lhr-host Documentation

Release 1.1.0

Ethan Li

Jun 01, 2021

CONTENTS:

CHAPTER

ONE

LIQUID-HANDLING-ROBOTICS

Controllers and unit tests for liquid-handling robotics with low-cost components. *src/* and *examples/* contain the Arduino library and example sketches, respectively. *lhrhost/* contains a python package. *docs/* contains the source for project documentation.

CHAPTER

TWO

ASSEMBLY INSTRUCTIONS

This is a reference guide to sequential assembly of the liquid-handling robot.

2.1 Basics

Before we build anything, let's set up a few basic things so we don't forget later.

2.1.1 Arduino

Put the Adafruit Motor Shield on the Arduino (insert further description/image).

Put the VIN Jumper onto the shield (insert image).

2.2 Pipettor

The pipettor subsystem is responsible for moving the plunger of the syringe up and down.

2.2.1 Mechanical Assembly

The pipettor subsystem uses a rack-and-pinion mechanism to move the plunger of the syringe. A motor rotates the pinion and slides up and down the rack, pulling and pushing the syringe plunger with it. A plate which travels with the motor moves the slider of a potentiometer, which senses the position of the plate.

Components

The list of acrylic plates will go here. Information about the motor and potentiometer will go here.

Rack-and-Pinion Mechanism

Step-by-step instructions for assembling the rack-and-pinion mechanism will go here.

Motor Housing

Step-by-step instructions for assembling the motor housing will go here.

Potentiometer Mount

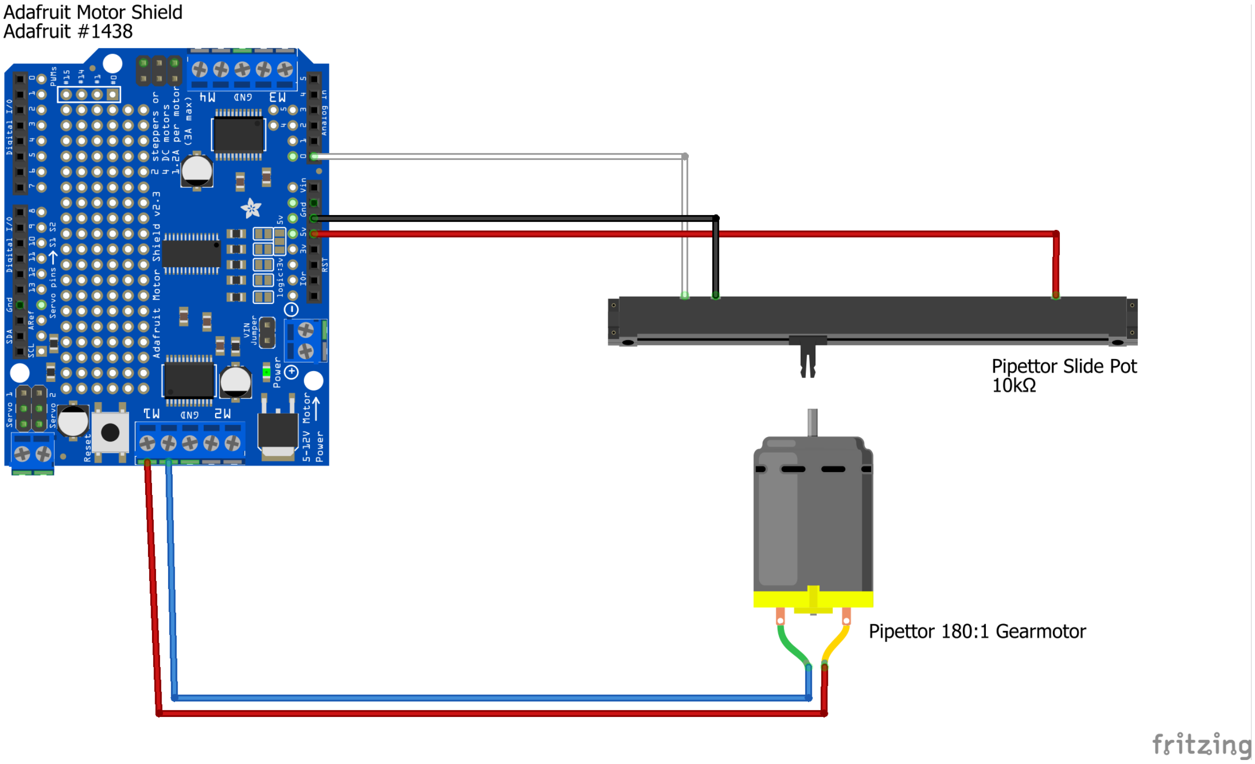
Step-by-step instructions for assembling the potentiometer mount will go here.

Syringe Holder

Step-by-step instructions for assembling the syringe holder will go here.

2.2.2 Electrical Assembly

The pipettor subsystem uses a motor to move the plunger of the syringe, and a slide potentiometer to sense the position of the syringe plunger.



Potentiometer Connections

The potentiometer has two groups of pins. One group has a single pin at one end of the potentiometer, and the other group has a pair of pins at the opposite end of the potentiometer. The pin the single-pin group should be connected to a red wire. For the other group, the pin at the end of the potentiometer should be connected to a white wire, while the remaining pin is between the other two pins and should be connected to a black wire.

(include an image)

The white wire should be connected to Analog In pin A0 on the motor shield. The red wire should be connected to Power pin 5V on the motor shield. The black wire should be connected to Power pin GND on the motor shield.

(include an image)

Motor Connections

When the output shaft of the motor is facing towards you and the pins of the motor are on top, the right wire should be red and the left wire should be blue or gray.

(include an image)

The red wire should be connected to the terminal on M1 which is farther from the GND terminal. The blue wire should be connected to the terminal on M1 which is closer to the GND terminal. Thus, the red wire will be closer than the blue wire to the USB port on the Arduino board.

(include an image)

2.3 Vertical Positioner

The vertical positioner subsystem is responsible for moving the pipettor subsystem up and down. This assembly guide covers assembly of the vertical positioner subsystem around an assembled pipettor subsystem.

2.3.1 Mechanical Assembly

The vertical positioner subsystem uses a rack-and-pinion mechanism to move the pipettor subsystem. A motor rotates the pinion and causes the rack to slide up and down, pushing the pipettor subsystem with it. The housing for the motor is be fixed in place. A plate on the motor housing holds the slider of a potentiometer, which moves up and down with the pipettor subsystem and senses the position of the pipettor subsystem.

Components

The list of acrylic plates will go here. Information about the motor and potentiometer will go here.

Rack-and-Pinion Mechanism

Step-by-step instructions for assembling the rack-and-pinion mechanism will go here.

Motor Housing

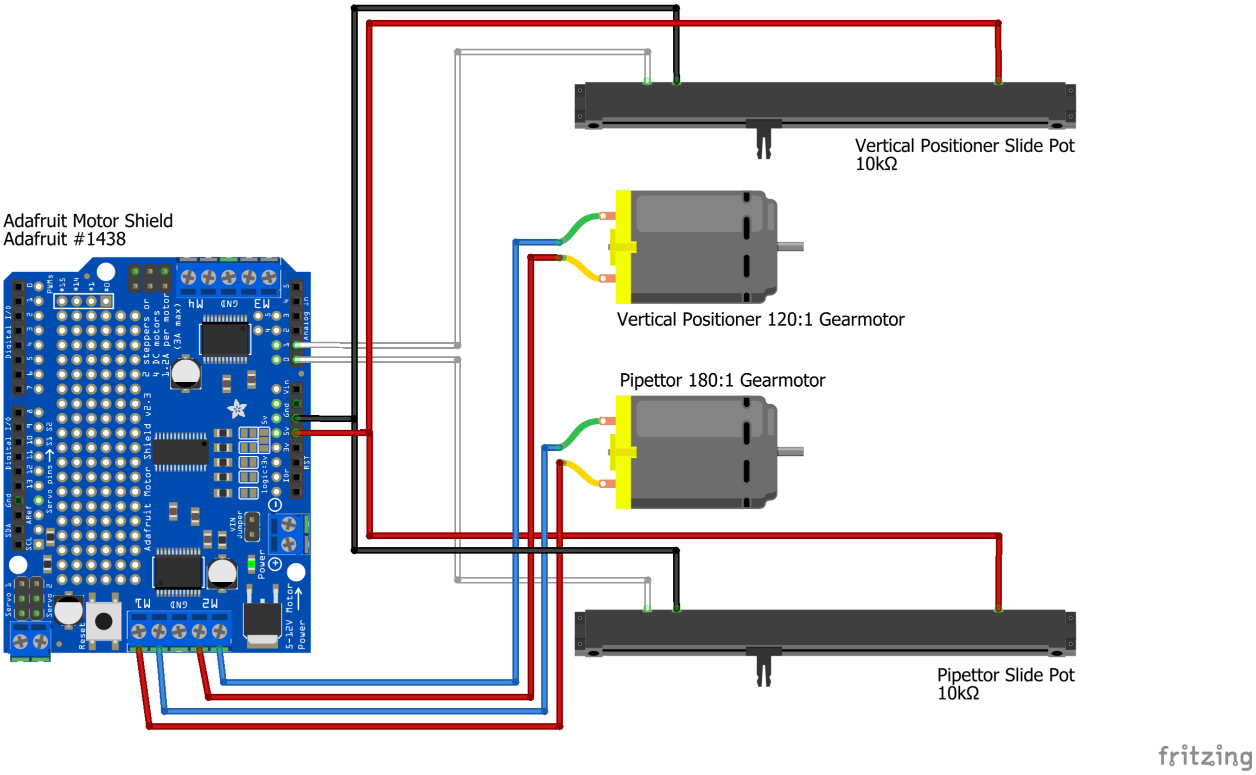
Step-by-step instructions for assembling the motor housing will go here.

Potentiometer Mount

Step-by-step instructions for assembling the potentiometer mount will go here.

2.3.2 Electrical Assembly

The pipettor subsystem uses a motor to move the pipettor subsystem, and a slide potentiometer to sense the position of the pipettor subsystem.



Potentiometer Connections

The potentiometer has two groups of pins. One group has a single pin at one end of the potentiometer, and the other group has a pair of pins at the opposite end of the potentiometer. The pin the single-pin group should be connected to a red wire. For the other group, the pin at the end of the potentiometer should be connected to a white wire, while the remaining pin is between the other two pins and should be connected to a black wire.

(include an image)

The white wire should be connected to Analog In pin A1 on the motor shield. The red wire should be connected to Power pin 5V on the motor shield. The black wire should be connected to Power pin GND on the motor shield.

(include an image)

Motor Connections

When the output shaft of the motor is facing towards you and the pins of the motor are on top, the right wire should be red and the left wire should be blue or gray.

(include an image)

The red wire should be connected to the terminal on M2 which is closer to the GND terminal. The blue wire should be connected to the terminal on M2 which is farther from the GND terminal. Thus, the red wire will be closer than the blue wire to the USB port on the Arduino board.

(include an image)

CHAPTER

THREE

DESIGN REFERENCE

This is a reference for the design of the liquid-handling robot.

3.1 Design Standards

Common standards are used across the liquid-handling robot to improve the clarity of design and assembly.

3.1.1 Mechanical

All acrylic plates are 1/8 inch thick. (describe delrin rods and nylon washers) (describe tab-hole joints)

3.1.2 Electrical

All components on the robot are connected by jumper wires. Components on the robot, such as motors and sensors, should have male connectors on them, either as male header pins or as wires with a male connector.

(insert image here)

Male-female jumper wires are used to connect components on the robot to the motor shield on the Arduino board.

Black jumper wires are used for ground wires; red jumper wires are used for power. White jumper wires are used for sensor signals. Red and blue (or gray, if blue is not available) jumper wires may be used for power, such as for motors.

3.2 Host-Peripheral Communication Protocol

Control of the liquid-handling robot hardware is split into multiple layers of abstraction. In the most simplified view, control between the *user* and the *hardware* is passed through a high-level *host controller* and a low-level *peripheral controller* (Fig. 3.1). The peripheral controller (abbreviated as *peripheral*) runs on an Arduino board and manages aspects of control which require precision timing for acceptable performance. The host controller (abbreviated as *host*) runs on a separate computing device and manages higher-level and/or more computationally demanding aspects of robot planning and control.

Communication between the host and the peripheral is based on passing messages over a communication link. The host sends *command messages* (abbreviated as *commands*) to - and receives *response messages* (abbreviated as *responses*) from - the peripheral, which in turn directly manages the liquid-handling robot hardware. This design can be viewed from the client-server model model, with the host acting as a client, the peripheral acting as a server, commands from the host equivalent to client requests, and responses from the peripheral equivalent to server responses. The protocol described on this page specifies the host and peripheral interfaces for such communication.

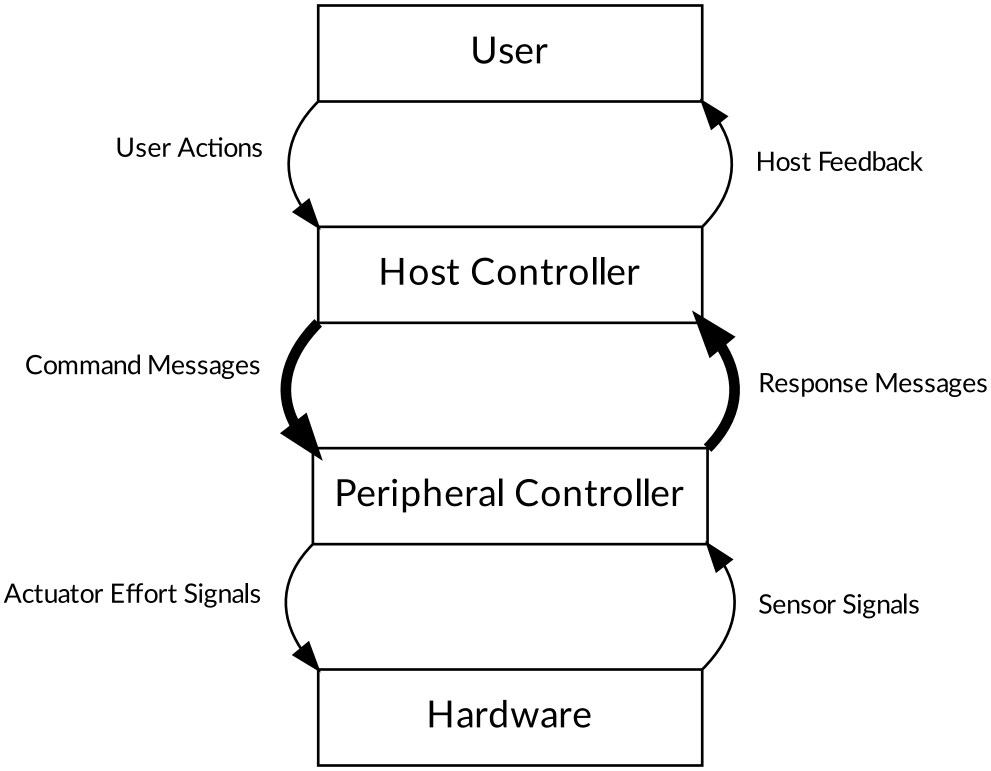


Fig. 3.1: : Simplified functional block diagram showing the interfaces and interactions between the user, the host controller, the peripheral controller, and the hardware. The host-peripheral protocol described on this page specifies the interfaces represented by the diagram's thick edges between the host controller and the peripheral controller.

3.2.1 Overview of Protocol Layers

The messaging system between the host and the peripheral is partitioned into layers of abstraction roughly conforming to the layers of the OSI model (Fig. 3.2):

- The lowest layer is the *physical layer*, which specifies the physical connection between the host and the peripheral. Currently, this is provided by USB.
- The next layer is the *data link layer*, for transport of raw data between the host and the peripheral. Currently, this is provided by UART.
- Above that is the *transport layer*, which serializes messages in discrete packets and which may also allow sending and receiving of other data besides messages. Currently, two transport layers are supported and interchangeable: a Firmata-based transport layer, and an ASCII-based transport layer. The transport layer also specifies the protocol for establishing a connection session for message-passing between the host and the peripheral; this role overlaps with the role of the OSI model's session layer. Once a connection session is established, messages can be passed at the presentation layer.
- Above the transport layer is the *presentation layer*, which specifies the encoding for representing messages. Currently, the protocol for this layer specifies a human-readable representation for messages.
- The final layer for the messaging protocol is the *application layer*, at which messages are sent and received. The protocol for this layer specifies the set of commands and responses which may be passed between the host and the peripheral.

Fig. 3.2: : Examples of data sent at the application, presentation, and transport layers. Messages are sent at the application layer, serialized message strings are sent at the presentation layer, and message packets are sent at the transport layer. The transport layer consists of either an ASCII-based protocol or a Firmata-based protocol.

Any implementation of a host controller or peripheral controller will need to implement the protocols for the transport, presentation, and application layers, which together specify how a message is sent over the data link layer between the host and the peripheral (Fig. 3.3).

3.2.2 Transport Layer

The transport layer provides a mechanism for transmitting and receiving messages (serialized by the presentation layer) as discrete *packets* over the data link layer. Messages can only be passed once a connection session at this layer has been established by a mutual handshake between the host and the peripheral. Currently two interchangeable transport layers are specified, to provide compatibility with different hosts:

- An ASCII-based transport layer is simplest and runs with the least memory overhead on the peripheral. It allows a user (acting in place of the host application and presentation layers) to send commands and receive responses via a host computer's serial console directly, without any additional software.
- A Firmata-based transport layer enables the peripheral to communicate with host controller software which requires using Firmata to connect to the Arduino.

The transport layer specifies a character marking the end of a packet; the transport layer may also optionally specify a character or character sequence marking the start of a packet.

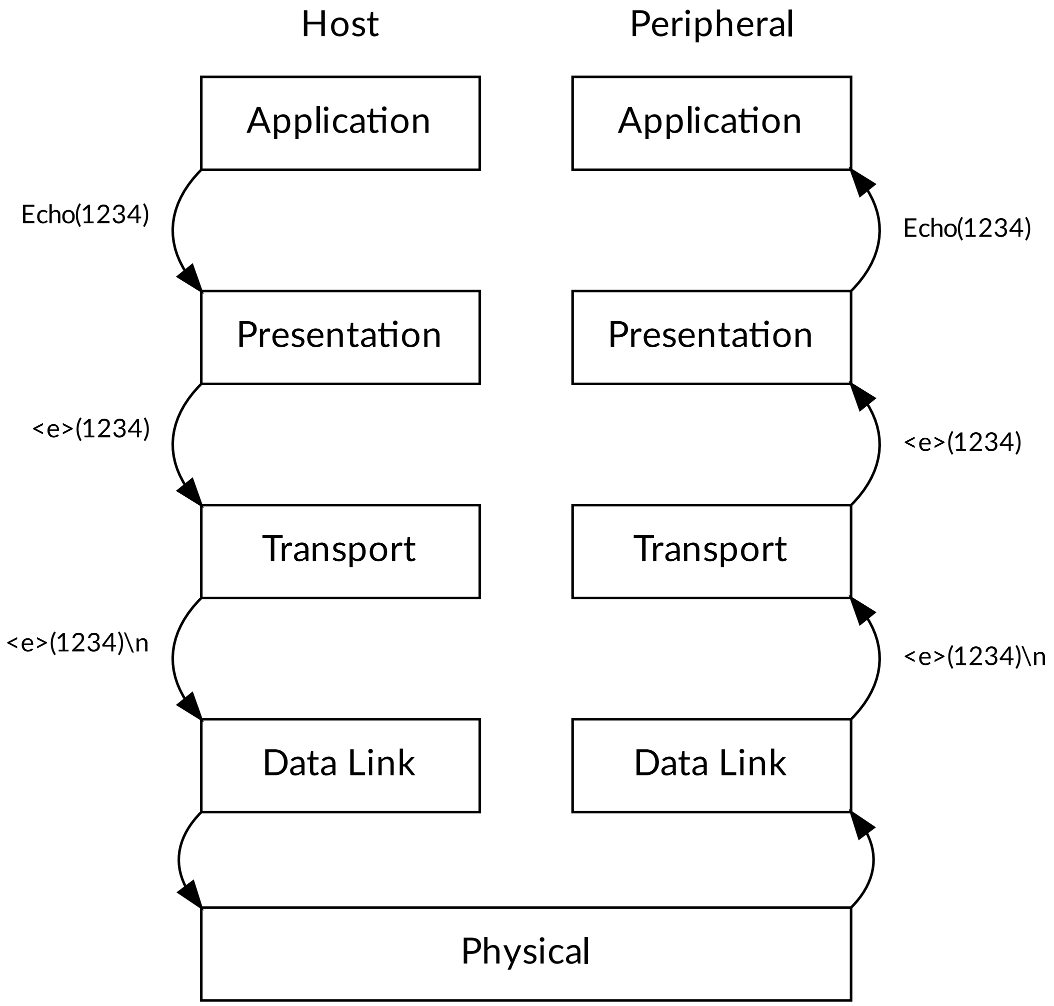


Fig. 3.3: : Overview of protocol layer architecture using the ASCII-based transport layer, with transmission of an echo command as an illustrative example. On the host controller, a message at the application layer is serialized into a string at the presentation layer, which is packaged in a packet at the transport layer before being sent at the data link layer to the peripheral, where it is unpacked from a transport-layer packet into a presentation-layer message string and deserialized into an application-layer message.

ASCII-Based Transport

In the ASCII-based transport layer, all data sent through the layer can be typed using standard ASCII characters on a keyboard and displayed literally. This layer only supports passing of messages conforming to the presentation layer protocol. A packet in this transport layer is constructed from a message (e.g. `<e>(1234)`) by appending a newline `\n` character to it (e.g. `<e>(1234)\n`); then this packet is sent over the data link layer.

A connection session is established by exchange of simple packets between the host and the peripheral. Specifically, once a connection is established at the data link layer, the peripheral will begin sending a tilde `~\n` ping packet to the host at a constant rate (once every 500 ms). When the peripheral receives an empty packet `\n`, it will acknowledge this by sending an empty packet `\n` back to the host. At this point, the handshake is complete, and the transport layer is ready to send messages between the host and the peripheral.

Firmata-Based Transport

In the Firmata-based transport layer, all messages sent through the layer are wrapped in a custom Firmata sysex packet type. The Firmata-based transport layer also supports multiplexing of the Firmata application layer together with the messaging application layer discussed below. Specifically, this transport layer supports a minimal subset of the core Firmata application-layer protocol, namely DigitalInputFirmata, DigitalOutputFirmata, AnalogInputFirmata, and AnalogOutputFirmata. Thus, a peripheral supporting this transport layer can also be controlled by any Firmata host controller with or without support for the full messaging protocol documented on this page.

Messages are sent and received using the reserved Firmata sysex ID `0x0F` byte. Because Firmata sysex packets are also wrapped with a `START_SYSEX` byte `0xF0` and an `END_SYSEX` byte `0xF7`, this means that every packet for a message (e.g. `<e>(1234)`) is constructed by prepending `\xF0\x0F` and appending `0xF7` to it (e.g. `\xF0\x0F<e>(1234)\xF7`); then this message packet is sent over the data link layer.

A connection session is established by the exchange of empty message packets `\xF0\x0F\xF7` between the host and the peripheral. Specifically, once a connection is established at the data link layer, the peripheral will begin sending a tilde `\xF0\x0F~\xF7` ping packet to the host at a constant rate (once every 500 ms). Until the peripheral receives an empty message packet `\xF0\x0F\xF7` from the host, it will only support the core Firmata protocol. Once the peripheral receives such an empty message packet, it will acknowledge this by sending an empty message packet back to the host. At this point, the handshake is complete, and the transport layer is ready to send messages between the host and the peripheral.

3.2.3 Presentation Layer

The presentation layer specifies how messages are serialized into strings for transmission over the transport layer, once a connection session has been established at the transport layer. The data in a message consists of a *channel*, which is used for routing the message to a message handler, and a *payload*, which is used by the message handler as needed.

The presentation layer serializes the *channel name* (a name corresponding to the channel, from a hierarchical namespace) and payload of a message together in a human-readable syntax. The channel name is a string, between 1 and 8 characters long (inclusive), consisting entirely of alphanumeric characters (from *a* to *z*, *A* to *Z*, and *0* to *9*). The payload is a string representing a number. The peripheral currently only accepts payloads representing a 16-bit integer (from -32,768 to 32,767 inclusive); however, in the future the peripheral may send payloads representing floating-point numbers.

The syntax for a message is always `<channel name>(payload)`, with the channel name surrounded by angle brackets and the payload surrounded by parentheses. Note that a message's payload may be empty, in which case it is represented as an empty string (a string of length zero), which has a special semantic meaning described later. Thus, the following strings are examples of syntactically valid messages to send to the peripheral:

- `<pkl>(1234)`
- `<pt123456>(4321)`

- `<v0>()`
- `<zt>(-5)`: Numeric digits, such as `5`, and a hyphen, namely `-`, at the start of the payload, are considered valid characters for the payload.

And the following strings are examples of malformed (syntatically invalid) strings for command messages sent to the peripheral:

- **`<e>(123456)`: The payload in this message will silently overflow, potentially producing unexpected behavior.**
  - Because the peripheral parses the payload as a 16-bit integer, the number 123,456 (which cannot be represented in only 16 bits) will silently overflow, so the peripheral will parse the payload as -7,616.
  - If you are working with a potentially large number in your payload, you can pass it to the echo command (discussed below), which will trigger a response whose payload is the value parsed by the peripheral.
  - Commands are designed to be robust to integer overflows by sanitizing their inputs, so that they will always constrain the parsed value to be within valid ranges, or so that they ignore invalid values. However, you may still get undesirable behaviors with the sanitized values.
- **`<>(2)`: Channel name cannot be an empty string. This command will be ignored by the peripheral.**
  - This will always be part of the parser's behavior.
- **`<v 0>()`: Channel name can only consist of alphanumeric characters, and the peripheral's response is undefined behavio**
  - All other characters are ignored on parsing, and each one triggers a warning sent over serial (not serialized as a message).
  - The command is handled as if those characters were not part of the channel name.
  - For example, `<v 0>()` is handled as `<v0>()`. Additionally, if error logging is enabled on the peripheral, the peripheral will send a warning report over serial: `W: Channel name starting with 'v' has unknown character '32'. Ignoring it!`.
- **`<pt1234567>(4321)`: Channel name is too long, and the peripheral's response is undefined behavior. In the current impl**
  - Any extra characters beyond the 8-character limit are discarded, and each one triggers an error report sent over serial (not serialized as a message).
  - The command is handled as if those characters were not part of the channel name.
  - For example, for the message `<pt1234567>(4321)`, the character `7` in the channel name will be discarded, and the command is handled as `<pt123456>(4321)`. Additionally, if error logging is enabled on the peripheral, the peripheral will send an error line over serial: `E: Channel name starting with 'pt123456' is too long. Ignoring extra character '55'!`.
- **`<zt>(5.0)`: Payload cannot be a floating-point number. If the channel name is handled by the peripheral, this could caus**
  - Invalid characters such as decimal points are ignored, so the command would be interpreted as if the message had been `<zt>(50)`.
  - Additionally, if error logging is enabled on the peripheral, the peripheral will send a warning report over serial: `W: Payload on channel 'zt' has unknown character '46'. Ignoring it!`, which is not serialized as a message.
- **`<zt>(1ab2 3)`: Payload can only be an integer. If it contains other characters, this could cause undefined behavior. In the**

  - Invalid characters are ignored, so the command would be interpreted as if the message had been `<zt>(123)`.
  - Additionally, if error logging is enabled on the peripheral, the peripheral will send a series of warnings over serial:

```
W: Payload on channel 'zt' has unknown character '97'. Ignoring it!
W: Payload on channel 'zt' has unknown character '98'. Ignoring it!
W: Payload on channel 'zt' has unknown character '99'. Ignoring it!
```

3.2.4 Application Layer

At the application layer, the peripheral receives and handles all commands which it recognizes, takes actions based on those commands, and sends one or more responses for each command. As mentioned in the description of the presentation layer, a message consists of a channel and a payload; the peripheral will not handle or send a response to any command with a channel which it does not recognize. The peripheral handles received commands in first-come-first-served order, and it will handle every received command (as long as the peripheral recognizes the command's channel).

After a transport-level connection is established, the peripheral runs an event loop in which it:

- Updates its internal state based on sensor signals received from hardware and commands received from the host.
- Sends responses to the host based on received commands and internal state.
- Emits actuator effort signals to the hardware.

At most one command is received and handled per iteration of the peripheral's event loop. The peripheral may also send one or more response messages without having received the usual corresponding command, namely upon completion of the transport-level connection handshake and as a result of certain modes set by commands. The peripheral will not send more than one response of the same channel per iteration of the event loop, but it may send multiple responses of different channels in the same iteration of the event loop.

Design Principles

The peripheral is a simple high-level interpreter which performs actions specified by instructions, following the discussion of interpreters in [SaltzerKaashoek2009]:

- Commands are the instructions specifying actions or sequences of actions for the peripheral to execute.
- The actions which the peripheral takes based on received commands consist of updating internal state, emitting actuator effort signals to hardware, and sending response messages to host.
- Different sequences of actions triggered by commands on different channels may execute concurrently, in the sense that the actions from these different sequences will be interleaved through the peripheral's event loop. For example, the peripheral may send a sequence of response messages to the host while it is also moving an actuator to a position specified by a previous command.
- If the peripheral is executing a sequence of actions due to a command from some channel but receives a subsequent command on that same channel, it will interrupt (i.e. stop executing) the previous sequence of actions from that previous command and start executing the new sequence of actions from the new command. For example, if the peripheral is in the middle of moving an actuator to some position but receives a new command to move that actuator to a new position, the peripheral will only take actions to move the actuator to the new position. Exceptions to this occur when multiple channels interact with a shared resource (represented by a separate channel), in which case they will interrupt each other; such exceptions are explicitly documented.

- Commands specifying sequences of actions will complete asynchronously, in the sense that the peripheral will send an initial response indicating that it has begun executing the sequence of actions, and - at a later time - zero or more subsequent responses on other channels as the peripheral completes subsequent actions in that sequence. For example, if the peripheral receives a command instructing it to start sending responses to the host reporting the position of an actuator at regular time intervals, the peripheral will first send a response on the same channel as the command, and then it will send responses on a different channel at the appropriate times.

Commands and responses are designed to correspond roughly to an associative memory abstraction with READ/WRITE semantics, in the spirit of the discussion of memory abstactions in [SaltzerKaashoek2009]. Some commands trigger additional actions beyond the READ/WRITE operations specified by this memory abstraction, but all commands provide READ/WRITE semantics:

- States, parameters, and operation modes in the peripheral are internal variables uniquely named by a command/response channel specified by the protocol documentation; names for variables do not change except with protocol design changes.
- A command with an empty payload corresponds to a READ operation of the value of the peripheral's variable named by the channel. The peripheral will handle such a command by sending a response on the same channel with the value of the variable named by that channel.
- A command with a non-empty payload corresponds to a WRITE operation of the command's payload value to the variable named by the command's channel. Every command corresponding to a WRITE operation will also trigger a response from the peripheral with the value of the variable immediately after completion of the attempted WRITE operation. Thus, a command with a non-empty payload is actually a WRITE + READ operation sequence. The host can use the payload of the response to check for faults in the command specifying the WRITE operation - for example, if the payload contained an invalid value which was changed to become a valid value written to the variable.
- Some variables are read-only; for these variables, an attempted WRITE operation does not change the value of that variable.
- All variables are initialized to operational default values when the peripheral's event loop begins.

All responses from the peripheral correspond to the result of a READ operation of an internal variable, whether or not the host had previousy issued a command with READ/WRITE semantics to the peripheral on that response's channel. Commands which do not cause the peripheral to take actions beyond READ/WRITE operations are idempotent. However, as mentioned above, some commands trigger additional actions beyond READ/WRITE operations and thus have further effects on the peripheral:

- Some commands correspond to execution of a WRITE operation of the variable specified by the command channel, followed by a READ operation of that variable, followed by additional actions. For example, a command instructing the peripheral to perform a hard reset will cause the peripheral to send a response on the same channel indicating that the peripheral is about to perform a hard reset, after which the peripheral will perform a hard reset.
- The additional actions specified by such commands may also include additional WRITE and/or READ operations. For example, a command instructing the peripheral to move an actuator to a specified position will set the target position of that actuator (WRITE), cause the peripheral to respond with the updated value of that target position (READ), set the actuator positioning mode (additional action for an implicit WRITE), cause the peripheral to respond with the updated value of that mode (additional action for an implicit READ), and cause the peripheral to adjust actuator effort signals to move the actuator towards that target position (additional actions).
- Changes to internal variables resulting from those additional actions will be visible through other commands corresponding to READ operations. For example, a command instructing the peripheral to move an actuator to a specified position will also cause a sequence of updates to an internal variable whose value is the current position of the actuator, which the host can query through a READ command.
- Such changes to internal variables may also be reported in responses from the peripheral which are not directly prompted by host commands, when such behavior is enabled by a mode in the peripheral. Such modes can be written to and read from by commands with READ/WRITE semantics. For example, if the appropriate mode is

enabled (either by default or by a WRITE command for that mode), the peripheral will send a response to the host with the final position of the actuator when the actuator stops moving; and that response will be on the same channel as a READ response for the internal variable whose value is current position of the actuator.

These design principles were chosen towards the principle of least astonishment, and informed by experiences with an earlier version of the protocol in which inconsistent command semantics produced an unintuitive interface.

Messaging Protocol

At the application layer, all peripherals must support the *Core Protocol Subset*, which provides basic communication functionality. Additionally, all peripherals must support either the minimal subset of the Firmata application-layer protocol provided by the Firmata-based transport layer or the *Board Protocol Subset*; either application-layer protocol subset allows control of the built-in LED on the peripheral, along with reading of hardware pin values. Note that the minimal subset of the Firmata application-layer protocol also enables writing of values to hardware pins, while the Board Protocol Subset currently does not.

3.2.5 Overview of Peripheral States

At a simplified level, the peripheral's externally-visible behavior can be described as a hierarchical state machine (Fig. 3.4). After it is first turned on, the peripheral runs a setup routine and enters the **ConnectionHandshake** state. It remains in this state, looping to establish a transport-layer connection session with the host. When a connection session is established, the peripheral will proceed to the **Operational** state.

In the **Operational** state, the peripheral loops and:

- Updates its substates and internal state variables based on sensor signals received from hardware and commands received from the host.
- Emits actuator effort signals to the hardware.

An **Operational** peripheral can also transition back into the **ConnectionHandshake** state by restarting when it receives a `<r>(...)` reset command, as if the hardware reset button of the Arduino running the peripheral controller has been pressed. Command syntax, and the semantics of this reset command, are described later.

Fig. 3.4: : Overview of peripheral controller's behavior. The peripheral starts by entering the **ConnectionHandshake** state and transitions between the **ConnectionHandshake** and **Operational** states based on commands received from the host.

The **Operational** state is actually a superstate with several orthogonal regions, each of which controls a single axis of the robot hardware and can be thought of as its own independent state machine. Thus, we will refer to each orthogonal region of the **Operational** superstate as an *axis controller*. Each axis controller receives all command messages received by the peripheral and is able to send response messages to the host. Each axis controller also generally behaves the same way (exceptions are documented later), follows the standard command/response message syntax, and has the same command/response protocol, so we will next discuss command/response syntax in the **Operational** state.

3.2.6 References

CHAPTER

FOUR

MESSAGING REFERENCE

This is a reference for the messaging protocol (at the application layer of the *Host-Peripheral Communication Protocol*) of the liquid-handling robot.

4.1 Core Protocol Subset

All peripherals must support the Core protocol subset, which provides basic protocol functionality. Channels are organized hierarchically, and the names of child channels contain the names of their parent channels as prefixes. Each level in the hierarchy corresponds to one character of the channel name.

All channels support a READ command which simpy causes the peripheral to send a READ response on that same channel, so READ commands are only explicitly documented for channels which are READ-only.

Here are the channels specified by the Core protocol subset:

4.1.1 Echo

- **Channel name**: *e*
- **Description**: This command allows the host to test its connection to the peripheral by sending an Echo command with a payload to the peripheral; the peripheral will send a response with the same peripheral it just received. This command also allows the host to test how its payload will be parsed by the peripheral.
- **Semantics**: READ/WRITE
  - **WRITE+READ command**: If payload is provided in the command, the peripheral uses it to set the internal variable named by the Echo channel to the payload's value, and then the peripheral sends a READ response.
  - **READ response**: The peripheral sends a response on the Echo channel with the latest value of the internal variable named by the Echo channel.

Fig. 4.1: : Examples of commands and responses associated with the Echo channel.

4.1.2 Reset

- **Channel name**: *r*
- **Description**: This command instructs the peripheral to perform a hard reset.
- **Semantics**: READ/WRITE + actions
  - **WRITE+READ+Actions command**: If the payload is *1*, the peripheral will send a READ response and then perform a hard reset by executing the actions described below. Otherwise, the peripheral will send a READ response and do nothing.
  - **READ response**: The peripheral sends a response on the Reset channel with payload *1*, if the peripheral is about to perform a hard reset, or else with payload *0*.
  - **Actions**: The peripheral will enter a brief infinite loop which blocks the peripheral's event loop and causes an internal watchdog timer in the peripheral to hard reset the entire peripheral, including resetting the host-peripheral connection.

Fig. 4.2: : Examples of commands, responses, and actions associated with the Reset channel.

4.1.3 Version

- **Child channels**:
  - Version/Major
  - Version/Minor
  - Version/Patch
- **Channel names**:
  - Version: *v*
    - Version/Major: *v0*
    - Version/Minor: *v1*
    - Version/Patch: *v2*
- **Description**: These commands allow the host to query the peripheral for its protocol version. The full version has three components (major, minor, patch), and so the Version/Major, Version/Minor, and Version/Patch child channels (each with their own channel name) enable querying the three respective components individually. The Version channel can also be used to query the three components together.
- **Semantics** for child channels: READ
  - **READ command**: The peripheral simply sends a READ response to any command on the Version/Major, Version/Minor, and Version/Patch channels.
  - **READ response**: The peripheral sends a response with the version component's value as the payload for the version component named by the (child) channel.
- **Semantics** for Version channel: READ
  - **READ command**: The peripheral sequentially sends a READ response to any command on Version channel.
  - **READ response**: The peripheral sequentially sends the READ responses for Version/Major, Version/Minor, and Version/Patch.

Fig. 4.3: : Examples of commands and responses associated with the Version channel and its child channels.

4.2 Board Protocol Subset

Peripherals using the ASCII-based transport layer are required to support the Board protocol subset, which provides basic hardware pin functionality for Arduino boards. The Board protocol subset should only be excluded if the peripheral does not have enough memory to support it or if the peripheral is running a Firmata-based transort layer, which also provides application-layer functionality redundant to (and more complete than) the Board protocol subset. Channels are organized hierarchically, and the names of child channels contain the names of their parent channels as prefixes. Each level in the hierarchy corresponds to either one character of the channel name or a multi-digit number.

All channels support a READ command which simply causes the peripheral to send a READ response on that same channel, so READ commands are only explicitly documented for channels which are READ-only.

Here are the channels specified by the Board protocol subset:

4.2.1 BuiltinLED

- **Child channels**:
  - BuiltinLED/Blink (child channels of BuiltinLED/Blink documented below)
- **Channel names**:
  - BuiltinLED: *l* (the lower-case letter "L")
    - BuiltinLED/Blink: *lb* (names of child channels of BuiltinLED/Blink documented below)
- **Description**: These commands allow the host to control the built-in LED (on pin 13) of the Arduino board.
- **Semantics** for child channels documented below.
- **Semantics** for BuiltinLED channel: READ/WRITE + Actions
  - **WRITE+READ+Actions command**: If the payload is *1*, the peripheral will set the built-in LED to HIGH and send a READ response. If the payload is *0*, the peripheral will set the built-in LED to LOW and send a READ response. If the payload is either *0* or *1* and the built-in LED was blinking, this command will interrupt the blinking first, so that a READ command on the BuiltinLED/Blink channel will return *0*. Otherwise, the peripheral will send a READ response and do nothing.
  - **READ response**: The peripheral sends a response on the BuiltinLED channel with payload *1* if the built-in LED is HIGH and *0* if the built-in LED is LOW.
  - **Actions**: If the WRITE+READ+ACTIONS command has a valid payload, the peripheral will set the state of the built-in LED accordingly.

Fig. 4.4: : Examples of commands and responses associated with the BuiltinLED channel.

BuiltinLED/Blink

- **Child channels**:
  - BuiltinLED/Blink/HighInterval
  - BuiltinLED/Blink/LowInterval
  - BuiltinLED/Blink/Periods
  - BuiltinLED/Blink/Notify
- **Channel names**:
  - BuiltinLED/Blink: *lb*
    - BuiltinLED/Blink/HighInterval: *lbh*
    - BuiltinLED/Blink/LowInterval: *lbl*
    - BuiltinLED/Blink/Periods: *lbp*
    - BuiltinLED/Blink/Notify: *lbn*
- **Description**: These commands allow the host to blink the built-in LED:
  - BuiltinLED/Blink instructs the peripheral to start or stop blinking the built-in LED.
  - BuiltinLED/Blink/HighInterval and BuiltinLED/Blink/LowInterval together specify the amounts of time (in milliseconds) the LED will be on and off, respectively, while the LED blinks.
  - BuiltinLED/Blink/Periods specifies the number of periods the LED will blink for, before turning off.
  - BuiltinLED/Blink/Notify sets the blinking notification mode - when it is enabled and the built-in LED is blinking, the peripheral will send a BuiltinLED READ response every time the built-in LED goes to HIGH or LOW.
- **Semantics** for child channels: READ/WRITE
  - **WRITE+READ command**: The peripheral updates the variable named by the child channel with the value specified in the payload and sends a READ response with the new value of the variable:
    - BuiltinLED/Blink/HighInterval and BuiltinLED/Blink/LowInterval: the payload's value must be a positive number (in milliseconds) for the corresponding variable to be updated with it; otherwise, the corresponding variable will remain at its previous value.
    - BuiltinLED/Blink/Periods: if the payload's value is negative, then the LED will blink forever (at least until a BuiltinLED/Blink command is set to stop blinking). If the payload's value is nonnegative, then the LED will blink for that number of HIGH/LOW cycles before the LED stops blinking and stays at LOW.
    - BuiltinLED/Blink/Notify: if the payload is *1*, the peripheral will enable the blinking notification mode. If the payoad is *0*, the peripheral will disable the blinking notification mode. Otherwise, the mode will not change from its previous value.
  - **READ response**: The peripheral sends a response whose payload is the value for the blinking parameter/mode named by the (child) channel.
- **Semantics** for BuiltinLED/Blink channel: READ/WRITE + Actions
  - **WRITE+READ+Actions command**: If the payload is *1*, the peripheral will start blinking the built-in LED and send a READ response; this blinking will interrupt the previous state on the BuiltinLED channel. If the payload is *0*, the peripheral will stop blinking the built-in LED and send a READ response. Otherwise, the peripheral will send a READ response and do nothing. After completion of every HIGH/LOW blink cycle, if the variable associated with BuiltinLED/Blink/Periods is nonnegative, that variable is decremented by

1. If that variable becomes *0*, then its value is reset to *-1*, blinking stops, and the peripheral sends a READ response for BuiltinLED/Blink and a READ response for BuiltinLED/Blink/Periods.

  - **READ response**: The peripheral sends a response on the BuiltinLED/Blink channel with payload *1* if the built-in LED is blinking and *0* if the built-in LED is not blinking.

Fig. 4.5: : Examples of commands and responses associated with the BuiltinLED/Blink channel.

4.2.2 IOPins

- **Child channels**:
  - IOPins/Analog
    - IOPins/Analog/0
    - IOPins/Analog/1
    - IOPins/Analog/2
    - IOPins/Analog/3
  - IOPins/Digital
    - IOPins/Digital/2
    - IOPins/Digital/3
    - IOPins/Digital/4
    - IOPins/Digital/5
    - IOPins/Digital/6
    - IOPins/Digital/7
    - IOPins/Digital/8
    - IOPins/Digital/9
    - IOPins/Digital/10
    - IOPins/Digital/11
    - IOPins/Digital/12
    - IOPins/Digital/13
- **Channel names**:
  - IOPins: *i*
    - IOPins/Analog: *ia*
      - IOPins/Analog/0: *ia0*
      - IOPins/Analog/1: *ia1*
      - IOPins/Analog/2: *ia2*
      - IOPins/Analog/3: *ia3*
    - IOPins/Digital: *id*
      - IOPins/Digital/2: *id2*
      - IOPins/Digital/3: *id3*

    - IOPins/Digital/4: *id4*
    - IOPins/Digital/5: *id5*
    - IOPins/Digital/6: *id6*
    - IOPins/Digital/7: *id7*
    - IOPins/Digital/8: *id8*
    - IOPins/Digital/9: *id9*
    - IOPins/Digital/10: *id10*
    - IOPins/Digital/11: *id11*
    - IOPins/Digital/12: *id12*
    - IOPins/Digital/13: *id13*

- **Description**: These commands allow the host to read the value of a specified pin of the peripheral. Note that IOPins, IOPins/Analog, and IOPins/Digital are not currently used for messaging - only the child channels of IOPins/Analog and IOPins/Digital are used. For more complete pin I/O functionality, it is recommended to use the Firmata-based transport layer, which enables support for more advanced pin modes, notification of pin value changes, and writing to pins.
- **Semantics** for child channels: READ-only
  - **READ command**: The peripheral simply sends a READ response to any command which is a child of IOPins/Analog or IOPins/Digital.
  - **READ response**: The peripheral performs an analog read of an analog pin (for a message on a child channel of IOPins/Analog) or a digital read of a digital pin (for a message on a child channel of IOPins/Digital) sends a response with that reading as the payload for the pin named by the (child) channel. Analog reads produce values in range 0-1023, inclusive; digital reads produce values in the set {0, 1}.

Fig. 4.6: : Examples of commands and responses associated with the child chnnels of the IOPins/Analog and IOPins/Digital channels.

4.3 LinearActuator Protocol Subset

Peripherals implement at least one instance of the LinearActuator protocol subset to provide linear actuator control functionality for control of liquid-handling robotics hardware. Channels are organized hierarchically, and the names of child channels contain the names of their parent channels as prefixes. Each level in the hierarchy corresponds to one character of the channel name. Peripherals may implement multiple instances of the LinearActuatorProtocol subset on multiple distinct top-level parent channels, enabling concurrent independent control of different linear actuators in the robotics hardware.

All channels support a READ command which simply causes the peripheral to send a READ response on that same channel, so READ commands are only explicitly documented for channels which are READ-only.

Here are the channels specified by the LinearActuator protocol subset:

4.3.1 LinearActuator

- **Child channels**:
  - LinearActuator/Position (its child channels are documented below)
  - LinearActuator/SmoothedPosition (its child channels are documented below)
  - LinearActuator/Motor (its child channels are documented below)
  - LinearActuator/FeedbackController (its child channels are documented below)
- **Channel names**:
  - LinearActuator: a unique character assigned per linear actuator. Pipettor Axis: *p*, Z-Axis: *z*, Y-Axis: *y*, X-Axis: *x*. In documentation below, a _ underscore character is used as a placeholder for this character.
    - LinearActuator/Position: *_p* (names of its child channels are documented below)
    - LinearActuator/SmoothedPosition: *_s* (names of its child channels are documented below)
    - LinearActuator/Motor: *_m* (names of its child channels are documented below)
    - LinearActuator/FeedbackController: *_f* (names of its child channels are documented below)
- **Description**: These commands allow the host to control a linear actuator of the liquid-handling robot.
- **Semantics** for child channels documented below.
- **Semantics** for LinearActuator channel: READ-only
  - **READ command**: The peripheral simply sends a READ response.
  - **READ response**: The peripheral sends a response on the LinearActuator channel with payload *1* if the actuator is operating in direct motor duty control mode with a nonzero duty, *2* if the actuator is operating in position feedback control mode, *0* if the actuator is operating in direct motor duty control mode with a null (zero) duty, *-1* if the actuator's controller has stopped from a motor stall, *-2* if the actuator's controller has stopped from convergence to a feedback control target, and *-3* if the actuator's controller has stopped from a timer timeout. Negative payloads correspond to stopped actuator states where the controller has stopped running, while positive payloads correspond to moving actuator states where the controller is still running, and null (zero) payload corresponds to the stopped actuator state where the controller was initially instructed to stop the actuator.

4.3.2 LinearActuator/Position

- **Child channels**:
  - LinearActuator/Position/Notify (its child channels are documented below)
- **Channel names**:
  - LinearActuator/Position: *_p*
    - LinearActuator/Position/Notify: *_pn* (names of its child channels are documented below)
- **Description**: These commands relate to tracking of the (raw values from the) position sensor in the linear actuator.
  - LinearActuator/Position represents the raw values of the position sensor.
  - LinearActuator/Position/Notify provides functionality for the peripheral to send raw values of the position sensor to the host.
- **Semantics** for child channels documented below.

- **Semantics** for LinearActuator/Position channel: READ-only
    - **READ command**: The peripheral simply sends a READ response.
    - **READ response**: The peripheral sends a response on the LinearActuator/Position channel with a payload whose value is the current raw position reading from the linear actuator's position sensor.

LinearActuator/Position/Notify

- **Child channels**:
    - LinearActuator/Position/Notify/Interval
    - LinearActuator/Position/Notify/ChangeOnly
    - LinearActuator/Position/Notify/Number
- **Channel names**:
    - LinearActuator/Position/Notify: *_pn*
        - LinearActuator/Position/Notify/Interval: *_pni*
        - LinearActuator/Position/Notify/ChangeOnly: *_pnc*
        - LinearActuator/Position/Notify/Number: *_pnn*
- **Description**: These commands relate to notification of the host of updates to the (raw values from the) position sensor in the linear actuator.
    - LinearActuator/Position/Notify sets the position notification mode - when the payload is *0*, the peripheral will not notify the host of updated positions; when the payload is *1*, the peripheral will notify the host of updated positions no more than once every *n* iterations of the peripheral's event loop, where *n* is specified by LinearActuator/Position/Notify/Interval; when the payload is *2*, the peripheral will notify the host of updated positions no more than once every approximately *n* milliseconds, where *n* is specified by LinearActuator/Position/Notify/Interval.
    - LinearActuator/Position/Notify/Interval specifies the interval between position update notifications.
        - When LinearActuator/Position/Notify is in the mode to notify the host no more than once every *n* iterations of the peripheral's event loop, the variable corresponding to this channel specifies *n*. The timing of this notification interval is approximate in that the timing between different iterations of the peripheral's event loop may vary slightly depending on amounts of data transmitted/received over the transport layer of communications between the host and the peripheral.
        - When LinearActuator/Position/Notify is in the mode to notify the host no more than once every *n* milliseconds, the variable corresponding to this channel specifies *n*. The timing of this notification interval is approximate in that there is a timer which is checked against *n* once per iteration of the peripheral's event loop, and a position update notification will only be sent once the timer exceeds *n*.
        - In either case, if LinearActuator/Position/ChangeOnly is enabled, the peripheral will not send position update notifications when the position has not changed, which will increase the notification interval above *n*.
    - LinearActuator/Position/Notify/ChangeOnly specifies whether the peripheral will avoid sending position update notifications when the position hasn't changed, to reduce the number of messages sent from the peripheral to the host.
    - LinearActuator/Position/Notify/Number specifies the number of position update notifications the peripheral will send, before it stops sending position update notifications.
- **Semantics** for child channels: READ/WRITE

  - **WRITE+READ command**: The peripheral updates the variable named by the child channel with the value specified in the payload and sends a READ response with the new value of the variable:
    - LinearActuator/Position/Notify/Interval: the payload's value must be a positive number for the variable to be updated with it; otherwise, the variable will remain at its previous value.
    - LinearActuator/Position/Notify/ChangeOnly: if the payload is *1*, the peripheral will not send position update notifications when the position does not change. If the payoad is *0*, the peripheral will send position update notifications regardless of whether the position has changed. Otherwise, the mode will not change from its previous value.
    - LinearActuator/Position/Notify/Number: if the payload's value is negative, then the peripheral will send position notification updates forever (at least until a LinearActuator/Position/Notify command is sent instructing the peripheral to stop sending position updates). If the payload's value is nonnegative, then the peripheral will send that number of position update notifications before the peripheral stops sending position update notifications.
  - **READ response**: The peripheral sends a response whose payload is the value for the position update notification parameter named by the (child) channel.
- **Semantics** for LinearActuator/Position/Notify channel: READ/WRITE+Actions
  - **WRITE+READ+Actions command**: If the payload is *1* or *2*, the peripheral will send a READ response and start sending position update notifications on the LinearActuator/Position channel based on the parameters specified by the payload and the child channels of LinearActuator/Position/Notify. If the payload is *0*, the peripheral will stop sending position update notifications and send a READ response. Otherwise, the peripheral will send a READ response and do nothing. After every position update notification is sent, if the variable associated with LinearActuator/Position/Notify/Number is nonnegative, that variable is decremented by *1*. If that variable becomes *0*, then its value is reset to *-1*, sending of position update notifications stops, and the peripheral sends a READ response for LinearActuator/Position/Notify and a READ response for LinearActuator/Position/Notify/Number.
  - **READ response**: The peripheral sends a response on the LinearActuator/Position/Notify channel with payload *1* if the peripheral is sending position update notifications with notification interval in units of number of iterations of the peripheral's event loop, *2* if the peripheral is sending position update notifications with notification interval in units of milliseconds, and *0* if the peripheral is not sending position update notifications.

Fig. 4.7: : Examples of commands and responses associated with the LinearActuator/Position channel.

4.3.3 LinearActuator/SmoothedPosition

- **Child channels**:
  - LinearActuator/SmoothedPosition/SnapMultiplier
  - LinearActuator/SmoothedPosition/RangeLow
  - LinearActuator/SmoothedPosition/RangeHigh
  - LinearActuator/SmoothedPosition/ActivityThreshold
  - LinearActuator/SmoothedPosition/Notify (child channels are the same as for LinearActuator/Position/Notify)
- **Channel names**:
  - LinearActuator/SmoothedPosition: *_s*
    - LinearActuator/SmoothedPosition/SnapMultiplier: *_ss*

        - LinearActuator/SmoothedPosition/RangeLow: *_sl*
        - LinearActuator/SmoothedPosition/RangeHigh: *_sh*
        - LinearActuator/SmoothedPosition/ActivityThreshold: *_st*
        - LinearActuator/SmoothedPosition/Notify: *_sn* (child channel names are the same as for LinearActuator/Position/Notify, except with *_s* instead of *_p*)

- **Description**: These commands relate to tracking of the (smoothed values from the) position sensor in the linear actuator.
    - LinearActuator/SmoothedPosition represents the smoothed values of the position sensor. Smoothing is done by an exponentially weighted moving average, with special case handling for detecting when a position is constant, as described in [Clarke2016].
        - LinearActuator/SmoothedPosition/SnapMultiplier specifies how responsive the smoothed position will be to changes in the raw position. This corresponds to the alpha coefficient for exponentially weighted moving averages.
        - LinearActuator/SmoothedPosition/RangeLow and LinearActuator/SmoothedPosition/RangeHigh specify the minimum and maximum, respectively, allowed values for the smoothed position. Setting these values correctly improves the accuracy of the smoothed position at the limits of the position range.
        - LinearActuator/SmoothedPosition/ActivityThreshold specifies how much the smoothed position must change when the position is considered constant for the position to no longer be considered constant.
        - LinearActuator/SmoothedPosition/Notify behaves the same way as LinearActuator/Position/Notify, but notifies the smoothed values from the position sensor instead of the raw values.
- **Semantics** for LinearActuator/SmoothedPosition/Notify and its child channels are the same as for LinearActuator/Position/Notify
- **Semantics** for child channels besides LinearActuator/SmoothedPosition/Notify: READ/WRITE
    - **WRITE+READ command**: The peripheral updates the variable named by the child channel with the value specified in the payload and sends a READ response with the new value of the variable:
        - Semantics for child channels are not yet defined.
- **Semantics** for LinearActuator/SmoothedPosition channel: READ-only
    - **READ command**: The peripheral simply sends a READ response.
    - **READ response**: The peripheral sends a response on the LinearActuator/Position channel with a payload whose value is the current smoothed position reading from the linear actuator's position sensor.

4.3.4 LinearActuator/Motor

- **Child channels**:
    - LinearActuator/Motor/Notify (child channels are the same as for LinearActuator/Position/Notify)
    - LinearActuator/Motor/StallProtectorTimeout
    - LinearActuator/Motor/TimerTimeout
    - LinearActuator/Motor/MotorPolarity
- **Channel names**:
    - LinearActuator/Motor: *_m*

    - ∗ LinearActuator/Motor/Notify: *_mn* (child channel names are the same as for LinearActuator/Position/Notify, except with *_m* instead of *_p*)
    - ∗ LinearActuator/Motor/StallProtectorTimeout: *_ms*
    - ∗ LinearActuator/Motor/TimerTimeout: *_mt*
    - ∗ LinearActuator/Motor/MotorPolarity: *_mp*

- **Description**: These commands relate to direct (open-loop) motor duty control and notification of the actuator effort for the motor in the linear actuator. Additionally, commands on child channels relate to conditions for interrupting motor controllers (such as direct motor duty control or feedback control) and stopping actuator effort.
  - – LinearActuator/Motor represents the actuator effort (roughly the duty cycle) of the actuator's motor. An actuator effort of *0* corresponds to putting the motor in brake mode, as if the actuator had a high (but finite) coefficient of friction. An actuator effort greater than *0* corresponds to moving the actuator towards higher positions. An actuator effort less than *0* corresponds to moving the actuator towards lower positions. A greater magnitude of actuator effort corresponds to a higher pulse-width modulation duty cycle for the motor, and thus more electric power delivered to the motor. Actuator efforts must be between *-255* and *255*, inclusive.
    - ∗ LinearActuator/Motor/Notify behaves the same way as LinearActuator/Position/Notify, but notifies the actuator efforts of the motor instead of the raw position values.
    - ∗ LinearActuator/Motor/StallProtectorTimeout specifies how long the motor should run without any changes to the smoothed position (on the LinearActuator/SmoothedPosition channel) before the actuator concludes that the motor has stalled. When the motor has stalled, the actuator interrupts any running motor controller (such as direct motor duty control initiated by LinearActuator/Motor with a non-zero actuator effort or feedback control initiated by LinearActuator/FeedbackController) and sets the motor duty to zero; the peripheral responses triggered by this interruption of the direct motor duty controller are discussed below.
    - ∗ LinearActuator/Motor/TimerTimeout specifies the maximum amount of time the motor should run before the actuator stops the motor. When the actuator stops the motor, it interrupts any running motor controller (such as direct motor duty control initiated by LinearActuator/Motor with a non-zero actuator effort or feedback control initiated by LinearActuator/FeedbackController) and sets the motor duty to zero; the peripheral responses triggered by this interruption of the direct motor duty controller are discussed below. Note that the controller may be interrupted before the full timer duration has elapsed, for example if the stall protector interrupts control first.
    - ∗ LinearActuator/Motor/Polarity specifies the polarity of the motor, namely whether the polarity should be flipped in software as if the wires for the motor were physically swapped.
- **Semantics** for LinearActuator/Motor/Notify and its child channels are the same as for LinearActuator/Position/Notify
- **Semantics** for child channels besides LinearActuator/SmoothedPosition/Notify: READ/WRITE
  - – **WRITE+READ command**: The peripheral updates the parameter named by the child channel with the value specified in the payload and sends a READ response with the new value of the parameter:
    - ∗ LinearActuator/Motor/StallProtectorTimeout: the payload's value must be nonnegative for the variable to be updated with it; otherwise, the variable will remain at its previous value. A value of *0* disables stall protection, while a positive value sets the timeout for stall protection.
    - ∗ LinearActuator/Motor/TimerTimeout: the payload's value must be nonnegative for the variable to be updated with it; otherwise, the variable will remain at its previous value. A value of *0* disables the timer, while a positive value sets the timeout for the timer.

        - LinearActuator/Motor/Polarity: the payload's value must be either *1* or *-1* for the variable to be updated with it; otherwise, the variable will remain at its previous value. If the payload is *1*, the motor's polarity is not flipped; if the payload is *-1*, the motor's polarity is flipped.

    - **READ response**: The peripheral sends a response whose payload is the value for the motor control parameter named by the (child) channel.

- **Semantics** for LinearActuator/Motor channel: READ/WRITE+Actions
    - **WRITE+READ+Actions command**: If the linear actuator was previously in another control mode, such as feedback control mode, it interrupts that control mode and changes the mode to direct motor duty control mode. The peripheral clamps the payload's value to be between *-255* and *255*, inclusive, and sets the result to be the actuator effort (namely, motor direction and PWM duty cycle) for the linear actuator's motor. Then the peripheral sends a READ response for LinearActuator/Motor. Then the peripheral sends a READ response for LinearActuator; if the actuator effort is *0*, the READ response will have payload *0* to indicate that the actuator is operating in direct motor duty control mode with a null duty; otherwise, the READ response will have payload *1* to indiate that the actuator is operating in direct motor duty control mode with nonzero duty. If/when the actuator interrupts the direct motor duty controller, the peripheral will send a response on the LinearActuator/Motor channel, then a response on the LinearActuator/Position channel, and finally a response on the LinearActuator channel.
    - **READ response**: The peripheral sends a response on the LinearActuator/Motor channel with a payload whose value is the current actuator effort of the linear actuator's motor.

Fig. 4.8: : Examples of commands and responses associated with the LinearActuator/Motor channel.

4.3.5 LinearActuator/FeedbackController

- **Child channels**:
    - LinearActuator/FeedbackController/Limits (its child channels are documented below)
    - LinearActuator/FeedbackController/PID (its child channels are documented below)
    - LinearActuator/FeedbackController/ConvergenceTimeout
- **Channel names**:
    - LinearActuator/FeedbackController: *_f*
        - LinearActuator/FeedbackController/Limits: *_fl* (names of its child channels are documented below)
        - LinearActuator/FeedbackController/PID: *_fp* (names of its child channels are documented below)
        - LinearActuator/FeedbackController/ConvergenceTimeout: *_fc*
- **Description**: These commands relate to closed-loop position feedback control of the actuator.
    - LinearActuator/FeedbackController is the target position (the setpoint) for position feedback control. The value should be within the range specified by LinearActuator/FeedbackController/Limits/Position/Low and LinearActuator/FeedbackController/Limits/Position/High.
        - LinearActuator/FeedbackController/Limits is the root of a tree of parameters specifying position and motor duty limits.
        - LinearActuator/FeedbackController/PID is the root of a collection of parameters specifying PID controller parameters.
        - LinearActuator/FeedbackController/ConvergenceTimeout specifies how long the controller should run the motor at null (zero) duty cycle (on the LinearActuator/Motor channel) before the controller concludes that the raw position (from LinearActuator/Position) has converged to the target position. When

convergence is reached, the controller stops; the resulting peripheral responses are discussed below. Note that the controller may be interrupted (such as by stall protection timeout or a timer timeout) before convegence is reached; the resulting peripheral responses are also discussed below.

- **Semantics** for child channels of LinearActuator/FeedbackController/Limits documented below.
- **Semantics** for child channels of LinearActuator/FeedbackController/PID documented below.
- **Semantics** for the child channel LinearActuator/FeedbackController/ConvergenceTimeout: READ/WRITE
  - **WRITE+READ command**: The peripheral updates the parameter named by the child channel with the value specified in the payload and sends a READ response with the new value of the parameter:
    - LinearActuator/FeedbackController/ConvergenceTimeout: the payload's value must be nonnegative for the variable to be updated with it; otherwise, the variable will remain at its previous value. A value of *0* disables convergence detection (so that the controller will continue running even if the position has converged), while a positive value sets the timeout for convergence detection. Disabling convegence detection functionality allows the actuator to correct any disturbances which may occur after convergence is reached, but it also leaves the actuator vulnerable to moving to account for sensor noise which doesn't constitute a true disturbance, which can produce undesired/spurious actuator motions.
  - **READ response**: The peripheral sends a response whose payload is the value for the feedback control parameter named by the (child) channel.
- **Semantics** for LinearActuator/FeedbackController channel: READ/WRITE+Actions
  - **WRITE+READ+Actions command**: If the linear actuator was previously in another control mode, such as direct duty control mode, it interrupts that control mode and changes the mode to feedback control mode. The peripheral clamps the payload's value to be between the values specified by LinearActuator/FeedbackController/Limits/Position/Low and LinearActuator/FeedbackController/Limits/Position/High, inclusive, and sets the result to be the setpoint (namely, target position) for the controller. Then the peripheral sends a READ response for LinearActuator/FeedbackController. Then the peripheral sends a READ response for LinearActuator, with payload *2* to indicate that the actuator is operating in feedback control mode. If/when the actuator stops or interrupts the feedback controller, the peripheral will send a response on the LinearActuator/Position channel, then a response on the LinearActuator/FeedbackController channel, and finally a response on the LinearActuator channel.
  - **READ response**: The peripheral sends a response on the LinearActuator/FeedbackController channel with a payload whose value is the current target position (setpoint) of the feedback controller.

LinearActuator/FeedbackController/Limits

- **Child channels**:
  - LinearActuator/FeedbackController/Limits/Position
    - LinearActuator/FeedbackController/Limits/Position/Low
    - LinearActuator/FeedbackController/Limits/Position/High
  - LinearActuator/FeedbackController/Limits/Motor
    - LinearActuator/FeedbackController/Limits/Motor/Forwards
      - LinearActuator/FeedbackController/Limits/Motor/Forwards/Low
      - LinearActuator/FeedbackController/Limits/Motor/Forwards/High
    - LinearActuator/FeedbackController/Limits/Motor/Backwards
      - LinearActuator/FeedbackController/Limits/Motor/Backwards/Low

      - LinearActuator/FeedbackController/Limits/Motor/Backwards/High

- **Channel names**:
  - LinearActuator/FeedbackController/Limits: *_fl*
    - LinearActuator/FeedbackController/Limits/Position: *_flp*
      - LinearActuator/FeedbackController/Limits/Position/Low: *_flpl*
      - LinearActuator/FeedbackController/Limits/Position/High: *_flph*
    - LinearActuator/FeedbackController/Limits/Motor: *_flm*
      - LinearActuator/FeedbackController/Limits/Motor/Forwards: *_flmf*
      - LinearActuator/FeedbackController/Limits/Motor/Forwards/Low: *_flmfl*
      - LinearActuator/FeedbackController/Limits/Motor/Forwards/High: *_flmfh*
      - LinearActuator/FeedbackController/Limits/Motor/Backwards: *_flmb*
      - LinearActuator/FeedbackController/Limits/Motor/Backwards/Low: *_flmbl*
      - LinearActuator/FeedbackController/Limits/Motor/Backwards/High: *_flmbh*
- **Description**: These commands specify input/output limit parameters for closed-loop position feedback control of the actuator.
  - LinearActuator/FeedbackController/Limits is the root of a tree of parameters specifying position and motor duty limits.
    - LinearActuator/FeedbackController/Limits/Position is the root of a tree of parameters specifying position limits.
      - LinearActuator/FeedbackController/Limits/Position/Low specifies the minimum allowed position for the PID controller to move to.
      - LinearActuator/FeedbackController/Limits/Position/High specifies the maximum allowed position for the PID controller to move to.
    - LinearActuator/FeedbackController/Limits/Motor is the root of a tree of parameters specifying motor duty limits.
      - LinearActuator/FeedbackController/Limits/Motor/Forwards is the root of a tree of parameters specifying actuator effort limits for when the motor moves forwards (towards higher positions).
      - LinearActuator/FeedbackController/Limits/Motor/Forwards/Low specifies the minimum allowed actuator effort (positive signed duty cycle) for the motor to run forwards at; when the motor runs forwards, actuator efforts below this are set to *0* to brake the motor.
      - LinearActuator/FeedbackController/Limits/Motor/Forwards/High specifies the maximum allowed actuator effort (positive signed duty cycle) for the motor to run forwards at; when the motor runs forwards, actuator efforts above this are clamped to the value specified here.
      - LinearActuator/FeedbackController/Limits/Motor/Backwards is the root of a tree of parameters specifying actuator effort limits for when the motor moves backwards (towards lower positions).
      - LinearActuator/FeedbackController/Limits/Motor/Backwards/Low specifies the minimum allowed actuator effort (negative signed duty cycle) for the motor to run backwards at; this corresponds to the minimum allowed magnitude of a negative duty cycle, but combined with a negative sign. When the motor runs backwards, actuator efforts above this are set to *0* to brake the motor.

            - LinearActuator/FeedbackController/Limits/Motor/Backwards/High specifies the maximum allowed actuator effort (negative sigined duty cycle) for the motor to run backwards at; this corresponds to the maximum allowed magnitude of a negative duty cycle, but combined with a negative sign. When the motor runs backwards, actuator efforts below this are clamped to the value specified here.

- **Semantics** for child channels: READ/WRITE
    - **WRITE+READ command**: The peripheral updates the parameter named by the child channel with the value specified in the payload and sends a READ response with the new value of the parameter:
        - LinearActuator/FeedbackController/Limits/Position/Low: the payload's value must be less than or equal to the value of the variable corresponding to LinearActuator/FeedbackController/Limits/Position/High for the variable to be updated with it; otherwise, the variable will remain at its previous value.
        - LinearActuator/FeedbackController/Limits/Position/High: the payload's value must be greater than or equal to the value of the variable corresponding to LinearActuator/FeedbackController/Limits/Position/Low for the variable to be updated with it; otherwise, the variable will remain at its previous value.
        - LinearActuator/FeedbackController/Limits/Motor/Forwards/Low: the payload's numerical value must be less than or equal to the numerical value of the variable corresponding to LinearActuator/FeedbackController/Limits/Motor/Forwards/High, and greater than or equal to the numerical value of the variable corresponding to LinearActuator/FeedbackController/Limits/Motor/Backwards/Low, for the variable to be updated with it; otherwise, the variable will remain at its previous value. Generally, this numerical value should be nonnegative, as positive numerical values correspond to forwards movement of the actuator.
        - LinearActuator/FeedbackController/Limits/Motor/Forwards/High: the payload's numerical value must be greater than or equal to the numerical value of the variable corresponding to LinearActuator/FeedbackController/Limits/Motor/Forwards/Low, and less than or equal to 255, for the variable to be updated with it; otherwise, the variable will remain at its previous value. Generally, this value should be nonnegative, as positive numerical values correspond to forwards movement of the actuator.
        - LinearActuator/FeedbackController/Limits/Motor/Backwards/Low: the payload's numerical value must be greater than or equal to the numerical value of the variable corresponding to LinearActuator/FeedbackController/Limits/Motor/Backwards/High, and less than or equal to the numerical value of the variable corresponding to LinearActuator/FeedbackController/Limits/Motor/Forwards/Low, for the variable to be updated with it; otherwise, the variable will remain at its previous value. Generally, this value should be negative, as negative numerical values correspond to backwards movement of the actuator.
        - LinearActuator/FeedbackController/Limits/Motor/Backwards/High: the payload's numerical value must be less than or equal to the numerical value of the variable corresponding to LinearActuator/FeedbackController/Limits/Motor/Backwards/Low, and greater than or equal to -255, for the variable to be updated with it; otherwise, the variable will remain at its previous value. Generally, this value should be negative, as negative numerical values correspond to backwards movement of the actuator.
    - **READ response**: The peripheral sends a response whose payload is the value for the feedback control parameter named by the (child) channel.

LinearActuator/FeedbackController/PID

- **Child channels**:
  - LinearActuator/FeedbackController/PID
    - LinearActuator/FeedbackController/PID/Kp
    - LinearActuator/FeedbackController/PID/Kd
    - LinearActuator/FeedbackController/PID/Ki
    - LinearActuator/FeedbackController/PID/SampleInterval
- **Channel names**:
  - LinearActuator/FeedbackController/PID: *_fp*
    - LinearActuator/FeedbackController/PID/Kp: *_fpp*
    - LinearActuator/FeedbackController/PID/Kd: *_fpd*
    - LinearActuator/FeedbackController/PID/Ki: *_fpi*
    - LinearActuator/FeedbackController/PID/SampleInterval: *_fps*
- **Description**: These commands specify PID controller parameters for closed-loop position feedback control of the actuator.
  - LinearActuator/FeedbackController/PID is the root of a collection of parameters specifying PID controller parameters.
    - LinearActuator/FeedbackController/PID/Kp is the proportional gain of the PID controller.
    - LinearActuator/FeedbackController/PID/Kd is the derivative gain of the PID controller.
    - LinearActuator/FeedbackController/PID/Ki is the integral gain of the PID controller.
    - LinearActuator/FeedbackController/PID/SampleInterval is the duration of the interval (in milliseconds) between updates of the PID controller's output.
- **Semantics** for child channels: READ/WRITE
  - **WRITE+READ command**: The peripheral updates the parameter named by the child channel with the value specified in the payload and sends a READ response with the new value of the parameter:
    - LinearActuator/FeedbackController/PID/Kp: the payload's value must be a positive number for it to be preserved when the variable is updated with it; otherwise, the variable's value will be a different, corrected value. The gain is a fixed-point representation of a real number, namely the real number multiplied by *100* and then rounded to the nearest integer. Thus, setting Kp to *0.1* requires sending a LinearActuator/FeedbackController/PID/Kp(10) command. The response is given in the same fixed-point representation.
    - LinearActuator/FeedbackController/PID/Kd: the payload's value must be a positive number for it to be preserved when the variable is updated with it; otherwise, the variable's value will be a different, corrected value. The gain is a fixed-point representation of a real number, namely the real number multiplied by *100* and then rounded to the nearest integer. Thus, setting Kd to *0.1* requires sending a LinearActuator/FeedbackController/PID/Kd(10) command. The response is given in the same fixed-point representation.
    - LinearActuator/FeedbackController/PID/Ki: the payload's value must be a positive number for it to be preserved when the variable is updated with it; otherwise, the variable's value will be a different, corrected value. The gain is a fixed-point representation of a real number, namely the real number multiplied by *100* and then rounded to the nearest integer. Thus, setting Ki to *0.1* requires sending a

LinearActuator/FeedbackController/PID/Ki(10) command. The response is given in the same fixed-point representation.

  * LinearActuator/FeedbackController/PID/SampleInterval: the payload's value must be a positive number for the variable to be updated with it; otherwise, the variable will remain at its previous value.

- **READ response**: The peripheral sends a response whose payload is the value for the feedback control parameter named by the (child) channel.

Fig. 4.9: : Examples of commands and responses associated with the LinearActuator/FeedbackController channel.

4.3.6 Quick Reference

Below is a list of common tasks you might want to tell the peripheral to execute, and the corresponding messages to send and wait for; additionally, there is a table of all command/response channels for a linear actuator axis. In this quick reference, replace _ with the name of your linear actuator, e.g. *p*, *z*, *y*, or *x*.

Functionalities

To query the actuator for its current position:

- Send *<_p>()*.
- Wait for a *<_p>(…)* response and read its payload.

To start receiving position notifications every 50 ms from the actuator:

- Send *<_pni>(50)*.
- Send *<_pn>(2)*.
- Wait for a *<_p>(…)* responses and read their payloads.

To query the actuator for its current motor duty cycle:

- Send *<_m>()*.
- Wait for a *<_m>(…)* response and read its payload.

To start receiving motor duty cycle notifications every 50 ms from the actuator:

- Send *<_mni>(50)*.
- Send *<_mn>(2)*.
- Wait for *<_m>(…)* responses and read their payloads.

To run the motor forwards at full power until the motor stalls:

- Send *<_m>(255)*. The motor will start running.
- The motor will eventually stop running. Wait for a *<_p>(…)* response, a *<_m>(0)* response, and a *<_>(…)* response, and read their payloads to determine the stopped state of the actuator.

To run the motor backwards at half power for 100 ms or until the motor stalls, whichever happens first:

- Send *<_mt>(100)*.
- Send *<_m>(-127)*. The motor will start running.

- After no more than 100 ms, the motor will stop running. Wait for a *<_p>(. . .)* response, a *<_m>(0)* response, and a *<_>(. . .)* response, and read their payloads to determine the stopped state of the actuator.

To make the actuator go to position 50 and stop when it reaches that position or until the motor stalls or until 6 seconds have elapsed, whichever happens first:

- Send *<_mt>(6000)*.
- Send *<_f>(50)* and wait for a *<_f>(. . .)* response to see what the actual target position was set to. The feedback controller will start running.
- The feedback controller will eventually stop running. Wait for a *<_p>(. . .)* response, a *<_m>(0)* response, a *<_f>(. . .)* response, and a *<_>(. . .)* response, and read their payloads to determine the stopped state of the actuator.

To limit the actuator between positions 20 and 400 for position feedback control:

- Send *<_flpl>(20)* and wait for a *<_flpl>(. . .)* response to see what the actual value was updated to.
- Send *<_flph>(400)* and wait for a *<_flph>(. . .)* response to see what the actual value was updated to.

To make the motor brake between duty cycles -20 and 40 for position feedback control:

- Send *<_flmbl>(-20)* and wait for a *<_flmbl>(. . .)* response to see what the actual value was updated to.
- Send *<_flmfl>(40)* and wait for a *<_flmfl>(. . .)* response to see what the actual value was updated to.

To limit the motor between duty cycles -150 and 200 for position feedback control:

- Send *<_flmbh>(-150)* and wait for a *<_flmbh>(. . .)* response to see what the actual value was updated to.
- Send *<_flmfh>(200)* and wait for a *<_flmfh>(. . .)* response to see what the actual value was updated to.

To set the PID controller to use Kp = 10, kd = 0.1, and ki = 0.5 for position feedback control:

- Send *<_pp>(1000)* and wait for a *<_pp>(. . .)* response to see what the actual value was updated to.
- Send *<_pd>(10)* and wait for a *<_pd>(. . .)* response to see what the actual value was updated to.
- Send *<_pi>(50)* and wait for a *<_pi>(. . .)* response to see what the actual value was updated to.

To make the Z-Axis actuator go to position 100, wait for 2 seconds after it finishes, then make the Pipettor Axis actuator go to position 200, wait for 2 seconds after it finishes, and then make the Z-Axis actuator go to position 900, and wait until it finishes:

- Send *<zf>(100)*; the peripheral will immediately send a *<zf>(100)* response and a *<z>(2)* response to acknowledge the command. Then wait for a *<zp>(. . .)* response, a *<zf>(100)*, and a *<z>(. . .)* response, which together indicate that the actuator has stopped moving.
- Wait 2 seconds
- Send *<pf>(200)*; the peripheral will immediately send a *<pf>(200)* response and a *<p>(2)* response to acknowledge the command. Then wait for a *<pp>(. . .)* response, a *<pf>(200)*, and a *<p>(. . .)* response, which together indicate that the actuator has stopped moving.
- Wait 2 seconds
- Send *<zf>(300)*; the peripheral will immediately send a *<zf>(300)* response and a *<z>(2)* response to acknowledge the command. Then wait for a *<zp>(. . .)* response, a *<zf>(300)*, and a *<z>(. . .)* response, which together indicate that the actuator has stopped moving.

To simultaneously make the Z-Axis actuator go to position 100 and the Y-Axis actuator go to position 360:

- Send *<zf>(100)* and *<yf>(360)*. The peripheral will immediately send *<zf>(100)*, *<z>(2)*, *<yf>(360)*, and *<y>(2)* responses to acknowledge these commands.

- Wait for a *<zp>(. . .)* response, a *<zf>(100)*, a *<z>(. . .)* response, a *<yp>(. . .)* response, a *<yf>(100)*, and a *<y>(. . .)* response (not necessarily in that order) to indicate that the respective actuators have stopped moving.

Commands Cheatsheet

Format:

- Channel Name
  - Semantics
  - Description

Commands:

- _
  - READ-only
  - The current state of the linear actuator. *1*: direct motor duty control; *2*: position feedback control; *0*: manual braking; *-1*: stopped from stall; *-2*: stopped from convergence; *-3*: stopped from timer.
- *_p*
  - READ-only
  - The current position of the linear actuator.
- *_pn*
  - READ/WRITE+Actions
  - Whether the peripheral will periodically send the position to the host with a specified interval. *0*: no notifications; *1*: event loop iteration intervals; *2*: time intervals).
- *_pni*
  - READ/WRITE
  - The interval at which the peripheral will send the position to the host.
- *_pnc*
  - READ/WRITE
  - Whether the peripheral will only send the position if it has changed since the last position the peripheral sent. Boolean payload.
- *_pnn*
  - READ/WRITE
  - The number of positions the peripheral will send before it stops sending positions. Use a negative number to send positions indefinitely (until notifications are manually disabled).
- *_s*
  - READ-only
  - The current smoothed position of the linear actuator.
- *_sn*
  - READ/WRITE+Actions
  - Whether the peripheral will periodically send the smoothed position to the host with a specified interval. *0*: no notifications; *1*: event loop iteration intervals; *2*: time intervals).

- *_sni*
  - READ/WRITE
  - The interval at which the peripheral will send the smoothed position to the host.
- *_snc*
  - READ/WRITE
  - Whether the peripheral will only send the smoothed position if it has changed since the last smoothed position the peripheral sent. Boolean payload.
- *_snn*
  - READ/WRITE
  - The number of smoothed positions the peripheral will send before it stops sending smoothed positions. Use a negative number to send smoothed positions indefinitely (until notifications are manually disabled).
- *_ss*: not yet specified/implemented
  - READ/WRITE
  - How response the smoothed position will be to changes in the raw position.
- *_sl*: not yet specified/implemented
  - READ/WRITE
  - The lowest allowed value for the smoothed position.
- *_sh*: not yet specified/implemented
  - READ/WRITE
  - The highest allowed value for the smoothed position.
- *_st*: not yet specified/implemented
  - READ/WRITE
  - How much the smoothed position must change from constant for it to no longer be considered constant.
- *_m*
  - READ/WRITE+Actions
  - The current signed motor duty cycle (-255 to 255). Positive for forward motion, negative for backwards motion. 0 for braking. Writing to this value initiates motor direct duty control with the specified signed motor duty cycle.
- *_mn*
  - READ/WRITE+Actions
  - Whether the peripheral will periodically send the motor duty cycle to the host with a specified interval. *0*: no notifications; *1*: event loop iteration intervals; *2*: time intervals).
- *_mni*
  - READ/WRITE
  - The interval at which the peripheral will send the motor duty cycle to the host.
- *_mnc*
  - READ/WRITE

  - Whether the peripheral will only send the motor duty cycle if it has changed since the last motor duty cycle the peripheral sent. Boolean payload.

- *_mnn*
  - READ/WRITE
  - The number of motor duty cycles the peripheral will send before it stops sending motor duty cycles. Use a negative number to send motor duty cycles indefinitely (until notifications are manually disabled).
- *_ms*
  - READ/WRITE
  - How long the motor is allowed to run without any changes to smoothed position (in any actuator control mode) before the actuator turns off the motor and enters the “stopped from stall” state.
- *_mt*
  - READ/WRITE
  - How long the motor is allowed to run without (in any actuator control mode) before the actuator turns off the motor and enters the “stopped from timer timeout” state.
- *_mp*
  - READ/WRITE
  - Whether the actuator should treat the motor as if its wires were flipped.
- *_f*
  - READ/WRITE+Actions
  - The current setpoint of the position feedback controller. Writing to this value initiates position feedback control with the specified target position.
- *_fc*
  - READ/WRITE
  - How long the controller is allowed to run with the motor in brake (0 duty cycle) before the controller stops and the actuator enters the “stopped from convergence” state.
- *_flpl*
  - READ/WRITE
  - The lowest position the feedback controller is allowed to target.
- *_flph*
  - READ/WRITE
  - The highest position the feedback controller is allowed to target.
- *_flmfl*
  - READ/WRITE
  - The lowest duty cycle the feedback controller is allowed run the motor at in the forwards direction. Below this value, the motor brakes instead.
- *_flmfh*
  - READ/WRITE
  - The highest duty cycle the feedback controller is allowed run the motor at in the forwards direction.

- *_flmbl*
    - READ/WRITE
    - The highest (negative) duty cycle the feedback controller is allowed run the motor at in the backwards direction. Above this value, the motor brakes instead.
- *_flmbh*
    - READ/WRITE
    - The lowest (negative) duty cycle the feedback controller is allowed run the motor at in the backwards direction.
- *_fpp*
    - READ/WRITE
    - The proportional gain of the PID controller, multiplied by 100.
- *_fpd*
    - READ/WRITE
    - The derivative gain of the PID controller, multiplied by 100.
- *_fpi*
    - READ/WRITE
    - The integral gain of the PID controller, multiplied by 100.
- *_fps*
    - READ/WRITE
    - The update interval time of the PID controller. Between these intervals, the PID controller maintains its last computed output.

4.3.7 References

CHAPTER

FIVE

LHRHOST PACKAGE

Support for host control of a connected liquid-handling robot device.

5.1 lhrhost.messaging package

5.2 lhrhost.util package

Various utilities to support other modules.

5.2.1 lhrhost.util.interfaces module

Support for specifying interface classes.

Summary

Classes

InterfaceClass	Metaclass that allows docstring inheritance.

InterfaceClass

class lhrhost.util.interfaces.**InterfaceClass**(*classname*, *bases*, *cls_dict*)

Bases: `abc.ABCMeta`

Metaclass that allows docstring inheritance.

References

https://github.com/sphinx-doc/sphinx/issues/3140#issuecomment-259704637

mro() → list

return a type's method resolution order

register(*subclass*)

Register a virtual subclass of an ABC.

Returns the subclass, to allow usage as a class decorator.

5.2.2 lhrhost.util.math module

Various functions to simplify math calculations.

Summary

Functions

map_value	Maps a value from an input range to a target range.

map_value

`lhrhost.util.math.`**`map_value`**(*x*, *in_min*, *in_max*, *out_min*, *out_max*)

Maps a value from an input range to a target range.

Equivalent to Arduino's *map* function.

Parameters

- **x** (Union[int, float]) – the value to map.
- **in_min** (Union[int, float]) – the lower bound of the value's current range.
- **in_max** (Union[int, float]) – the upper bound of the value's current range.
- **out_min** (Union[int, float]) – the lower bound of the value's target range.
- **out_max** (Union[int, float]) – the upper bound of the value's target range.

Return type float

Returns The mapped value in the target range.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[SaltzerKaashoek2009] Saltzer JH, Kaashoek MF. Principles of Computer System Design: An Introduction. 1st ed. Burlington (MA): Morgan Kaufmann Publishers; c2009. Chapter 2, Elements of Computer System Organization; p. 43-114.

[Clarke2016] Blog post describing algorithm: Writing a better noise-reduction algorithm for Arduino; Arduino library on Github

PYTHON MODULE INDEX

INDEX