U.S. DEPARTMENT OF ENERGY

PNNL-22599

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

The Fluke 1529 Server

DES-0028

Revision 1

Charlie Hubbard

August 2012

Pacific Northwest
NATIONAL LABORATORY

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

The Fluke 1529 Server

Charlie Hubbard

DES-0028
Revision 1
August 2012

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

Contents

1 Introduction

This document describes the *Fluke 1529 server* application. This server amounts to a user-mode device driver capable of managing one or more Fluke/Hart model 1529 precision temperature measurement instruments. It is implemented by the *fluke1529Server.cpp* source module. The server supports the client interface defined and implemented by the files *fluke1529ClientLib.h* and *fluke1529ClientLib.cpp* respectively. A standard text-mode test menu client for the server is implemented by *fluke1529Menu.cpp*.

The primary documentation for the Fluke 1529 server source code is the Doxygen-generated HTML documentation associated with each of the above source files. That documentation set is automatically built based on the source code itself. It provides the most detailed, most up-to-date descriptions of the code. The document you are reading now is supplemental, and is intended to provide deeper background for the server and its client API. If contradictions between this document and the Doxygen-generated documentation are found, the Doxygen-generated documentation should be considered correct.

Like most of our hardware interfacing servers, the Fluke 1529 server is not necessarily intended to provide an exhaustive interface to all of the hardware's capabilities. Instead, only that subset of functionality that we need for our current projects is supported. For our particular applications, that means the ability to configure the hardware to work with PRT or thermistor type temperature probes, and to read back the temperatures sensed by those probes at any time.

2 Fluke 1529 Hardware

The Fluke 1529 digital thermometer is a four-channel instrument that delivers very high-accuracy, high-precision temperature readings[1] using PRTs, thermistors and/or thermocouples as its input probes (only the first two of these are currently supported by the server). The instrument can be configured directly via its front panel display or configured and read remotely over an RS-232 port.

3 Serial Communications

3.1 IEEE-488.2 Standard

The server communicates with the Fluke 1529 instruments it manages via RS-232 serial links using a text-based messaging protocol based on the IEEE-488.2 specification[2]. A

[1] When using thermistor based temperature probes, temperature accuracy can be as high as 0.002 degrees Celsius with 0.0001 degree precision.

[2] This specification is also called *SCPI 1994* where *SCPI* stands for *Standard Commands for Programmable Instruments*.

detailed description of this protocol is beyond the scope of this document; however, a detailed description of the protocol as it pertains to the Fluke 1529 instrument specifically can be found in section 8 of the *Fluke 1529 Users Guide*. The first few paragraphs of the relevant section appears below. Consult the guide for details on individual commands and command syntax.

> *The 1529 accepts commands that set parameters, execute functions or respond with requested data. These commands are in the form of strings of ASCII-encoded characters. As far as possible, the 1529 conforms to IEEE-488.2, 1992 and SCPI-1994. One notable exception is that compound commands are not allowed as explained below.*
>
> *Commands consist of a command header and, if necessary, parameter data. All commands must be terminated with either a carriage return (ASCII 0D hex or 13 decimal) or new line character (ASCII 0A hex or 10 decimal).*
>
> *Command headers consist of one or more mnemonics separated by colons (:). Mnemonics may use letter characters, the underscore character (_), and possibly numeric digits as well. Commands are not case sensitive. Mnemonics often have alternate forms. Most mnemonics have a long form that is more readable and a short form consisting of three or four characters that is more efficient.*
>
> *A mnemonic may end with a numeric suffix that specifies one of a set of independent function blocks such as input channel data paths. If a numeric suffix is omitted when a particular block must be specified, an error is generated ("Header suffix out of range").*
>
> *Query commands are commands that request data in response. Query commands have a question mark (?) immediately following the command header. Responses to query commands are generated immediately and placed in the output buffer. Responses are then transmitted automatically over the RS-232 port. When using the IEEE-488 interface, responses remain in the output buffer until read or until another command is received or the power is turned off. Responses are lost if not read before the next command is received. Each input buffer holds 128 characters and each output buffer holds 256 characters.*
>
> *Some commands require parameter data to specify values for one or more parameters. The command header is separated from the parameter data by a space (ASCII 20 hex or 32 decimal). Multiple parameters are separated by a comma (,).*
>
> *The 1529 does not allow compound commands (multiple commands per line separated with semicolons). All commands are sequential. The execution of each command is completed before subsequent commands are processed.*

3.2 Shortcomings

Unfortunately, the IEEE-488.2 protocol, at least as implemented on the Fluke 1529, has some significant shortcomings that make it somewhat difficult to work with in the source code. Specifically, it is not strictly a command/response protocol. That is, commands to the unit (as opposed to queries) do not result in any kind of acknowledgment response. This causes two problems. First, there is no way to know for sure if the command was properly received and processed by the instrument. There is also no way to know when the instrument is done processing the command and is now prepared to accept a new command. This causes serious problems during initialization. During the initialization phase, the server needs to send a series of configuration commands to the unit, one after the other. If the instrument sent command acknowledgments, receipt of those acknowledgments could be used as a natural flow-control mechanism. Without that ability, the server can easily send commands much faster than the Fluke's ability to process them. The author has discussed this issue with Fluke engineers. Their best advice is to

> "...wait maybe a quarter of a second between commands, and if that still causes problems, increase the wait time to half a second."

This solution is clunky at best, but apparently it is the only solution available, so that is the strategy we have adopted in the server's initialization phase (although we use full one second delays to ensure this is never an issue).

Fortunately, initialization occurs only once during initial server start up. Once complete, all other commands sent to the hardware are queries. Queries always respond with a result, and we can use the receipt of those responses as a flow-control mechanism.

4 Server Theory of Operation

In this section we'll briefly look at the implementation of the Fluke 1529 server code. For a more complete description, the reader is urged to consult the Doxygen-generated documentation for the `Fluke1529Server` class, its helper classes, and the server's source code in the file *fluke1529Server.cpp*.

4.1 Sensor Naming Convention

The server presents a view to clients based on individual temperature probes, not individual Fluke 1529 units. Individual temperature probes are assigned short, human-readable text names and it is by those names that clients reference the temperature probes. Clients have no way of knowing (and no reason to care) which probes are connected to specific Fluke 1529 units. Probe names must be unique across the project. That is to say, if a probe has been assigned the name "t101," no other project server can have a sensor or control output by that same name. The server uses these text names internally to identify the individual probes and the specific Fluke 1529s that the probes are connected to. Probe names are assigned in the server's configuration file (more fully described in section 6).

4.2 The Need for Data Caching

The process of reading new temperature data for each unit's connected temperature probes takes place over an RS-232 serial interface and is therefore comparatively slow. When many clients are requesting current data from a particular sensor or set of sensors 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 Fluke 1529 unit managed by the server has a data update thread associated with it. At approximately one-second intervals, the thread requests the temperature data for all attached temperature probes and stores them in an internal cache structure. Client requests are then satisfied with data from this cache rather than querying the hardware directly. In this way, clients never have to wait for data to arrive on a slow serial connection, and each unit's serial connection is burdened with a traffic load it can easily handle regardless of the number of requesting clients.

4.3 Threads and Mutexes

The server uses *pthread* library style threads internally. These threads need to access data structures (like the data cache) that the main server application also accesses. Because the threads run concurrently with the main application code, there is the danger that a thread will try to access/modify one of these shared resources at the same time the main code is trying to access/modify the same resource. Allowing that to happen is a recipe for disaster. Instead, pthread library *mutexes*[3] are used to guarantee that these shared resources can only be accessed by one thread at a time. More specific details will be given in section 4.4.2.

4.4 A Layered Implementation

As with most of our servers that control hardware devices via serial links, the Fluke 1529 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 Fluke 1529 units. Next there is the `Fluke1529Unit` class. This class provides all the functionality to configure and operate a single Fluke 1529 unit. Each `Fluke1529Unit` class maintains a dedicated instance of the `SerialProtocol` class, which it uses to communicate

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

with its associated Fluke. Finally there is the `Fluke1529Server` class. This class is responsible for client/server message handling and high level interaction with the units. It maintains one instance of the `Fluke1529Unit` class for each unit the server manages. These are used to query and manipulate the individual Fluke units. In this section, we'll look at each of these three components in detail. The reader is also urged to consult the Doxygen-generated documentation for these three classes.

4.4.1 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 Fluke 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 `SerialProtocol` class is responsible for providing access to the serial port associated with a specific 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 settings. Conversely, when a `SerialProtocol` object is destroyed, the serial port is automatically closed. All communication with the server's Fluke units is done through the `SerialProtocol` class.

4.4.2 The `Fluke1529Unit` Class

The `Fluke1529Unit` class provides the high-level interface to a single Fluke 1529 unit. 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 are needed by the server or the server's clients. This class also provides the thread function (a static class method called `CommThread()`) that implements the thread that keeps the unit's local cache up to date.

Each instance of the `Fluke1529Unit` class...

- has its own internal `SerialProtocol` object which it uses to communicate with its associated Fluke 1529
- 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. At approximately one-second intervals, this thread queries the attached unit for the current temperatures being measured by all of its attached probes and uses the results to update the unit'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 shared structures at the same time

Finally, the Fluke 1529 server's *simulator mode* is also implemented at the `Fluke1529Unit` 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 `Fluke1529Unit` 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[4].

In many places in the server source code that implements the `Fluke1529Unit` 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
```

4.4.3 The `Fluke1529Server` Class

The `Fluke1529Server` class, derived from our standard `BaseServer` class[5], is what makes the Fluke 1529 server application an actual server.

On initial server startup, one instance of the `Fluke1529Server` class is instantiated. Its constructor processes the server's configuration file (see section 6) and instantiates `Fluke1529Unit` objects for each unit defined therein, initializes the units, and then begins normal client/server message processing.

4.4.4 Unit and Probe Maps

In the code, there are two types of *Standard Template Library* (STL) maps that are used to relate a probe name (the short text labels assigned to individual temperature probes in the server's configuration file) to the physical Fluke 1529 unit that is responsible for it and the

[4]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.

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

physical probe channel on that unit the probe is attached to. The first of these is the channel/unit map, called `channelUnitMap` in the code maintained by the `Fluke1529Server` class. This map is keyed on probe name. Each element contains a pointer to the `Fluke1529Unit` object responsible for the unit to which that probe is attached.

The second is called `probeMap`. Each `Fluke1529Unit` objects maintains its own `probeMap` map. It is also keyed on probe name, but here each element is a structure of type `ProbeRecord`. Among other things, the probe record contains the ID of the physical input channel to which the probe is attached, the probe type (PRT or thermistor), and the calibration coefficients that are associated with the probe.

During server operation, the server uses the `channelUnitMap` to determine which of its `Fluke1529Unit` objects is responsible for a specific probe. Then calls made through that object use its `probeMap` to determine what physical probe input the probe is attached to.

5 The Client API

The Fluke 1529 server, like all servers written for our client/server architecture, relies on a client API class for its client interface. Client API classes provide one public method for each message supported by their corresponding server. These methods handle the details of formatting and sending the request message to the server, and receiving, parsing and returning the server's response. This hides all the messy details of client/server message passing from the client.

The client API class for the Fluke 1529 server is called `Fluke1529Client`. It is defined in *fluke1529ClientLib.h* and implemented in *fluke1529ClientLib.cpp*. The most important thing to note about the client interface is that, from the client perspective, individual temperature probes are known by small, human-readable text names (like "inlettemp" or "t203"). These text names are assigned in the server's configuration file, which is described fully in section 6.

5.1 Client-Specific Messages

The Fluke 1529 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[6].

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

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

this section[7].

5.2 FLUKE_GET_TEMP

This message is implemented by the client class' `GetTemp()` method. It returns the current data record for a single temperature probe.

5.3 FLUKE_GET_ALL_TEMPS

This message is implemented by the client class' `GetAllTemps()` method. It returns an STL map of `FlukeChannelRecord` structures, which contain the current temperature values reported by *every* temperature probe managed by the server. The map is keyed on probe name.

5.4 FLUKE_GET_NUM_SENSORS

This message is implemented by the client class' `GetNumSensors()` method. It simply returns the number of temperature probes the server is managing.

6 The Server Configuration File

The Fluke 1529 server uses a configuration file to associate individual Fluke 1529 units with specific serial ports and map human-readable text names to specific temperature probes.

Each temperature probe attached to a Fluke 1529 unit comes with a unique set of calibration coefficients that have been determined at the factory for that specific probe. These calibration coefficients must be sent down to the corresponding Fluke units at initialization time. These coefficients are also stored in the server's configuration file.

6.1 Configuration File Syntax

The server's configuration file is an ASCII text file meant to be hand-edited with a text editor like *gedit*, *emacs* or *vim*. In the text file, blank lines and lines that begin with a "#" character are ignored. Beyond that, there are two separate line types — unit/serial-port definition lines and temperature probe definition lines.

[7] The symbolic constants that comprise the following subsection headers come from the file *fluke1529ClientLib.h*. Please review the Doxygen documentation about those files for more information.

6.1.1 Serial Port Definition Lines

Serial-port definition lines have the following syntax:

```
@<serialPort>
```

where the "@" symbol is the literal character "@," and `<serialPort>` is the name of the serial port that will be used to communicate with the Fluke unit.

The very first (non-comment, non-blank) line in the configuration file *must* be a serial-port definition line. The serial port listed here implicitly defines one Fluke 1529 unit. All subsequent lines (temperature probe definitions) apply to that unit until the next serial port configuration line is encountered.

6.1.2 Temperature Probe Definition Lines

Temperature probe definition lines have the following syntax:

```
name type id serialNumber coef1 coef2 [coef3 coef4]
```

where

- `name` is the text label that is assigned to this temperature probe. Labels are *not* case sensitive, and they cannot contain space characters. These labels must be unique not just across the server, but also across all sensor/control labels used by all servers that make up the project software. Both the server and its clients refer to individual temperature probes by these labels.
- `type` is either the text string "prt" or the string "therm" (without the quotes). Two types of temperature probes are supported by the server – PRT probes and thermistor probes. PRT probes have two associated conversion coefficients (see below). Thermistor probes have four.
- `id` is the hardware input channel on the current Fluke 1529 to which the temperature probe is attached. Fluke 1529s refer to their channels by number. Valid values are 1, 2, 3 or 4.
- `serialNumber` the serial number of the temperature probe attached to that channel. All Fluke temperature probes have a unique serial number, and specific conversion coefficients that apply only to that specific probe. Both the serial number and the conversion coefficients are provided with the probe documentation. The Fluke 1529 can be configured with the serial numbers of the individual probes attached, and that's what this field is for. However, the serial number values are not actually used internally by the units in any way. They are just used as part of the unit's front panel read-out.
- `coef1` is the first calibration coefficient to use when converting the probe's resistance to a temperature (the conversion is handled automatically by the 1529 hardware). This is the "RTPW" coefficient for PRT probes or it's the "A0" coefficient for thermistor probes. See the Fluke 1529 User's Manual for details.

- `coef2` is the second calibration coefficient to use when converting the probe's resistance to a temperature. This is the "A" coefficient for PRT probes or the "A1" coefficient for thermistor probes. See the Fluke 1529 User's Manual for details.
- `coef3` is the third calibration coefficient to use when converting the probe's resistance to a temperature. This is the "A2" coefficient, and it should only be specified for thermistor type probes. See the Fluke 1529 User's Manual for details.
- `coef4` is the forth calibration coefficient to use when converting the probe's resistance to a temperature. This is the "A3" coefficient, and it should only be specified for thermistor type probes. See the Fluke 1529 User's Manual for details.

6.2 Example Configuration File

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

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

# First Fluke 1529 unit
@/dev/ttyS0
temp1   therm  1  A922617  1.0694391e-3   2.5347988e-4   -1.9895463e-6    1.5392257e-7
temp2   therm  2  A922513  1.0694391e-3   2.5347988e-4   -1.9895463e-6    1.5392257e-7
temp3   prt    3  A922502  1.0694391e-3   2.5347988e-4
temp4   therm  4  A922512  1.0694391e-3   2.5347988e-4   -1.9895463e-6    1.5392257e-7

# Second Fluke 1529 unit
@/dev/ttyS1
temp5   prt    1  A922001  1.0694391e-3   2.5347988e-4
temp6   prt    2  A922002  1.0694391e-3   2.5347988e-4

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

7 The Test Menu Program

For development and testing purposes, a small text-mode menu client application, called *fluke1529Menu*, (see *fluke1529Menu.cpp*) has been developed for the Fluke 1529 server. The program uses the `Fluke1529Client` 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:

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

```
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

Fluke 1529 Server Specific Items:
  1 - Get Temperature
  2 - Get All Temperatures
  3 - Get Number of Sensors

  0 - Exit Program

Enter Selection >
```

The first seven menu items correspond to standard messages that all servers can support[9]. Following this are three items that correspond to the three client messages described in section 5. Users choose the number of the message they want to send, and are prompted for additional parameters as needed.

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

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