfigFile()` method makes extensive use of the `RE_c` regular expression utility class to parse the configuration file. See design document ***DES-0008*, *The RE_c Utility Class*** for details.

In general, these addressing peculiarities are well described by comments in the source code, but they'll be covered here as well for the sake of completeness.

- **All Low-density Digital I/O Channels**
  All low-density digital I/O channels (that is, I/O channels hosted on digital modules with four or fewer I/O channels) are addressed using a single address value, called a *point-number*. Point-numbers are computed from the module number and relative channel number like so:

  pointNumber = moduleNumber * 4 + channelNumber.

  In the calls made through the Opto-22 communication library to read or set low-density digital channels, the module number/channel number addressing scheme used by the configuration file is first converted to a point-number using this equation.

- **All High-density Digital I/O Channels**
  All high-density digital I/O channels (that is, I/O channels hosted on digital modules with more than four I/O channels) actually use the same module number/channel number scheme as used by the configuration file. In the calls made through the Opto-22 communication library to read or set high-density digital channels, the module number and channel number values are used directly.

- **All Analog I/O Channels**
  All analog I/O channels, regardless of whether they are hosted on old two- or four-channel modules or on new high-capacity modules, are addressed using a single address value, also called a *point number*. These point-numbers are computed from the module number and relative channel number like so:

  pointNumber = moduleNumber * 64 + channelNumber.

  (Note that the multiplier is "64" instead of "4" as it is in the low density digital channel case).

  In the calls made through the Opto-22 communication library to read or set analog channels, the module number/channel number addressing scheme used by the configuration file is first converted to a point-number using this equation.

- **Low-density Digital Channel CONFIGURATION**
  Unfortunately, there is one exception to the above addressing scheme for low-density digital channels. It only applies to initial channel *configuration*. During configuration, a point-number address is still computed, but the multiplier used is 64 instead of 4 (same as analog channels). Again, this scheme is only used for initial channel configuration (not to get or set channel values). The reason behind it is described more fully in comments in the server's source code.

7.1.3 Hardware Configuration

Tied closely to parsing the server's configuration file is actually configuring the Opto-22 hardware as appropriate for the specific modules installed on the backplane(s). Like the parsing of the configuration file itself, Opto-22 hardware configuration is a one-time operation that occurs during server initialization only. There is currently no mechanism in place to re-initialize the hardware once it has been configured, and no such mechanism is planned. If changes are made to the configuration file (beyond client requested changes to the unit-conversion coefficients), the server needs to be stopped and restarted in order to re-initialize the hardware.

Digital input and output channels hosted by Opto-22 *high-density* modules do not need to be explicitly configured. They are automatically detected and configured by the Opto-22 brainboard on power-up. All other modules require explicit configuration. This is accomplished in the `OPTOServer` class constructor by calling the `ConfigurePoints()` helper method. This method traverses each of the I/O maps in turn. For each analog channel, and each *low-density* digital channel, the code first looks up the IP address for the brainboard responsible for the channel in the *device map*. The result of this look-up provides the pointer to the communication object that is used to communicate with that specific brainboard. That pointer is then used to write the channel's *point-type* value (specified in the configuration file) to the brainboard.

With that done, one other operation is performed if the channel happens to be a digital output channel (either *high-* or *low-density*), and that is to set the output of the channel to the *initial state* value specified for that channel in the configuration file.

7.1.4 Low-Level Hardware Access

The `OPTOServer` class provides four methods for directly reading or setting analog and digital I/O channels. These are the `_GetDigital()`, `_SetDigital()`, `_GetAnalog()`, and `_SetAnalog()` methods. Other than `ConfigurePoints()`, which is called only once during initial server start-up, these are the only four methods that communicate directly with the Opto-22 hardware. They all work in the same way.

1. The name of the channel to be manipulated is looked up in the appropriate I/O map. One of the bits of information that is returned from this look-up is the IP address of the brainboard responsible for the channel.
2. That IP address is now looked up in the device map. From that look-up, a pointer to the communication object responsible for communicating with the brainboard is obtained.
3. That pointer is used to call the appropriate Opto-22 comm library functions to read or set the channel.
4. Read results (in the case of `_GetAnalog()` and `_GetDigital()`) are returned to the caller.

These four low-level hardware access functions hide the details of the underlying hardware from the rest of the server. It is here that the various addressing modes discussed in section 7.1.2 are implemented. It is also here that native-to-engineering unit-conversion takes place (analog channels), and negative vs. positive logic is dealt with (digital channels).

In addition to these four methods, two others become important when the server is operating in simulator mode (see section 7.2). These are `_SetDigitalInput()` and `_SetDigitalOutput()`. These methods don't actually talk to the Opto-22 hardware (after all, in simulator mode, it is expected that no actual hardware is present); however, they do appear to clients as if they do. The purpose of these two methods is to allow clients to tell the server what values it should report for its various *input* channels. Versions of these methods exist for both the `OPTOServer` class and the `OPTOSimServer` class. When called through the former (that is, when the server is not in simulator mode), the methods simply return an error. When called through the latter, they appropriately update the internal caches the server uses in simulator mode to remember what values to report back to requesting clients.

7.1.5 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***. Those message handlers that need to communicate with the Opto-22 hardware, do so using the low-level hardware interfacing functions defined in section 7.1.4 above.

7.1.6 State-of-Health Reporting

All servers written to our client/server architecture specification have at least the potential to respond to state-of-health (SoH) 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 Opto-22 server has legitimate SoH data to return for each I/O channel 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 digital channel managed by the server, we return a single piece of information: whether the channel is asserted (on) or not asserted (off). For each analog channel we also return a single piece of information: the value (in engineering units) currently on the channel. So reimplementing `GetNumSohParams()` is easy. We simply return the total number of I/O channels being maintained by the server. For the next two functions, we simply traverse the I/O maps, and build appropriate responses based on what values are currently set on the I/O channels. 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.

7.2 Simulator Mode and the `OPTOSimServer` Class

Like all other hardware interfacing servers on the system, the Opto-22 server supports a *simulator* mode. When in simulator mode, the server is able to run without any actual Opto-22 hardware attached. Simulator mode is enormously useful during project development and server testing, because it allows the server software and the clients that use it to be developed and tested long before the actual system hardware is ready.

In the Opto-22 server, simulator mode is implemented as a separate class called `OPTOSimServer`. This class is derived from the normal `OPTOServer` class. Almost all of the functionality of the original class is left alone. That means, when operating in simulator mode, the code to parse and re-write the configuration file, and all of the client/server communication code is exactly the same code that runs normally. This is important, because it means that tests performed on the simulator version of the class are exercising exactly the same software that runs when the server is not simulator mode, and are therefore legitimate tests of the non-simulator version of the server as well.

The main difference between the `OPTOServer` class and the `OPTOSimServer` class is that the latter overrides the six low-level hardware interfacing methods discussed in section 7.1.4. The simulator version of these methods still query the same I/O maps used by the non-simulator version to verify that specified I/O channels actually exist, and they still return the same set of error messages. The difference is, they never pass hardware requests on to actual Opto-22 brainboards. Instead they maintain internal caches in the form of STL maps[11] that remember the values that the server's various D/A I/O channels are supposed to be set at, and return those values to requsting clients. Client changes to output channels are written to the cache maps so the new values will be returned when requested. In addition, while in simulator mode, two new client messages become active (see section 5) that allow clients to tell the server what values to report on its *input* channels as well. This feature is useful for testing the server itself and the various client applications that rely on it.

In the server source code, which version of the server gets built (simulator or non-simulator) depends on the value of a preprocessor macro called `SIMULATOR`. If this macro is set to zero, then the fully functional version of the server is compiled. If set to one, then the simulator version is compiled. How this works is easily seen by examining the server source code itself. In several places you'll find coding constructs similar to the following:

```
#if SIMULATOR == 0
    ...
    non-simulator code here
    ...
```

[11] These are the maps `diSimMap`, `doSimMap`, `aiSimMap`, and `aoSimMap` defined in the `OPTOSimServer` class.

```
#else
    ...
    equivalent simulator code here
    ...
#endif
```

Setting the SIMULATOR macro is done by passing its value to the GNU *make* utility used to build the full set of project software. Specifically, if *make* is executed with no command-line parameters, thusly:

```
make
```

then the non-simulator version of the server (and all other hardware interfacing servers) will be built. To build the simulator version, execute *make* as follows.

```
make SIMULATOR=1
```

7.3 Positive Logic vs. Negative Logic

In many places previously in this document, mention has been made of *positive* and *negative* logic as they pertain to digital channels; but we haven't talked about what that means or why it is important. Now we'll take a closer look.

Electrically, a positive logic digital signal is one in which a high voltage on the digital channel indicates that the channel is *on* or *true*, and a low voltage indicates the channel is *off* or *false*. Negative logic channels are exactly opposite, with a high voltage indicating the channel is *off* (*false*) and a low voltage indicating that the channel is *on* (*true*).

It is quite common for real-world systems to be *mixed logic*, meaning some digital channels use negative logic and others use positive logic. In general, which way a channel operates is a detail that should be hidden from client applications. First, having some signals behave one way, and others behave the other way is confusing. As a client author, do I set a channel true or false in order to turn on a pump? More importantly, during the development of a system, the logic sense for a specific channel may change. If that happens, ALL clients that use the signal would need to be modified so that places where the channel was previously set *true* would now be set *false* and vice versa. That can potentially mean a lot of code changes in a lot of clients, and it opens the door to introducing bugs easily.

As an example of where this might happen, consider an example in which a heater is wired to a mechanical relay such that a low voltage applied to the relay turns the heater on and a high voltage turns the heater off (negative logic). Later, the developers decide to replace the mechanical relay with a more reliable solid-state relay. Unfortunately, the solid-state relay turns the heater on when a high voltage is applied to its control pin, and off when a low voltage is applied (positive logic). If the server didn't make provisions to conveniently

handle this, the only choice would be to modify all clients that manipulate the heater to use the reverse logic sense. But clients don't (and shouldn't) care how the heater is wired electrically. Clients just want to know if the heater is on or off.

The Opto-22 server solves this problem with the *logic* field specified in the server configuration file for all digital I/O channels. For the example above, the relevant channel would initially have been tagged as a negative logic channel. When the relay technology was changed, all that would be needed is to change the field from negative to positive logic. All clients would continue to operate normally with no need to make any code changes. The Opto-22 server handles the change internally. Because of this ability, all clients view the all digital I/O channel logic such that *true* means *on*, *asserted* or *active* and *false* means *off* regardless of the actual situation from an electrical standpoint.

8 The Test Menu Program

For development and testing purposes, a small text-mode menu client application, called *anadigMenu*, (see *anadigMenu.cpp*) has been developed for the Opto-22 server. The program uses the `AnalogClient` and `DigitalClient` classes to provide the two client APIs the server supports. It also maintains a client interface to the system event logger[12] 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

Server Digital Control Items:
 1 - Get num digital input channels
 2 - Get num digital output channels
 3 - Get digital
 4 - Get ALL digital
 5 - Set digital
 6 - Set digital input

Server Analog Control Items:
 7 - Get num analog input channels
 8 - Get num analog output channels
 9 - Get analog
10 - Get ALL analog
11 - Set analog
12 - Set analog input
13 - Set gain and offset
```

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

```
  0 - Exit Program

Enter Selection >
```

The first seven menu items correspond to standard messages that all servers can support[13]. Following this are six items that correspond to the six client messages that are provided by the digital client API, and then the seven client messages provided by the analog client API. Users choose the number of the message they want to send, and they are prompted for additional parameters as needed.

The menu program is a full client, supporting every client request message the Opto-22 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.

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

Pacific Northwest
NATIONAL LABORATORY

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

U.S. DEPARTMENT OF
ENERGY

902 Battelle Boulevard
P.O. Box 999
Richland, WA 99352
1-888-375-PNNL (7665)
www.pnnl.gov