Hedgehog Documentation

Release 0.0.1

Practical Robotics Institute Austria

Dec 09, 2018

Contents

Hedgehog is a robot controller: a device for controlling robotics components. Hedgehog is particularly suited for educational use, but great care was taken to make it flexible and usable in a wide range of use cases. Its most important features and properties are as follows:

- **full control over your device**
    - use all capabilities of the built-in Raspberry Pi 3
    - all open hardware and software
    - no limitations for advances users

- **easy connectivity to programming devices**
    - wired via Ethernet
    - wireless via WiFi
    - Web-based Hedgehog IDE or SSH

- **unified command protocol**
    - control your Hedgehog locally or over the network
    - protocol stack designed for reliability
    - easy to implement in many popular programming languages
    - few assumptions to allow for various approaches in programming interfaces

- **versatile application programming interfaces (APIs)**
    - visual programming via Blockly
    - textual programming mainly via Python, but node.js and Java already supported as well
    - simple, yet not limiting

- **made for tinkering and hacking**
    - compatible with RC servos & DC motors, various analog and digital sensors
    - controller case can be mounted to Lego models
    - hardware blueprints (case & circuit boards) available for modification
    - CAD software, microcontroller toolchain, git, . . . pre-installed on the controller

- **classroom ready**
    - One WiFi for all controllers avoids network congestion
    - many educational use cases, appropriate for various ages
    - visual & textual programming, closed-loop control, autonomous driving, microcontroller programming, distributed systems, swarm intelligence, . . .

Learn More

1.1 Usage Basics

Hedgehog is simple to use, but without a display to show you exactly what to do after booting up, some documentation and explanation is necessary.

1.1.1 Setting up a network connection

Although a screen and keyboard may be connected to Hedgehog directly, the default operating system does not provide a graphical user interface and thus does not allow using the Hedgehog IDE. Also, especially while built into a robot, accessing Hedgehog wirelessly is more convenient anyway.

Using the default WiFi configuration

Unless you re-installed the controller's operating system, Hedgehog comes with a WiFi pre-configured:

```
SSID: hedgehog
Key: hedgehog
Encryption: WPA-PSK
```

You can use your phone to open a WiFi hotspot with that configuration, and Hedgehog will then automatically connect with it. Using a WiFi router or similar works in the same way, of course.

Using a wired network

If you prefer a wired connection, you can use the Hedgehog's Ethernet port, but (as with WiFi) you will have to ensure that the network provides a DHCP address to the controller. If you want to connect directly to your computer, you will have to configure something like connection forwarding in a platform-dependent way. In the installation document, *Connecting via Ethernet directly to your computer* describes this for selected platforms.

Changing the network configuration

If you can't (or don't want to) use the default `hedgehog` network, there is a simple mechanism to provide alternative network configurations over a config file on a flash drive. The new configuration is stored on the controller, so this is only necessary when network settings change.

The caveats for flashdrives on the Raspberry Pi/Raspbian apply: excessively large drives (> ~64GiB) and unsupported file systems (such as NTFS) may not work. An example config file looks like this:

```
[default]
# the controller name. this can be overwritten by passing `-n`, `--name`
name = Hedgehog {mode} {mac}
# the server port. this can be overwritten by passing `-p`, `--port`
# it is STRONGLY advised to not change the port, because local programs
# connect to this port without using the discovery mechanism
port = 10789
# additional services to advertise for discovery; 'hedgehog_server' is always␣
↪advertised
# services listed here and on the command line using `--svc`, `--service` are merged
services =

[wifi]
# these commands are not stored as configuration on the controller,
# they are executed once by passing them to wpa_cli
commands =
    # "flush" clears any previous configuration; it is generally necessary to have␣
↪predictable network numbers
    flush
    add_network
    set_network 0 ssid "hedgehog"
    set_network 0 key_mgmt WPA-PSK
    set_network 0 psk "hedgehog"
    enable_network 0
    # "save_config" stores the settings to wpa_supplicant.conf, so that the settings␣
↪persist after a reboot
    save_config
```

Todo: Skipping the `default` section will currently delete that part of the configuration. Make it non-mandatory!

The relevant part here is the `[wifi]` section, containing a `commands` option. This option contains any number of indented lines used to configure WiFi using the Linux `wpa_cli` command. For details on using `wpa_cli`, refer to *the installation document*, particularly the comments on non-interactive usage. You can also refer to the Linux man pages for more detailed information:

```
man wpa_cli
man wpa_supplicant.conf
```

1.1.2 Using the Hedgehog IDE

The Hedgehog IDE is the most common interface for working with Hedgehog. Once a network connection is established, it can be accessed via a web browser. The IDE was most thoroughly tested on Chrome/Chromium, so we recommend this browser whenever possible. Other browsers should work as well, though.

In the browser, open the address `http://raspberrypi.local/`, where `raspberrypi` is the Hedgehog's host name. Write out the full address, as some browsers may think `raspberrypi.local` is a search term, not an

address.

If you changed the host name of your device, replace that part with whatever you configured. If you got a Hedgehog controller with a number on it, the host name is normally set to `hedgehog#`, with # being that number.

The browser should show a circular loading sign and quickly load the IDE's main screen.

Todo: more IDE usage

1.2 Installation

This document describes how to install the software that runs on a Hedgehog controller. If you have bought a complete controller, it should have come with everything installed and you shouldn't have to do this. If you somehow can't connect to your controller anymore, you can re-install everything as described here as a last resort.

1.2.1 Using a Hedgehog SD card image

By using a prepared image, you can save the time needed to follow the setup described below. Most often this is the way to go, as you can still install custom software on top of the prepared image, while saving time because most of the software and more recent system updates are already installed.

Installing a Hedgehog image works the same way as installing plain Raspbian, installation instructions can be found at raspberrypi.org. Of course, instead of using a Raspbian image file, a Hedgehog image is used. Hostname (`raspberrypi`), username (`pi`) and password (`raspberry`) are the Raspbian defaults, but unlike with Rasbian, SSH is enabled! We might disable SSH by default in future images.

If you are re-installing your controller, that is probably it. However, if you have a never-used hardware board or there was a firmware update, or you just want to be on the safe side, go to the *firmware installation* section.

1.2.2 Installing from scratch

Installing a Hedgehog is a simple step-by-step procedure:

- installing the operating system
- **ensuring you can run commands on the hedgehog:**
    - by connecting keyboard and monitor; or
    - by enabling SSH
- **connecting Hedgehog to the internet**
    - via Ethernet
    - via WiFi
- executing the setup scripts

If you don't want to use keyboard and monitor, Hedgehog is a *headless* device. This means that you need to access your Hedgehog over the network, and/or provide some configurations before you boot your controller. We provide instructions for both options, but configuring WiFi requires you to access the ext4 root file system, for which you may need a Linux machine. If you run another operating system and can't access the root partition, please use a wired connection or keyboard and mouse initially.

If you're using keyboard and monitor, you can still follow the headless instructions instead of equivalent headed instructions if you feel like it.

Installing Raspbian

Like any Raspberry Pi, Hedgehog needs an operating system, for which we use Raspbian. Downloads and installation instructions can be found at raspberrypi.org. There is a "lite" version without a graphical user interface, which is sufficient, but you can also install the full version.

Once you are done, don't plug in your SD card and boot your Hedgehog just yet, especially if you want to work headless. Continue with the next section.

Pre-Boot Setup in Headless Environment

Without monitor and keyboard, some configurations need to be done before you boot your controller for the first time. Even if you use computer peripherals, you have the choice to follow these instructions instead of the equivalent instructions later on.

SSH

In a headless environment, network access is necessary to control a device; with peripherals, it's optional. SSH is a simple way to run commands from another machine; newer versions of Raspbian require you to enable SSH first as a security measure. Follow the instructions for headless Raspberry Pis:

> For headless setup, SSH can be enabled by placing a file named ssh, without any extension, onto the boot partition of the SD card. When the Pi boots, it looks for the ssh file. If it is found, SSH is enabled, and the file is deleted. The content of the file does not matter: it could contain text, or nothing at all.

The instructions also contain an overview of client software you can use on your computer.

WiFi

For this, you will need to access the ext4 root file system, for which you may need a Linux machine. If you can't access the root partition, please use a wired connection, or keyboard and a monitor.

Todo: add instructions

Host Name (optional)

Again, this requires accessing the root partition.

By default, Raspbian configures a network name of `raspberrypi`. If you don't like that, or have multiple Raspberries (including Hedgehogs), you should change the hostnames to be unique (and to your liking, of course).

To do so, look into the files `etc/hosts` and `etc/hostname` on the Raspbian root file system, and change the ocurrences of `raspberrypi` to the *same* single word of your liking. For example, on Linux, this can be achieved like so:

```
# change into the Raspbian root partition, then:
sudo sed -i s/raspberrypi/new-name/ etc/hostname etc/hosts
```

Booting up & connecting

Now, put the SD card into your Hedgehog, connect the controller to a battery, and turn it on.

Whatever way you use to log in, the default credentials are `pi/raspberry`.

Connecting with keyboard & monitor

This is straight forward: as soon as the Pi has booted, you should be prompted for username and password.

Connecting via Ethernet directly to your computer

To connect the controller directly to your computer, your computer will need to act as a DHCP server. Configuring this depends on your operating system. For Ubuntu Linux, it can be achieved like this:

> *Menu > Network Connections >* (select or create an Ethernet connection) *> Edit > IPv4 Settings > Method: Shared to other computers > Save*

In addition to providing addresses via DHCP, this will also let connected devices use your internet connection - during installation, this is necessary. At other times, you may deactivate your internet connection if you want to prevent that.

Finally, use an Ethernet cable to connect your controller and computer, and make sure that the saved configuration is used.

Connecting to an existing network

If you configured WiFi or connected your Hedgehog to a router via Ethernet, the Hedgehog should auto-connect to the network and receive a DHCP address. If you use a network without DHCP (if you don't know what DHCP is, you're probably using it), we assume that you know how to configure IP addresses manually; we won't cover that here.

Now, to connect to the controller, you need either its host name or its IP address. Best, first try this (substitute your hostname):

```
ssh pi@raspberrypi.local
```

Note: `pi` is the user name and `raspberrypi.local` is the host to connect to; this is how you use the default Linux SSH client. If you use a different SSH client program, refer to its documentation.

Host name resolution is platform dependent and might not work on some platforms out of the box, especially Windows. (the actual software setup will install a package that adds Windows support, but that doesn't help for the very first connection). If it doesn't work, you need to find out the controller's IP address. If you also have a keyboard and monitor, you can simply execute this command:

```
ifconfig
```

It will show IP addresses for all network interfaces; look out for the `inet addr:` label. If you determined your Hedgehog's IP address to be, for example, `10.0.0.102`, use this command:

```
ssh pi@10.0.0.102
```

Post-boot setup

Depending on your way of connection and what you configured pre-boot, you can skip some of the following steps.

SSH (optional)

If you plan on using SSH, either now or later on, read on. Otherwise, you can skip this.

Newer versions of Raspbian require you to enable SSH before using it; this is a security measure, as SSH allows remote access to a computer. The instructions also contain an overview of client software you can use on your computer.

Connecting to a network & the Internet

During installation only, an Internet connection is required. Whenever you use SSH, you will also need a network connection, even if that network does not have Internet access.

If you're not using SSH right now and thus already have a network connection, do one of the following. Needless to say, whatever network you connect to, it needs to be connected to the Internet:

- *Share your computer's internet connection over Ethernet*,
- Connect to a DHCP-enabled wired network, or
- Configure a wireless connection as explained right below.

Note: These instructions are also useful after the initial installation, to add new network connections.

In a headed environment (or if you already have an SSH connection via another network), the `wpa_cli` command can be used to configure wireless networks interactively, like this:

```
sudo wpa_cli
> add_network
0
> set_network 0 ssid "network-1"
OK
> set_network 0 key_mgmt WPA-PSK
OK
> set_network 0 psk "secret"
OK
> enable_network 0
OK
...
> save_config
OK
```

This was for a WPA Personal secured WiFi. Configuring a WPA Enterprise secured WiFi might look like this:

```
sudo wpa_cli
> add_network
1
> set_network 1 ssid "network-2"
OK
> set_network 1 key_mgmt WPA-EAP
OK
> set_network 1 eap PEAP
```

(continues on next page)

```
OK
> set_network 1 identity "username"
OK
> set_network 1 password "password"
OK
> enable_network 1
OK
...
> save_config
OK
```

`wpa_cli` can also be used non-interactively. For example, our Hedgehogs come with the following WiFi pre-configured:

```
sudo wpa_cli <<EOF
    flush
    add_network
    set_network 0 ssid "hedgehog"
    set_network 0 key_mgmt WPA-PSK
    set_network 0 psk "hedgehog"
    enable_network 0
    save_config
EOF
```

Note the initial `flush` command: this deletes all previous network connections. This is generally necessary for non-interactive network configuration because you want to be sure the network numbers are known, i.e. start at zero.

More options and usage information can be found in the man pages:

```
man wpa_cli
man wpa_supplicant.conf
```

Host Name (optional)

Changing the host name works the same way as in the pre-boot instructions for Linux. It's necessary to reboot the controller for the change to take effect:

```
sudo sed -i s/raspberrypi/new-name/ /etc/hostname /etc/hosts
sudo reboot
```

Running the Hedgehog setup

Now with network connections figured out, we can run the actual Hedgehog setup. To do this, run the following commands:

```
curl -O https://raw.githubusercontent.com/PRIArobotics/Hedgehog/master/Makefile
make setup-rpi setup-python setup-hedgehog install-server install-ide
```

The first command will download a Makefile, the actual Hedgehog setup script. The second command runs it, performing a couple of tasks; this will download a lot of software (so make sure you don't run into a data limit, and have enough battery and time) and install it:

- configure the current locale

If you connect via SSH, the shell will use the connecting system's locale, which may not be installed and in turn lead to errors. Installing the necessary locale will prevent errors now and for subsequent connections.

- extend partition

  Before installing software, the root partition is expanded to the full SD card size to accomodate it.

- activate serial connections

  Hedgehog uses the Raspberry's serial port to talk to the hardware controller, so this needs to be enabled. This only goes into effect after a reboot.

- update system software

  The freshly-installed image may not contain latest software updates, so install them

- install additional system software

  - `git` is installed to handle Hedgehog software

  - `usbmount` allows to automatically mount USB flash drives, e.g. to auto-load configuration files

  - `samba` enables hostname resolution with Windows

- install Python

  Considerable parts of Hedgehog are written in Python, so the necessary software is installed

- install Hedgehog packages:

  - The *Hedgehog Server*

  - The *Hedgehog Firmware*

  - The *Hedgehog IDE*

  The Server and IDE are installed to start automatically. Installing the firmware requires a serial connection, and that requires a reboot.

You are now done installing the Raspberry Pi software! If you are re-installing your controller, that is probably it. However, if you have a never-used hardware board or there was a firmware update, or you just want to be on the safe side, the next section shows how to install the firmware.

1.2.3 Installing the Hedgehog Firmware

If you just installed a fresh SD card, make sure that you reboot your controller to let serial connection settings take effect:

```
sudo reboot
```

Now connect, and install the firmware like this. The server is stopped before that to make sure the serial connection is free:

```
sudo service hedgehog-server stop
make install-firmware
sudo service hedgehog-server start
```

That's it! Your controller's firmware should be properly reinstalled.

1.2.4 Tips & tricks

These are some tricks that may or may not be useful in your workflow.

Modifying image files

On Linux, the `losetup` command can be used to use an image file as a loopback device:

```
# add the -r option the work read-only
sudo losetup -P /dev/loop0 path/to/image.img
# when finished, unmount the partitions, then detach the device:
sudo losetup -d /dev/loop0
```

After setting up the loopback device, most linux systems will automatically mount the boot and root partitions. You can then inspect and even change the image contents, as if it were a real SD card.

Installing development versions

The setup shown above installs released versions of the Hedgehog software. If you want to benefit from unreleased features and bugfixes, you can also install the latest code in our repositories: just replace the `setup-hedgehog` target with `setup-hedgehog-develop`.

Note: The makefile does currently not contain a development setup for the IDE.

1.3 Architecture

Hedgehog consists of two main pieces of hardware: high-level software runs on a Raspberry Pi (called software-controller, SWC). The SWC is connected to a "shield" that grants access to actual robotics hardware. At the core of this so-called hardware-controller (HWC) sits an STM32F4 microcontroller for time-critical and low-level tasks.

1.3.1 Hardware Controller

The HWC is responsible for

- providing a steady power supply from a 6V-19V power source;
- controlling servos and motors;
- connecting analog and digital sensors with optional pullup/pulldown resistors;
- providing additional connectors for serial buses, such as SPI;
- some miscellaneous outputs: a buzzer and four status LEDs.

Except for the power supply, the HWC's microcontroller makes these features accessible to the SWC over a UART connection.

1.3.2 Software Controller

The SWC's responsibilities lie mostly in software. Hardware-wise, the Raspberry Pi's Ethernet and Wifi are used for connectivity, and additional peripherals such as webcams can be connected via USB. In addition, the SWC connects to the HWC over both UART for communication and JTAG for debugging, and includes additional connection pins for resetting the HWC microcontroller and other miscellaneous purposes. Of course, keyboard, mouse and monitor can also be directly connected via USB and HDMI, respectively.

The SWC runs Raspbian as its operating system, and hosts the Hedgehog Server and Hedgehog Web IDE. Hedgehog Server acts as an intermediary between client programs and the HWC. It makes sure that all clients can access the hardware correctly, even if there are multiple clients, and allows clients to connect bot locally and over the network using the Hedgehog Protocol.

The Hedgehog Web IDE is a web application that is hosted on the controller, meaning that an internet connection is not required for using Hedgehog. Yet, using a browser-based development environment means that no installation is required and multiple operating systems can be supported with ease. The Hedgehog IDE provides a visual Blockly programming environment and a Pyhon code editor (Ace) with syntax highlighting, code completion, folding and more.

This allows both beginners and intermediate users to write software easily. For advanced users that prefer to write code in an IDE of their choice, Hedgehog can be accessed via SSH as well.

1.3.3 Protocol

When users write their robot programs, they of course need to access the robot's hardware somehow. With Hedgehog, this is done by connecting to the Hedgehog Server and communicating using Hedgehog Protocol messages. Usually the client program will run on the controller and connect locally, but that is not required.

Once a client has initiated communication, both ends can send data at any time. Requests by clients are answered with a dedicated response, or a generic acknowledgement. In certain cases, the server may send additional updates after that. For example, after starting a child process, the server will send a message to the client when that process terminated.

The Hedgehog Protocol is based on ZeroMQ and Protobuf: ZeroMQ is a message queue library that makes it easy to do asynchronous and reliable networking, while Protobuf provides efficient and extensible serialization of datatypes. At the same time, both technologies support various programming languages. This makes it easy to implement the Hedgehog protocol in C, C++, C#, Go, Haskell, PHP, or Ruby, to name a few. Implementations in Python, node.js and Scala already exist, with the Python version being the most mature.

1.3.4 Client APIs

While using the Hedgehog Protocol directly is the most flexible way of accessing Hedgehog's hardware, most people will use an API that provides a higher level of abstraction. Goals of a client API implementation should be:

- clean and preferably asynchronous implementation of the Hedgehog Protocol;

- making all features of Hedgehog accessible to the API's users;

- providing a further, possibly limited set of APIs that are easy to use;

- achieving these in a way that fits the spirit and intent of the programming language.

The last of these goals means that Hedgehog APIs in different languages are not necessarily meant to look the same. Each implementation should use the features and conventions of their language, so that using the API feels "right". It would not make sense to force the structure of a great C API onto a Haskell one; learning proper Haskell with such a library would not work, and experienced Haskell programmers would get frustrated.

1.4 Project Structure

Hedgehog's source files are divided into several repositories hosted on GitHub. This page shortly describes their roles and relations to each other.

1.4.1 Hardware

Hedgehog_PCB KiCad schematics and layouts of Hedgehog's hardware controller.

1.4.2 HWC Software & Tools

HedgehogFirmware Firmware to run on the HWC's STM32 microcontroller, written in C.

HedgehogHWCFlasher A simple Python program for flashing the HWC microcontroller with binaries. It uses *HedgehogPlatform* To determine how to connect to the microcontroller, i.e. what interface to use and how to reset the microcontroller.

1.4.3 Common Python Packages

HedgehogUtils Collection of Python utilities for Hedgehog, but not specific to a particular part of the software. Utilities are mostly for Protocol Buffers and ZeroMQ, additionally there is a stand-alone discovery component building on both.

HedgehogPlatform Python library that detects the "platform" that Hedgehog software is run on, i.e. the kind of software controller used, and provides platform independent interfaces on top of features that differ by platform. Right now, the only feature that is abstracted in this way is access to the HWC, used by *HedgehogHWCFlasher* and *HedgehogServer*.

The only platform that is currently used for Hedgehog is the Raspberry Pi 3. In the past, an Orange Pi 2 was used instead. As there is only one platform in use at the moment, and it is hard to anticipate what platform differences may arise, this repository's code is currently somewhat stale.

HedgehogProtocol Contains protocol buffers definitions and Python classes for the Hedgehog Protocol. In addition to the classes generated from the protobuf definition, this repository contains:

- Exception classes that correspond to error acknowledgement codes

- Wrapper classes that correspond to Hedgehog protocol messages: a single protobuf message type may correspond to multiple Hedgehog message types; for example, sensor requests and sensor replies are transported as the same message type. The wrapper classes convey what Hedgehog message is represented, and provide message-specific validation.

- helpers for working with Hedgehog protocol messages over ZeroMQ sockets, such as sockets that encode and decode single- or multipart messages.

HedgehogProtocol depends on *HedgehogUtils* for its ZeroMQ and Protobuf capabilities.

1.4.4 Server Software

HedgehogServer Serves clients that use the Hedgehog protocol. The server is written in Python and builds on *HedgehogProtocol*. To handle client requests, the server communicates with the HWC, utilizing *HedgehogPlatform* to determine how to do that. *HedgehogUtils'* discovery mechanism is used for advertising the controller to its peers.

hedgehog-ide Hedgehog's web IDE. The server part uses Node.js, the browser side Angular 4; both parts are written in TypeScript. To communicate with the *HedgehogServer*, *HedgehogNodeClient* is used.

1.4.5 Client Software

HedgehogClient Python client library for Hedgehog. The library builds on top of *HedgehogProtocol*, and uses *HedgehogUtils'* discovery mechanism for finding controllers.

HedgehogNodeClient Node.js client library for Hedgehog. At the moment, this library also includes the protocol implementation for Node.js. The *hedgehog-ide* uses this to communicate to the *HedgehogServer*.

1.4.6 Miscellaneous

Hedgehog Contains this documentation, and also a Makefile that serves as the entry point into Hedgehog software installation. Executing the Makefile will clone the *HedgehogBundle* and install different components.

HedgehogBundle Bundles installation scripts into one repository. The bundle contains folders for installations of the HWC firmware, server, IDE and client.

HedgehogGui A desktop Python application based on Kivy for writing and running Python client programs. It is not actively developed, as it was superseded by *hedgehog-ide*.

1.5 Hedgehog Protocol

Hedgehog uses a client-server protocol based on ZeroMQ and Protocol Buffers. An overview is given in the *architecture document*; this document describes details about the structure and order of messages, to allow developers to implement the protocol properly.

Hedgehog uses ZeroMQ, a message queue library. ZeroMQ supports different transport layer protocols, handles reconnects transparently, and offers different kinds of unicast and multicast communication models. This means that we won't concern ourselves with stream fragmentation or recovery from disconnects, instead focusing on the structure of communication and the meaning of messages that make up the protocol.

1.5.1 Message kinds and order

In the Hedgehog protocol, a message can be one of three kinds:

A "request" or "action" is a message sent by a client to the server; conceptually, they are the same. If the message's main purpose is to retrieve data, it is a request; if it's purpose is causing a side effect, it's an action.

Requests and actions are answered by the server with "replies" or "acknowledgements". Replies carry result data, whereas acknowledgements do not; they just signal that a client's message was handled. Again, these are equivalent from a protocol standpoint, but depending on the particular message in question, one of the expressions is used. When talking about the protocol in general, they are used interchangeably.

Lastly, the server may send "asynchronous updates" to a client. These messages are triggered by server-side events, but are only sent to clients that are interested in them, e.g.:

- a client has subscribed to a sensor, and receives updates about that sensor asynchronously; or

- a client has started a child process, and receives updates about data written to stdout and stderr, or about the process terminating.

Without first sending a request that signifies the client's interest (such as subscribing to a sensor value), the server won't send asynchronous updates to a client. Likewise, after telling the server that there is no interest any more (such as cancelling the subscription), the server will cease to send updates.

Every message sent by the server can be identified as either a reply or an asynchronous update, and every request gets exactly one reply. This means that the order of replies is sufficient to match requests and replies on the client side.

A conversation might look like this:

```
Client        Server
  | -- Req 1 -> |      # client sends a request
  | <- Rep 1 -- |      # server sends reply to Req 1
  |             |
  | -- Req 2 -> |      # client sends another request
  | -- Act 3 -> |      # client sends an action, without waiting for a reply first
  |             |      #   let's assume the action subscribes the client to sensor 0
  | <- Rep 2 -- |      # server responds to Req 2 first
  | <- Ack 3 -- |      # server then responds to Act 3 with an acknowledgement
  |             |
  | <- Upd 3a - |      # server sends an update regarding the subscription established
  |             |      #   by Act 3. Upd 3a does not identify Act 3 as the source, but
  |             |      #   it's an update for sensor 0, so the client can recognize it
  |             |
  | -- Req 4 -> |
  | <- Upd 3b - |      # an update comes interspersed between a request and its reply
  | <- Req 4 -- |
  |             |
  | -- Act 5** |       # Act 5 unsubscribes from the sensor again
  |             |      #   in reality, the message takes some time in transit
  | <- Upd 3c - |      # another update comes after the client sent Act 5, but before
  |             |      #   the server processed it.
  |   **Act 5 -> |     # Act 5 arrives at the server
  | <- Ack 5 -- |      # After processing the unsubscribe action,
  |             |      #   no more Updates for sensor 3 will be sent.
  |             |
```

Multipart messages

When talking about messages, what was meant here were the units of information that are transferred between client and server. Let's shortly forget about this meaning and talk about the concept of a "ZeroMQ message".

In ZeroMQ, a message is the unit of transport: a single message is either received completely, or not at all. Messages consist of one or more "frames", with each frame basically being an array of bytes. What each frame means depends on the application; for example, ZeroMQ router sockets use header frames to identify a message's sender, so that replies (and later messages) can be sent to the correct original sender.

So, a ZeroMQ message consists of several frames, some of them containing routing information or similar metadata. In the Hedgehog Protocol, there are one or more additional payload frames, each containing a single serialized message, i.e. a request, reply or asynchronous update. Most of the time, there is only one payload frame per ZeroMQ message, but this can be used to make sure that multiple messages are executed with low latency between them, e.g. when moving multiple servos in coordination. When multiple requests are sent as part of the same ZeroMQ message, then and only then their responses will also be sent in a single response ZeroMQ message. There is no such guarantee for asynchronous updates; multiple updates may come in a single or separate ZeroMQ messages.

1.5.2 Message serialization

Up until now, the communication structure was described: how to match requests with replies, and when asynchronous updates may be sent. Of course, it is also essential to know what particular messages there are, and what they mean. Before exploring this, let's describe how messages are encoded inside ZeroMQ frames.

For serializing messages, i.e. converting messages to sequences of bytes, Protocol Buffers, or Protobuf, is used. In Protobuf, a message type specification looks like this:

```
message Msg {
    uint32 field = 1;
}
```

Each message type has a name, and consists of a number of fields, each with a data type and a numeric label. A field's data type can in turn be another message type. You can find all details at the original website, but what's important is how working with a Protobuf message type works. Let's look at a small Python example:

```python
# create & initialize message
msg = Msg()
msg.field = 42

# serialize message
msg_bytes = msg.SerializeToString()

# create empty message
msg = Msg()

# deserialize message
msg.ParseFromString(msg_bytes)

print(msg.field)  # 42
```

It's important to note that, to deserialize the message, we have to know it's a `Msg` in advance! This means there has to be a single top-level message type for the Hedgehog protocol, which is fittingly called `HedgehogMessage`, and some sort of discrimination for the wrapped message types. Protobuf gives us the `oneof` feature, which does just that:

```
// `HedgehogMessage` represents a message of any kind of the Hedgehog protocol.
message HedgehogMessage {
    // Contains any one of the different Hedgehog commands.
    // See their respective files for command information.
    oneof payload {
        // ack.proto
        Acknowledgement acknowledgement = 1;
        // io.proto
        IOStateAction io_state_action = 2;
        IOStateMessage io_state_message = 19;
        // ...skipped...
    }
}
```

So `HedgehogMessage` is at the top of the message hierarchy; the `oneof payload` contains one of several concrete message types.

1.5.3 Message types

The rest of the document will describe the different message types. Each type comes with a link to its message definition on GitHub, a short description, and the message's syntax. The message syntax describes how the message is used as a request, reply or asynchronous update, and for requests what is sent as a reply. For example, let's look at `AnalogMessage`. Here is the definition, for reference:

```
message AnalogMessage {
    uint32 port = 1;
    uint32 value = 2;
```

(continued from previous page)

```
    Subscription subscription = 3;
}
```

This is the corresponding syntax description:

```
=> (port):  analog request => analog reply
<= (port, value):  analog reply
=> (port, subscription):  analog subscribe => ack
<- (port, value, subscription):  analog update
```

This tells us that `AnalogMessage` can be used in four different ways:

- as *analog request*. The initial => denotes it is a request, => `analog reply` denotes the the kind of reply. Only the `port` field is used in this case.

- as *analog reply*. The <= denotes this is a reply message.

- as *analog subscribe*. This is an action, its reply is an acknowledgement.

- as *analog update*. The <- identifies this as an asynchronous update.

Two additional notations are used: `[field]` indicates a field is optional in that syntax, and `field1/field2` means a choice of fields (usually through a `oneof`).

It should be noted that for primitive values (such as `value`), the default value (e.g. zero) is indicated by skipping the field – one can not determine whether a primitive field was given or not. Embedded messages (such as `subscription`) on the other hand are encoded in a way that makes it possible to check for presence.

This means that *analog request* and *analog reply* can not be distinguished on the wire. However, as they go in different directions, this does not pose a problem. In cases where there would be ambiguity, one message would have to use a different field in `HedgehogMessage`.

List of message types

Acknowledgement

```
<= (code, [message]):  ack
```

IOAction

```
=> (port, flags):  IO action => ack
```

IOCommandMessage

```
=> (port):  IO command request => IO command reply
<= (port, flags):  IO command reply
=> (port, subscription):  IO command subscribe => ack
<- (port, flags, subscription):  IO command update
```

AnalogMessage

```
=> (port):  analog request => analog reply
<= (port, value):  analog reply
=> (port, subscription):  analog subscribe => ack
<- (port, value, subscription):  analog update
```

DigitalMessage

```
=> (port):  digital request => digital reply
<= (port, value):  digital reply
=> (port, subscription):  digital subscribe => ack
<- (port, value, subscription):  digital update
```

MotorAction

```
=> (port, state, amount):  indefinite motor action => ack
=> (port, state, amount, reached_state, relative/absolute):  terminating motor␣
↪action => ack
```

MotorCommandMessage

```
=> (port):  motor command request => motor command reply
<= (port, state, amount):  motor command reply
=> (port, subscription):  motor command subscribe => ack
<- (port, state, amount, subscription):  motor command update
```

MotorStateMessage

```
=> (port):  motor state request => motor state reply
<= (port, velocity, position):  motor state reply
=> (port, subscription):  motor state subscribe => ack
<- (port, velocity, position, subscription):  motor state update
```

MotorSetPositionAction

```
=> (port, position):  set motor position action => ack
```

ServoAction

```
=> (port, active, position):  servo action => ack
```

ServoCommandMessage

```
=> (port):  servo command request => servo command reply
<= (port, active, position):  servo command reply
=> (port, subscription):  servo command subscribe => ack
<- (port, active, position, subscription):  servo command update
```

ProcessExecuteAction

```
=> (*args, [working_dir]):  process execute action => process execute reply
```

ProcessExecuteReply

```
<= (pid):  process execute reply
```

ProcessStreamMessage

```
=> (pid, fileno, chunk):  stream data action => ack
<- (pid, fileno, chunk):  stream data update
```

ProcessSignalAction

```
=> (pid, signal): process signal action => ack
```

ProcessExitUpdate

```
<- (pid, exit_code):  process exit update
```

TODOs

Todo: add instructions

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/hedgehog/checkouts/latest/docs/source/installation.rst, line 88.)

Todo: Skipping the `default` section will currently delete that part of the configuration. Make it non-mandatory!

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/hedgehog/checkouts/latest/docs/source/usage-basics.rst, line 74.)

Todo: more IDE usage

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/hedgehog/checkouts/latest/docs/source/usage-basics.rst, line 104.)