Chordata

The motion capture system that you can build yourself.

How it works

By Bruno Laurencich

Proofreading: Diego Laurenti Sellers

Part of the original content that can be found at:

http://wiki.chordata.cc

Intro and motivation

The field of motion capture has been receiving a lot of attention in the last few years. Not only is the scientific community creating an incredible amount of publications presenting several types of enhancements to the different techniques involved in the discipline, but also the average person with a minimum level of technological knowledge is becoming more aware of the many areas of application.

There are several reasons for this tendency. The main one might be the fact that domestic access to high computational devices has become granted to the average user. On top of that a variety of commonly available tools allow (and sometimes require) the processing of motion capture information: from 3D creation tools and video game engines, to multiple data analysis techniques.

While the cinematographic and medical industry rely on the more expensive and accurate optical capture, inertial captures are applied to situations where portability or costs are a concern. In this field the MEMS type of inertial sensor has reduced prices while increasing accuracy. Other promising techniques require only one camera and achieve the capture through the application of Artificial intelligence algorithms are making their appearance.

We live in a historic moment when for the first time doing motion capture is (virtually) easy: all the tools to achieve it are easily available. In this scenario, the lack of a systematic guide or framework to help people implement motion capture gears draws attention. To remedy that shortcoming Chordata was developed.

Our system was specially designed to meet the following criteria:

- Being inexpensive and easy to implement for the basic user, who is just interested in getting a capture.
- Flexible enough to allow an advanced user to develop a custom system, which requires implementing a motion capture solution.

Chordata is a hardware/software motion capture framework. It constitutes a starting point to begin building a new type of motion capture, one that is available to everyone and to which anyone can contribute. It also offers a launching platform in which to build and test new ideas and techniques, or deploy new discoveries in order to share them with average users beyond the academic circles.

What this document is about

Chordata was developed to be used. Is not the consequence of an extensive research, instead it was the progressive implementation of each one of its parts that guided me through the study of the several disciplines that give life to motion capture.
A consequence of this pragmatic approach is a system about which I'm aware of every single implementation detail, but that still lacks a comprehensive taxonomy. This is a first attempt to create a complete description of how the system is implemented.

If you are reading this in a pdf format be aware that what you have in your hands is just a dump of part of the knowledge base that can be found at the Chordata's wiki (http://wiki.chordata.cc). That type of content structure allows this document to constantly grow through the contribution of whoever thinks have something to say about the Chordata system, the motion capture in general or any of the many technical fields that make that discipline possible. The starting point of this document was written by just one person during september and october 2018, but since it is meant to be a collective creation from this point on the pronoun "we" will be used to reference the author(s).

We will begin with a general description of the main parts of the system and the way that they are related to each other in §1.

An specific description of each of the specific parts will follow in §2. By "specific" we intend hardware and software designed exclusively for this system (as opposed to parts commonly available).

Power considerations can be found on §3.

A description of how the sensor data is processed can be found on §4. This chapter begins by stating that the most fundamental property that we should try to pull out from the sensors is their orientation (not the position as you might believe!). Then we will briefly describe how an orientation can be expressed using a Quaternion and how the raw data from the sensors is transformed into an orientation.

Chapter 5 will cover calibration: The differences between sensor calibration and in-pose calibration, and a detailed explanation of the latter.

A fundamental aspect of the Chordata functionality is the network communication between the gear on the performer's body and the client which is receiving the capture. A description of the forms that the network can take is included in §6, and a summary of the Chordata specific networking protocols can be found in §7.

As a way to apply all the concepts in §8 we will go through the creation of a simple client in python.

As you can see many fields of knowledge are covered in this document. There's a not negligible probability that at least some aspects of them are described in a not technically accurate manner. If you are an expert in one specific topic, and find some mistakes or ways to improve parts of this document please feel free to contribute by suggesting modifications to the wiki.

At the end of this document you can find a glossary describing most of the specific terms used in this guide, and in the Chordata documentation in general, in the way that they are intended within Chordata.

Disclaimer: This document is Work in progress, you might find some chapters still missing. Please refer to the wiki to get the updated content.

What this document is not about

Please note that this is not a guide to build the system, and won't attempt to extensively describe the basic technical knowledge required to understand the system. If you are just interested on creating your own mocap gear you will probably want to take a look the guide on which the building procedure is explained taking nothing for granted.

On the other hand this document will take the different software pieces of the system as closed entities, in the sense that even if it will describe their internal mechanisms or design, no systematic description of the source code will be included here. That type of documentation will be published together with the release of the first production version of the system.

Table of contents:

1. General description of the the system and its components

The system is composed of hardware and software parts. Most of its software and all of its specific hardware is placed and works attached to the the body of the person whose movements we are trying to capture. We call this person is **the performer**, and all the hardware and software attached to him or her is called the **host side,** as opposed to the **client side** which takes place on a computing device dettached to the performer. We take this definition from the networking jargon in part because many aspects or the system reassemble the server-client model.

The host side hardware is in part specific to the chordata system, and in part is made of commonly available components. All the hardware is interconnected forming a **hierarchy**. In the root, as the main processing unit there's an **SBC (single board computer).** All the development and testing of the system until now has been done using a Raspberry Pi 3 as the SBC, but any other one with at least one exposed i2c hardware interface and preferably a WIFI adapter can be used.

The branches and leaves of the tree can be composed of any arbitrary configuration of any of the following types of chordata specific nodes:
-Hubs
-K-Ceptors
-Custom sensing units

In this page you will find a general block diagram of the complete system. In the rest of this chapter we will describe the role of the components and how they all relate to each other. You can find detailed explanations of each one in §2.

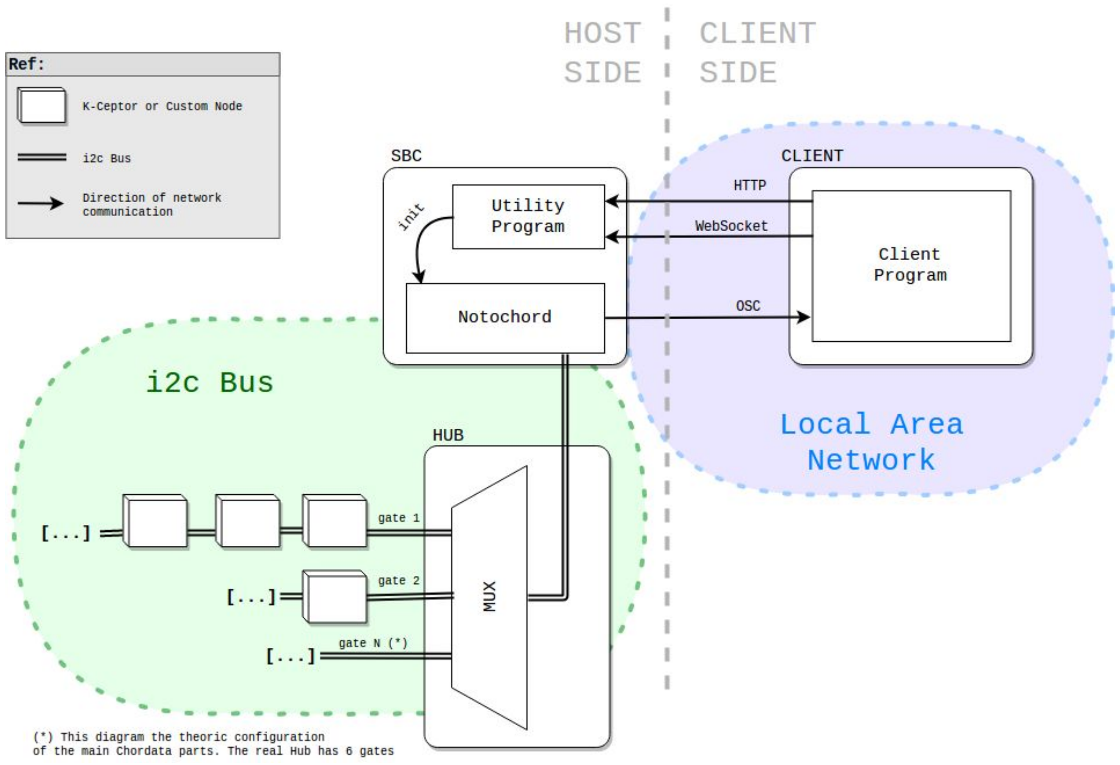


Figure 1: General Chordata Diagram

1.1 Hardware configuration

The Chordata nodes can be arranged arbitrarily but most of the time a default configuration is used, it allows a complete capture of the human body, while keeping the nodes count to a minimum. We call it the **default biped configuration** and from now on in this document we will treat it like the only possible one (just for the sake of simplicity). Keep in mind the is actually just one of the many possible ones: for example if you what to capture the movements of a dog or a robotic arm you will probably need to use a different one.

In the default biped configuration the first child is the Hub, after it six branches diverge containing the 15 K-Ceptor more or less evenly distributed among them.
Here a diagram of the default byped configuration:

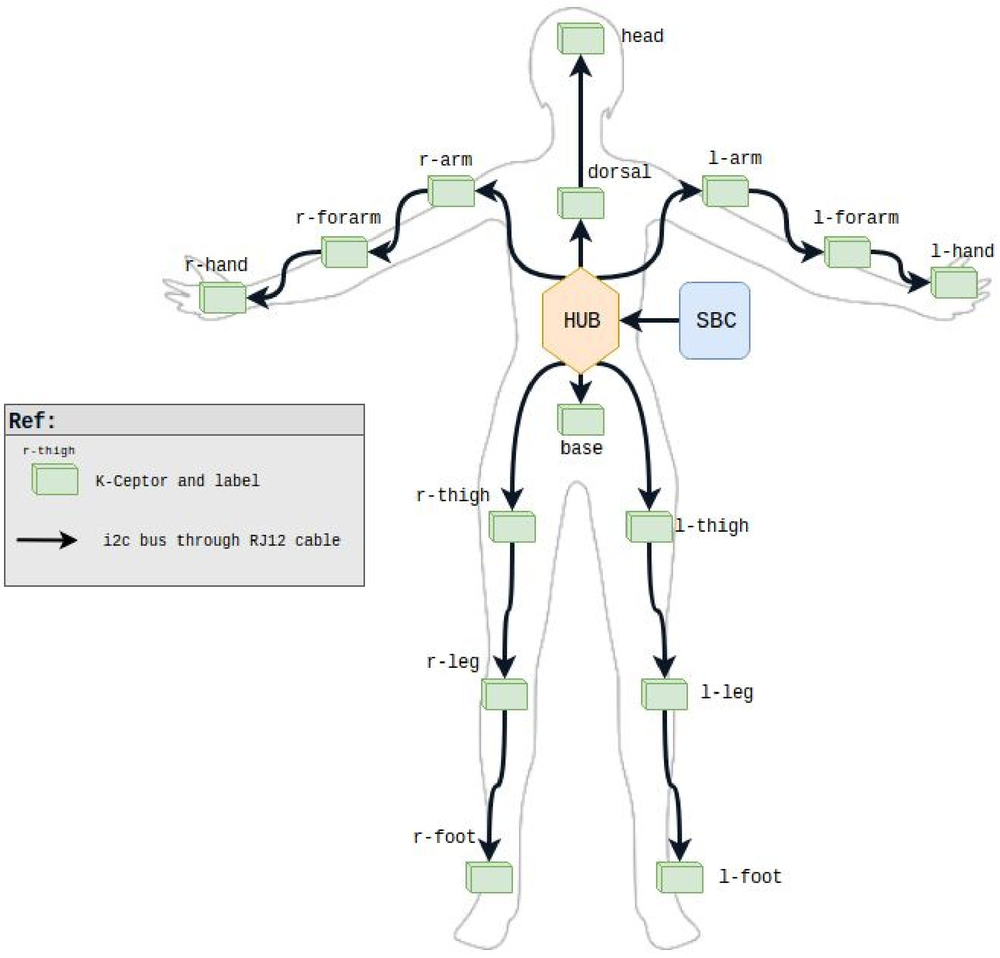


Figure 2: Default biped configuration. The most commonly used hierarchy

Custom sensing units is just a name that means "any possible sensor that might be used to complement the inertial capture". See §2.1.3 for details

1.1.1 Host side communications

On the host side all connections are wired, and the i2c protocol is used for the communications. Despite of it being at the limits of its possible physical application, it never showed issues regarding crosstalks, RF interferences or errors due to excess of capacitance on the bus. In future versions of the system alternatives to address those possible problems might be proposed.
Like many design decisions on chordata, the use of the same protocol that the sensors natively implement allows the hardware complexity to be pulled to the minimum. As a consequence the difficulty of building the system, and its derived cost gets reduced.

1.1.2 Role of the SCB

The SBC hosts the core of the system, and is responsible for all of the data processing on the host side. Apart from the processing it handles the reading or writing to the nodes of the hierarchy and the networking communication with the clients. On the hierarchy it always act as the i2c master. When communicating with a client it always assumes the role of the server[1].
It achieves these tasks through several programs and services. It's most important program is called **Notochord**[2], and by extension we sometimes refer to the hardware SBC as the notochord.

1.1.3 Role of the K-Ceptor

This is the main sensing unit of the system, and the first building block that then shapes the whole system. At its core there's a MARG sensor array[3], consisting in an accelerometer, a gyroscope and a magnetometer. No microcontroller apart from the one embedded on the MARG is used keeping the hardware configuration on this node as minimal as possible. In the hierarchy this node always act as a slave on the i2c protocol.

1.1.4 Role of the Hub

The main role of the Hub is splitting the hierarchy tree in branches. It features mainly an i2c multiplexer. Secondary functionality include:

- Physically transforming the connector on the SBC to the RJ-12 ports.
- Powering the hierarchy tree
- Performing as a simple visual interface to give the user information about the state of the system (through an RGB led).

[1] Even if, as you will see in §2: on some communications is actually implemented as a client.
[2] What about all those strange names?! Well I just happened to like the word "Chordata" which designates a biological phylum: a kind of taxonomic rank higher than the class. To get and idea it gathers together us the humans and all the vertebrates with many jelly-looking invertebrates and several strange creatures. What we all possess in common is a notochord, a hollow dorsal nerve cord and other organs on at least the embryonary state: that's what in part inspired our logo. If you are a biologist and you are reading this please don't be mad at me, I swear not to keep blindly citing wikipedia anymore :)
[3] MARG stands for "Magnetic, Angular Rate, and Gravity". That kind of sensing unit is also called AHRS (Attitude and heading reference system), and sometimes mistakenly referred as IMU (inertial measurement unit). The last form is not completely wrong, but is not taking into consideration the magnetometer.

It always acts as a slave on the i2c protocol, receiving simple commands that open or close its gates.

1.2 Client side

For the client Chordata relies on some consumer-level computing device running a specific client software. In this guide and in many places of the Chordata documentation we refer to that specific software as just the **client** . For the initial release the only client available can run on a desktop PC under any of the main operating systems, but clients to be used on mobile devices (smartphones or tablets) are planned for the future. The implementation of custom clients is relatively straightforward.

1.2.1 Current client as a Blender add-on

The current client takes the form of an add-on for the popular open source 3D manipulation software: Blender.
The reasons to use Blender as a platform for the first Chordata client are multiple. First of all it offers a Python API that can control practically all aspects of the interface of the program. It was used during development as a 3D sandbox in which to rapidly prototype and test most of the implementation steps. As a result, at the time of start creating this first release we had already created an informal mocap client library written in Python for Blender. The add-on is just a formalization of that codebase, with the addition of an Graphical User Interface (**GUI**).

On the other hand: having a rich and solid 3D manipulation tool working underneath our thin layer of specialization allowed us to offer to users a complete client, with a GUI and 3D visualization, in a fraction of the time that it would have taken to write a client program from scratch. Apart from that, many users might take advantage of some of the many tools already implemented in Blender to pre and post process the capture, edit the animations, apply them to meshes, export the capture to some 3D interchange format, or even to some other format oriented to data analysis, etc.
Not to say that we personally love Blender and we are happy to contribute to keep expanding its already broad possibilities :)

1.2.2 Future client for basic usage.

That being said, the drawback with the client as a Blender addon is that being buried in the vast availability of tools that the Blender interface offers might scare or confuse a user that just want to get a capture done[4]. Not to say that the setup process requires the installation of a completely different software, and then the activation of the addon.
To tackle those difficulties we are planning to create a simpler client software. We are still deciding the implementation details, but it should be in essence the opposite of what the blender addon is:

[4] Even if with the incoming release of the version 2.80 of Blender the GUI will suffer many changes with the explicit intention of making it friendlier to non experienced users. It is scheduled for the first months of 2019, we are looking forward to it!

- The installation procedure should be as minimal as possible, and the target platforms should be as many as possible without duplicating the codebase. Ideally it will come already hosted on the system running on the SBC, and served to the clients at the moment of use.
- It should implement just the basic features that allow the user to initialize, visualize, and record the capture. The interface should be as clear and minimal as possible.

It you carefully think about it, there aren't no so many ways to implement such a client. But we don't want to spoil the surprise. Stay tuned for some news on this aspect.

1.2.3 Custom clients.

Writing clients for chordata is easy, provided that you have access to a programing environment that can handle 3D visualization and networking. The main tasks of the client are initializing the capture and applying the incoming capture data to some armature-like 3D structure. An overview of the communication protocols follows, and they are described in detail on §7. Directives on how to build a custom client are given on §8.

1.3 Client-host communication

The softwares running on the SBC offer both an HTTP and a Websocket API to control the state of the host side. And within the capture the pose information is sent using the OSC protocol.

The **notochord control API** exposes commands that allow a simplified yet flexible handling of the capture. The same commands can be given through an HTTP GET request as key-value pairs of a query string or a token followed by argument flags in a POSIX terminal fashion through Websocket.
Examples of commands are `init`, `close`, `config` among others. A complete description of this protocol can be found in §7.

On the other hand when the notochord collects data from the sensors, after processing it sends the pose information using the **chordata pose protocol** which is built on top of OSC/UDP.
The data is structured in a really simple and straightforward manner, basically including a timestamp, the node label, and the payload that in the case of the K-Ceptors is an orientation expressed in form of a Quaternion.

The reason for this proliferation of protocols and transport layers is not arbitrary. For the control API, which issues just a few commands over a whole session the use of a reliable communication was desirable.
When transmitting the pose data packets are sent in an average rate of 700 packets/second. In this scenario losing a package is preferable to a potential network congestion due to packet retransmission, so an unreliable transport method is prefered.
Another benefit of using UDP for the pose transmission is that it supports multicasting, allowing more than one client to consume the capture data at the same time (especially useful in live performance scenarios).

2. Chordata’s parts specifications

As stated in §1 the parts of the chordata system can be classified as either hardware or software, as specially designed or commonly available. In this chapter we will go through all the specially designed parts and their specifications, and for doing so we will use a different type of classification:
We will start with the core parts, those that were first conceived and designed, and that then shaped the rest of the system.
We will then review the peripheral parts: those that serve the purposes of allowing the core to work. And to lastly the utility parts: those that are there just to make the interaction with the user easier.

2.1 Core parts

2.1.1 K-Ceptor

To do inertial motion capture the first thing that has to be done is collect data from a number of sensors. The idea of putting an MCU next to each one makes sense, but it would make the system more complex, increasing price and decreasing code maintainability.
Instead having a single MCU reading all the sensors was one of the first design choices on Chordata. The problem arises when more than two of them share the same bus, since the i2c addresses fixed in hardware[5] clashes would occur, making the bus behave in an undefined way.
The K-Ceptor sidesteps that problem by using an i2c address translator (LTC4316) to create an ad-hoc translated i2c bus branch where an LSM9DS1 MARG and an EEPROM memory are placed.

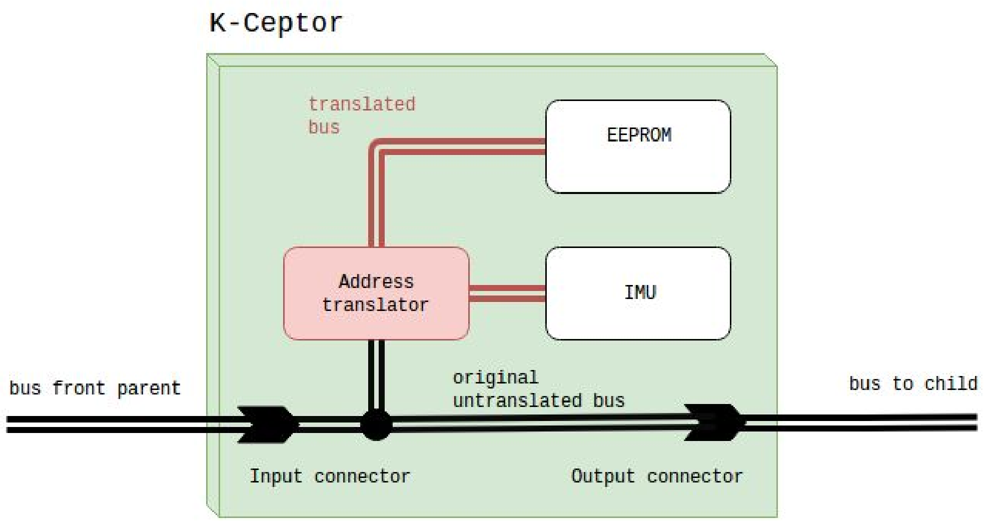


Figure 3: K-Ceptor internal configuration.

[5] The LSM9DS1 has one hardware configurable address bit, that’s why we refer to “more than two on the same bus”

The address translator behaves in a completely transparent manner from the master's point of view, changing just the address part of the message on the fly and bypassing the payload. The translation value is hardware configurable, and is applied as an XOR operation to all incoming addresses.

What figure 3 clearly shows is another of the characteristics of the K-Ceptors: they can be daisy-chained together extending the original untranslated bus and allowing the creation of linear hierarchies. The term "hierarchy" doesn't make much sense here, but it is used on several parts of the system to refer to how the nodes are connected. Keep in mind that is no describing the correct relation between the K-Ceptors on the bus, on which they are all siblings and should have an unique translation value (more on translation values on §2.2.3).

The EEPROM memory is used to store the calibration information and other details of the node (see appendix A).

2.1.2 Notochord

The notochord is the main program running inside the SBC. Its role is not extremely complex: Upon launching it has to figure out how the hierarchy is arranged (**armature parsing**), calculate the correct order to read all the nodes (**scheduling**), process the data obtained from each sensor and output it using one or more from a couple of configurable channels.

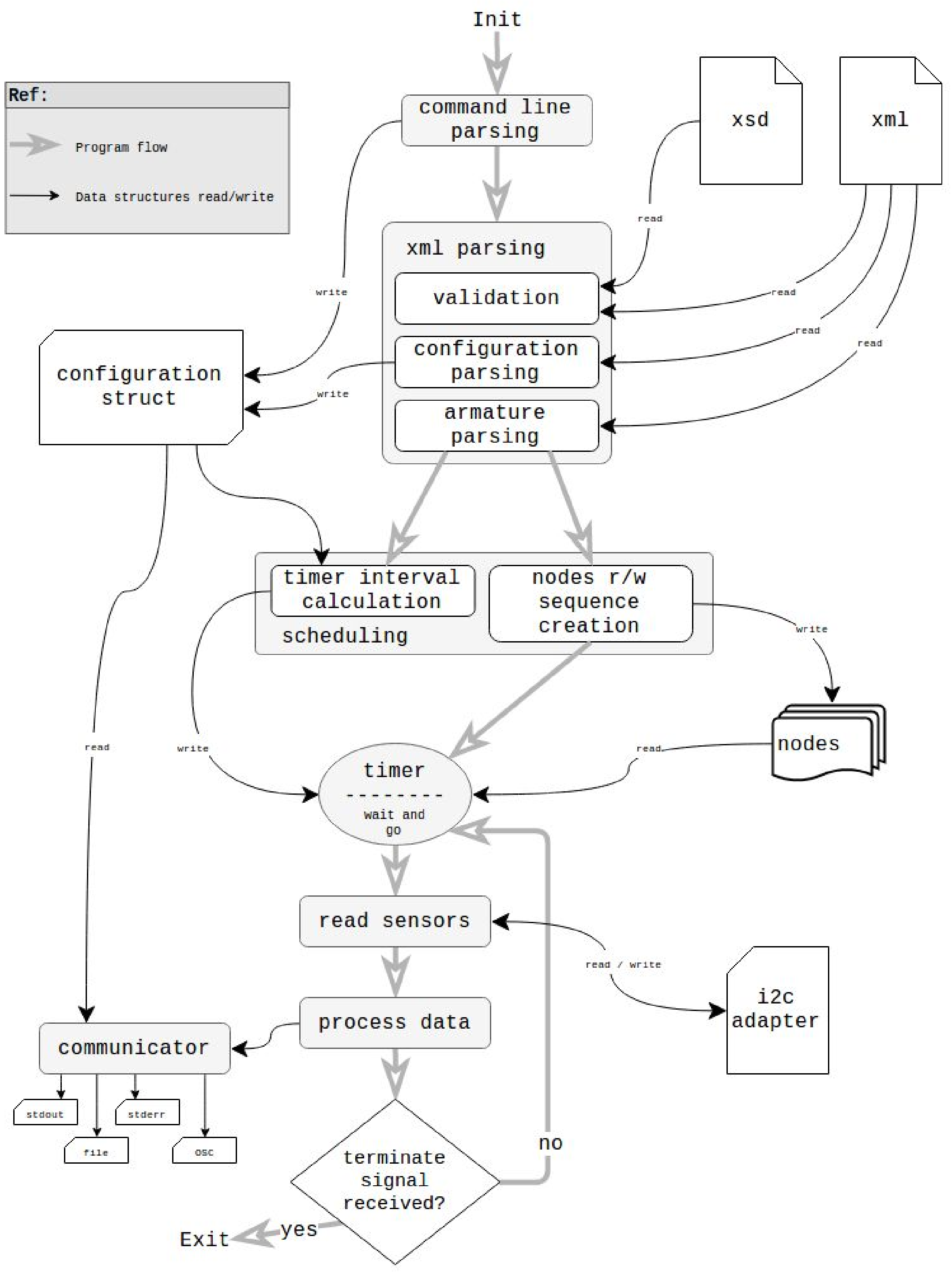


Figure 4: Notochord program functional diagram.

It’s a command line interface program written in C++. A complete description of its usage and configuration details is beyond the scope of this document. You can type `notochord -h` to get a list of all the possible arguments it can take.

2.1.2.1 Notochord: Armature parsing

Notochord ain’t no sorcerer, and it can’t guess how the hierarchy is set. Instead it needs to be told how. This is done by an xml file: the `Chordata.xml`. Apart from containing much of the configuration parameters that can also be passed as command line interface arguments, it contains a description of the hierarchy. If you want to know exactly how i is defined you can take a look to the `Chordata.xsd` file, which is a formal description of the `Chordata.xml` file for the purposes of validation. But to better grasp how it works what about some examples?

To describe a hierarchy like this:

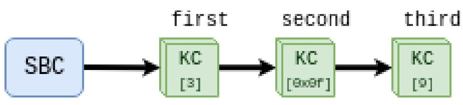


Figure 5: Simple linear hierarchy with 3 K-Ceptors.

The armature part of the xml file would look like this:

```
<Armature>
     <K_Ceptor Name="first" id="0">
          3
          <K_Ceptor Name="seconf" id="1">
               0x0f
               <K_Ceptor Name="third" id="2">
                    9
               </K_Ceptor>
          </K_Ceptor>
     </K_Ceptor>
</Armature>
```

Ti should be clear now why we are always talking about “hierarchy” when referring to the arrangement of nodes. On the `Chordata.xml` nodes can have children and parents, representing the way they are physically connected to each other[6]. The value contained on each K-Ceptor node is its translation value (expressed as a decimal of hexadecimal number),

[6] Apart from the physical connection of the nodes, the term hierarchy is inherited from the 3D skeletal animation techniques where bones are arranged on a hierarchical configuration. Generally speaking, on 3D a child element transformations are expressed in terms of its parent’s coordinates system. As a consequence the child “follows” all the parent “movements”

and the name property will be used to match it with the corresponding bone in the virtual skeleton.

Let’s take a look to a slightly more complex configuration:

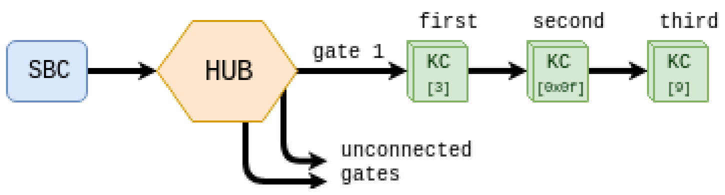


Figure 6: Branching hierarchy with 1 Hub and 3 K-Ceptors.

And its xml representation:

```
<Armature>
    <Mux Name="main" id="0">
          0x77
      <Branch Name="left" id="1">
                CH_1
           <K_Ceptor Name="first" id="2">
                     3
                <K_Ceptor Name="second" id="3">
                          0x0f
                     <K_Ceptor Name="third" id="4">
                               9
                     </K_Ceptor>
                </K_Ceptor>
           </K_Ceptor>
      </Branch>
    </Mux>
</Armature>
```

Here we introduced the Hub, which has many gates: Both these elements should be reflected on the xml. While a `<Mux>` node can have many `<Branch>` nodes inside, only one `<K-Ceptor>` is allowed inside the `<Branch>` of `<K-Ceptor>` nodes.

Whatever hierarchy this file contains, the notochord reads it on startup, validates it against the `Chordata.xsd` file and then parses it creating an internal representation of its contents.

2.1.2.2 Scheduling

In order to be able to correctly process the data the time interval between reads has to be known. There are some fancy techniques to do it, some of them take advange of the interrupt signal that the sensor emits when data is available. Notochord just goes for the

simpler of them: Determine a decent output data rate (ODR) for each sensor taking into account their number and the bus capacities. Calculate the sleep time that the program would need to wait between each lecture to approximate as best as possible the theoretical ODR, taking into consideration all the sensors and also the opening and closing of the gates. Accommodate a sequence of reads and writes needed to perform a complete round and then.. Just go for it!.
A couple of control mechanisms complete the process, but basically that's how it works.

2.1.2.3 Data processing

Once the data is obtained, most of the time it should be processed in order to pull out something significant of it. When the K-Ceptor is our sensing node processing means converting the 9-dimensional reading from the accelerometer, gyroscope and magnetometer into an absolute orientation. This process is called sensor fusion, and is briefly described in §4.

For custom sensing nodes a processing routine should be implemented in C/C++ or even as a Matlab/Octave script. Of course the latter wouldn't be the most efficient way to do so, it would allow the implementation of new routines without the necessity of recompiling the program, perhaps they perform just simple calculation or might be useful as a testing step for a later formal implementation.
If you are asking yourself how on earth might the notochord be able to process those kind of scripts is because I forgot to mention it.. a GNU Octave interpreter is already implemented inside it. It was used at the beginning for quickly testing different sensor calibration algorithms, and now is there, why not use it? Take a look at the next section (§2.1.3) for information about the current state of the custom sensing nodes.

2.1.2.4 Data output

By far the most common and useful way of outputting the data is transmitting it using the open sound control (OSC) protocol[7]. But the notochord has the ability to configure the output on different ways to suit the needs of the user's requirements.
To start with: the data processing step can be bypassed to output the raw data as it was collected from the sensors, that's achieved by passing the `-r` or `--raw` flag on startup.
The verbosity level can be set with `-v <0-2>` or `--verbose <0-2>`.

The messages generated inside notochord can be of any of these three types:

- Log (Messages to the user)
- Error (Errors and warnings)
- Transmission (The important ones)

[7] As the name suggests: this protocol was born to be used to control synthesizers or other devices for the purposes of musical or multimedial performances (like a MIDI on steroids to make it short). It turned out to be a pretty useful protocol for transmitting many types of (usually short range) time sensitive information, so it was widely adopted in the fields of computer-based new interfaces, distributed music systems, inter-process communication, among others. OSC's advantages include been a simple yet flexible and accurate protocol with a detailed documentation, a numerous community and the availability of implementations on almost all programing languages. Take a look at http://opensoundcontrol.org/introduction-osc for more information.

Even if two of them obviously takes its name from the output method they use by default[8] they can be arbitrarily piped to one or more of these sinks

- Stdout
- Stderr
- File
- Osc
- None

These pipings can be set from the configuration part of the Chordata.xml or from the command line interface. Following some valid examples:

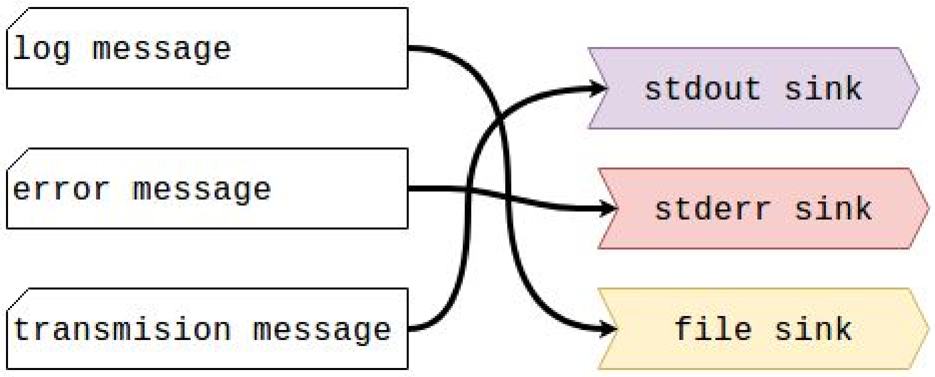


Figure 7: First example of message piping in notochord.

For a piping scheme like the one in figure 6 the arguments for the call to notochord should be:

```
notochord --log file --error stderr --transmit stdout
```

(using long option arguments)

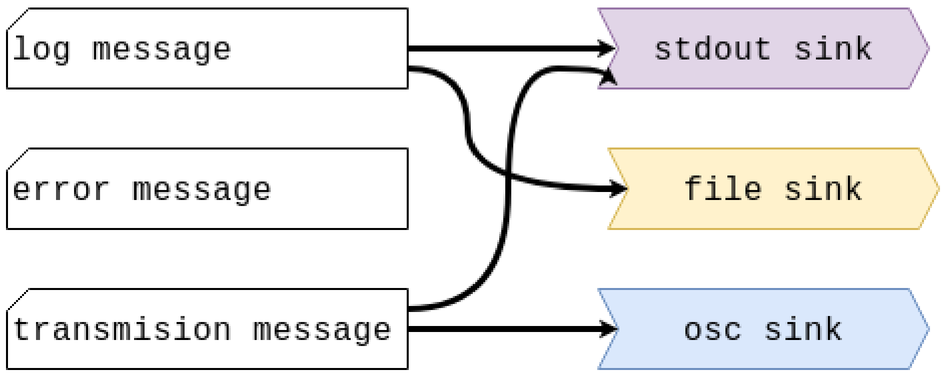


Figure 8: Second example of message piping in notochord.

Or to get something like the figure 7:

```
notochord -l stdout -l file -e none -t osc -t stdout
```

[8] Yeah, I know.. Is not the smarter thing to do. We might change this naming on futures releases.

(using short option arguments, and using more than one output for some of the messages types: notice how the argument is repeated with a different sink)

Or this is the same as before but using the xml configuration

```
<Configuration>
    <Communication>
      <Log>
           stdout, file
      </Log>
      <Error>
           none
      </Error>
      <Transmit>
           osc, stdout
      </Transmit>
    </Communication>
</Configuration>
```

There's more: it provides a well defined mechanism for implementing custom sinks. For the most part the process is described on the documentation of the library used for logging in notochord: spdlog
(https://github.com/gabime/spdlog/wiki/4.-Sinks#implementing-your-own-sink).
Expect more detailed information on how to do in notochord with the release of the first production version and the documentation for the C++ source code.

2.1.3 Custom sensing nodes

At the moment the custom sensing nodes are more a design decision than an actual feature. It's name to say: Any given sensor that might contribute to some scope complementary to the capture, and that can be included in the hierarchy in the same way that the current nodes can. The K-Ceptor is formally nothing more than a custom sensing node, but it is one that already ships with the system.
Just to name another as an example: many research and expressive areas would greatly benefit with the addition of electromyography (EMG) data to the capture.

There's of course the possibility of modifying the code, and building a hardware in a manner compatible with the rest of the system. But Chordata still lacks a specific mechanism to make it easy for someone alien to the codebase to do so. It would also be great to take advantage of the already implemented GNU Octave interpreter as a way to quickly apply processing algorithms for new the sensors, but it's still work to be done.

We are hoping to be able to implement such features in the following versions.

2.2 Periferic parts

2.2.1 Hub

The most important of the peripheral parts is without doubt the Hub. Apart from allowing the hierarchy to be splitted in branches, it distributes power to all the nodes (apart from the SBC), physically translates the connectors from the hub to the RJ-12 type, and serves as a basic visible interface keeping the user informed about the state of the system.

Its most fundamental task (branching the hierarchy) is accomplished thanks to the PCA9548 i2c multiplexer. That component receives commands from the same bus that then distributes by opening and closing its gates. The current version of the Hub exposes 6 gates as RJ-12 6p6c connectors. As you can see on figure 9 the i2c pair takes the sides of the connector, paired with a ground connector, this pattern provides some crosstalk and interference proofing. The other two pins of the connector are used for the 3.3v rail and an ENABLE signal coming from the SBC.

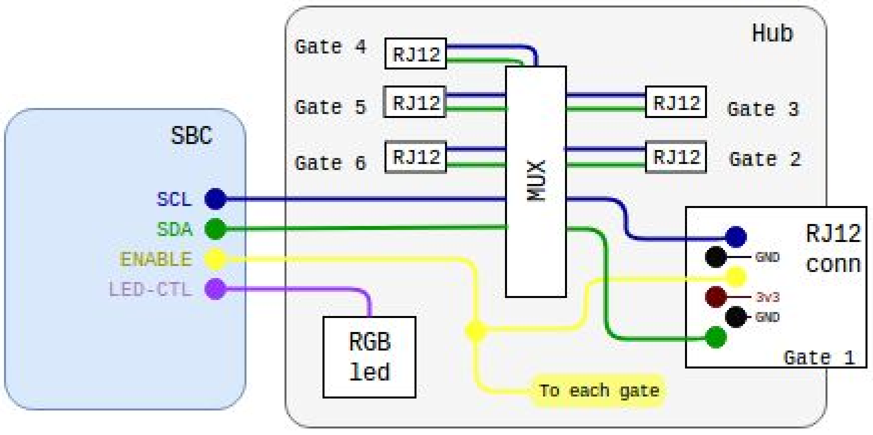


Figure 9: Scheme of the connections on the Hub.

The multiplexer is never asked to open more than one gate at the same time. This strategy reduces the possibility of address clashing coming from nodes on completely different branches and helps to reduce the problem associated with an excess of capacitance on the bus by reducing the length of cable present on the bus at any given point in time to just the length of one of the branches.

We said that the hub distributes power to its children, but where does the power came from in the first place? The short answer is that it can use the 3,3v power rail on the SBC or optionally a dedicated 5V inlet in the form of a micro USB B connector. More information on power handling on §3.
The Hub also serves as a simple visual interface. It has a pilot led to indicate correct power supply and an RGB led that informs about the state of the system, for example it turns ORANGE when the system has finished booting, WHITE the notochord is transmitting data, etc. A complete list of all the states and their colors can be found on the chordata user guide.
This RGB led is controlled from the SBC using the WS2818 single wire protocol.

2.2.2 Client

A Chordata client is not a definite structure, the only task that it has to mandatorily accomplish is receive the capture data. As you will see in §8 Clients can take many forms and be as simple or as complex as needed. Here we will discuss how a generic client would work. And by "generic" we intend one that allows a user without programming experience to do the more common tasks on the motion capture, namely: controlling the state of the notochord, building and sending the representation of the working armature in the form of a Chordata.xml, doing the calibration procedure, visualizing the capture, and finally doing something useful with it, like recording it for later use, etc. The only client with those characteristics in existence is the Blender addon, so in this section we will describe what's happening inside it.

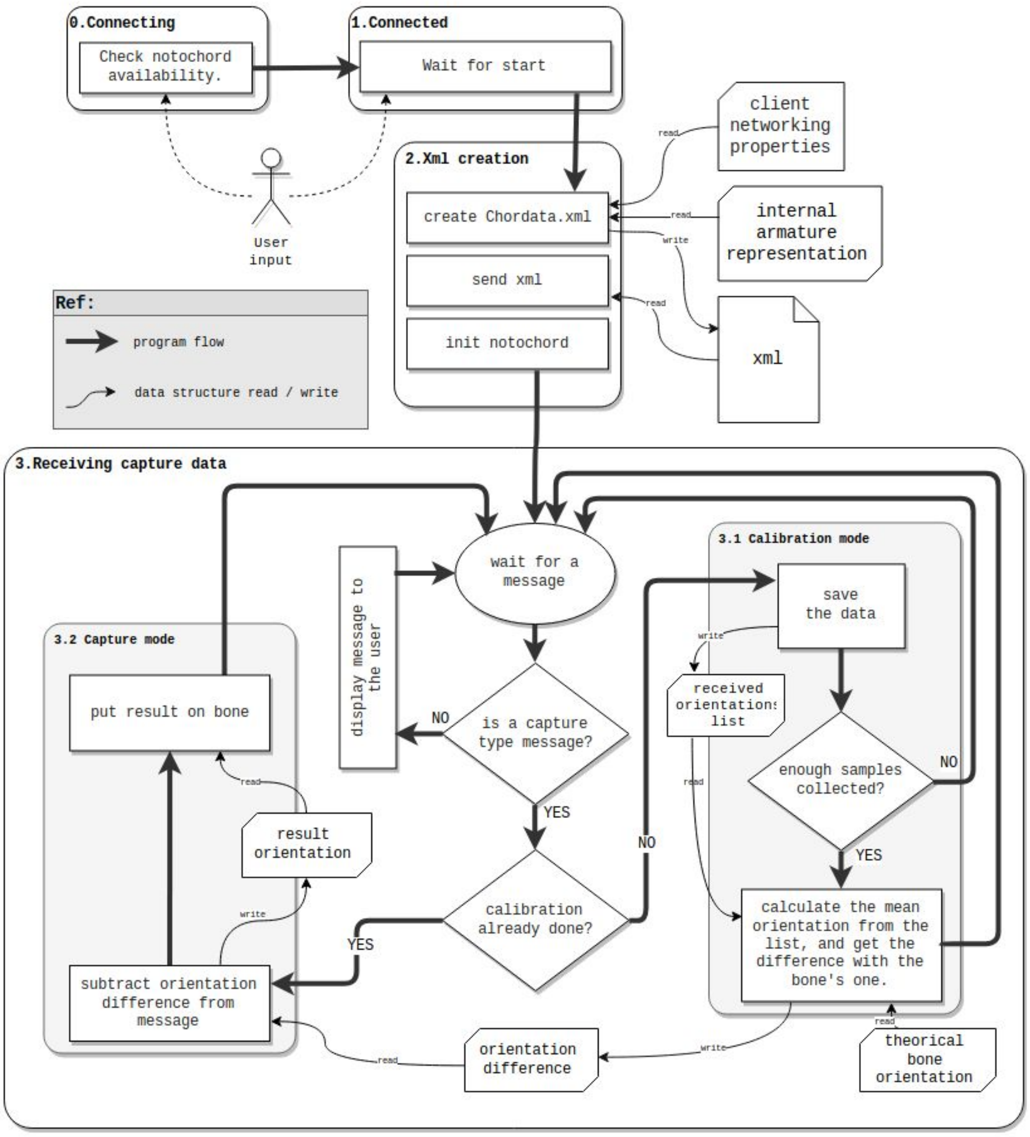

0.Connecting
Check notochord availability.
1.Connected
Wait for start
User input
client networking properties
internal armature representation
xml
2.Xml creation
create Chordata.xml
send xml
init notochord
read
write
Ref:
program flow
data structure read / write
3.Receiving capture data
wait for a message
display message to the user
is a capture type message?
NO
YES
calibration already done?
3.1 Calibration mode
save the data
received orientations list
enough samples collected?
calculate the mean orientation from the list, and get the difference with the bone's one.
theorical bone orientation
orientation difference
3.2 Capture mode
put result on bone
result orientation
subtract orientation difference from message

Figure 9. Finite state machine representation of a generic client's flow. It's a simplified version that just takes into consideration one bone and many control structures or transitions resulting from errors are omitted.

Blender already offers 3D visualization, has an internal representation of a 3D skeleton, exposes a GUI to interact with it. Talking about creating a Chordata client on top of it sounds like cheating: all the harder parts are already done. But the good news is that nowadays it's not hard to find an environment on your programming language of choice that is able to give you access to all those features more or less out of the box.

Back to our Blender addon: everything starts with an armature on the 3D view. It can be a custom one, or we can ask the addon to pull out an example our beloved default biped configuration. If the user goes for the custom one he or she would have to give some extra information in order to allow the client to map the bones to the distribution of sensors and hubs in the performer's body.

With the base armature already set the next step is to `Connect` to the notochord. The Blender client uses exclusively the HTTP version of the control API, so no long lasting connections are ever established. Instead when the user clicks `Connect` what's happening is just a connectivity test to assure that there's a notochord ready to start capturing on the other side.
If the "connection" was successful the user can then `Start` the capture. Calling this operator triggers several actions: first it generates the `Chordata.xml` containing not only the representation of the hierarchy, but also the configuration that will make the notochord transmit the data to the correct place. Then it sends the file issuing an HTTP PUT request. On success it will issue an HTTP GET request with the command "init". That will cause the notochord to start, read the configuration on the xml which will tell it to point all of its outputs to OSC and use the client's ip as a destination.
In the meantime the client has opened an UDP socket and has started listening for the messages from the notochord on it.

When a message arrives the client checks the type: `log` or `error` messages are shown to the user, and might cause the operator to terminate.
Other types of messages contain the capture data, those are used to directly modify the internal representation of the armature. Blender then takes care of displaying the updated armature to the user.
To be more precise what comes in a `capture type` message is basically the label of the K-Ceptor and an orientation in the form of a quaternion. The client just grabs the orientation and applies it to the bone with the same label.

If it wasn't for a little detail that would be the end of the story of our client's hard work. Instead it has to also take into consideration the in-pose calibration which is the process of correcting the difference in orientation of the sensor with respect to the theoretical orientation on the bone. If the last sentence doesn't makes sense to you that's completely natural, a more detailed explanation of the in-pose calibration process will be given on §5.2. For now what you need to know is that during a short period of time (calibration mode) on the start of each capture session the orientations are not directly assigned to a bone, but instead stored and used to get the difference in orientation of which we talked before. Once the calibration is done the program enters "capture mode" and the incoming messages are used to modify the orientation of the bones, But! The difference obtained in calibration is subtracted to the incoming orientation before assigning it to a bone.

I know, your mind probably hurts by this point. If that's not the case you have probably heard of these processes before. Thing is: In-pose calibration is one of the trickiest parts of the client's job. **It would be awesome to let the notochord do it**, greatly simplifying the job of writing a Chordata client. Doing it is one of the **main priorities on the roadmap for future versions of Chordata.**

At this point we have covered all the steps on a generic client, except for the specific part: How is the capture used? The options given to the user on the Blender addon are: recording it for later use, use scripting to process it in real time, or re transmit it to another program. We won't cover how any of those tasks are achieved, since it's the job of the client writer to decide what to do with the data, and how to implement it.

2.2.3 ID_Module

TODO.

2.3 Utility parts

2.3.1 Notochord control server

The notochord is a command line interface program, in order to run it a user should have access to a terminal on the SBC. An ssh client might provide a convenient way to do it[9], but opening an ssh session each time you want to perform a capture might not be the more portable and user friendly technique. Besides making a script initialize a program on a remote computer through ssh doesn't sound like a easy task, I just don't want to even think how it can be done.
As an alternative everytime the Chordata's SBC finishes booting, launches a service that provides an HTTP and Websocket interface through which a user can perform the common operations that a capture requires. We call it the notochord control server

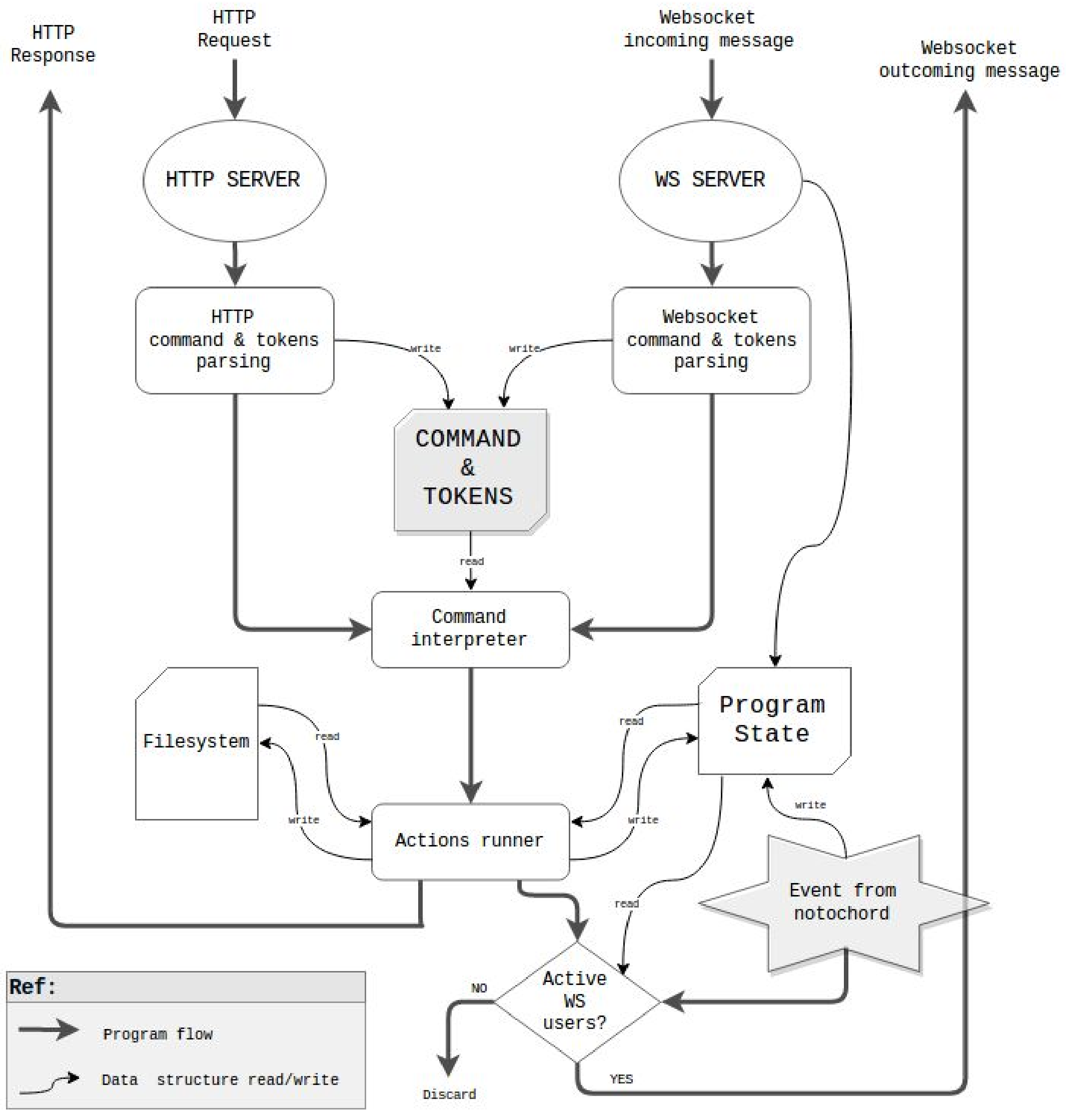


Figure 10. Diagram of the notochord control server.

[9] Because trying to attach a monitor and a keyboard to an SBC that is worn by a performer who has to move might not be much fun.

If you have already read §2.2.2 who might already have an idea of how it works. As a recap: the most important command that can be given to the server is "init" which might take any number of argument and forward them to a call to the notochord program.
If passing all the configuration parameters as arguments to the init command results on a excessively verbose construction, or if a description of the current physical hierarchy is not already on the notochord directory, then the command `put` should be used to pass a `Chordata.xml` file[10].

The program is written in python and has 3 main parts: An HTTP server, a WebSocket server, and a single command interpreter and actions runner. This means that regardless of the channel on which a message arrives, the possible commands and arguments are the same, and the state that they might query or modify is shared.

In terms of functionality the only thing that changes is that the websocket part is listening for messages from the notochord while there's at least one connected client. If there's some news about the state of the notochord the websocket users will receive a message immediately, while an HTTP user will have to continuously poll for that information.

And there's another formal difference between both communication channels: the syntax with which the command and arguments are passed. Let's see as an example how a call to notochord with a destination ip and a verbose argument number is issued. A complete description of the API can be found on §7

This is the HTTP version (non url escaped)

```
http://notochord.local/command/init?p=-v&p=2&p=192.168.85.2
```

And in the WS version, after connecting to ws://notochord.local:8000 you need to send a message as TEXT like:

```
init -v 2 192.168.85.2
```

This duplication of protocols might seem bizarre. Why not just choosing one and making it work fine? Isn't this duplicating the job of maintaining and expanding the code?
Well, those are not completely incorrect considerations. The implementation of each of the protocols just happened as a consequence of the creation of different clients while in development. Now they both are there, and keeping them live doesn't really implies an additional job, since (we hope) the servers and parsing part will remain for the most untouched while the cmd interpreter can keep growing.
On the other hand having the possibility of using both of them can really facilitate the job of writing a client. For example let's consider how a Blender addon is allowed to work: it can offer operators to the user to interact with the program, but those operators have to completely end execution before returning to the user the control of the GUI. In such case having the possibility of issuing self conclusive HTTP command is best suited.

[10] The `put` command can be used to pass any file and place it in a directory relative to the notochord one.

Instead in some other applications the bidirectional communication of the Websocket just makes the life of the programer easier. In the next section there is an example of such application: a terminal like interface for the control API where the user get the messages emitted from the notochord as soon as they exit the program. The same idea can be implemented using asynchronous HTTP requests, but it would make the client code bulkier and more error prone.

2.3.2 Remote Console

TODO.

2.3.3 WS2812 Server

TODO.

3. Powering the system

To power the system a 5V, 2A source is needed. It can be connected to the SCB only, or also to the Hub. Both connections use a micro USB-B connector.
By far the easiest way to achieve it is to use a common power bank. It should be rated for at least 2A.

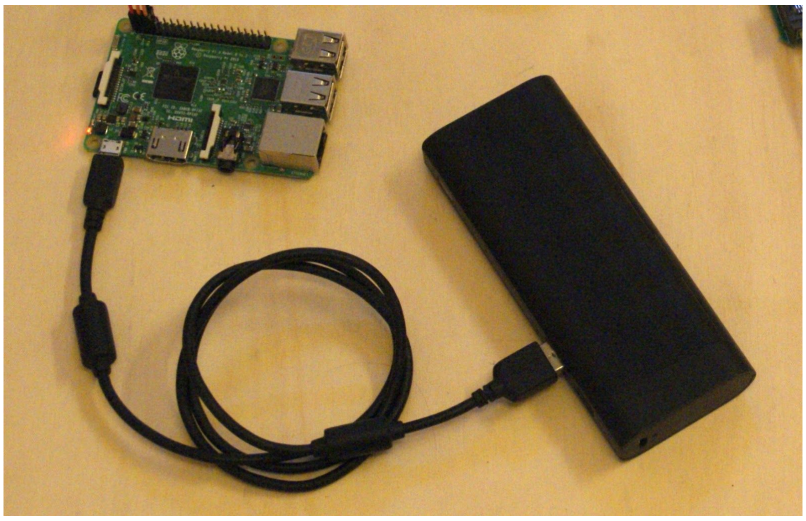


Figure 11. 10400mAh Power bank and Raspberry Pi.

To have a durable supply a capacity of at least 10000mAh is desirable. Under normal capture conditions such a power bank was able to keep the system running for around 8 hours.

3.1 Power rates

The SBC is the major contributor to the power consumption on the host side of the system. A Raspberry Pi 3 consumes about 500mA when in idle, and about 1500mA when under heavy computing requirements. The manufacturer recommends a power supply rated at at 2A or more. In the Chordata system the SBC should always get a direct connection to the power source.

The Hub is responsible for distributing power to the rest of the hierarchy. Only the 3.3V rail is used from this point on.

On regular usage a K-Ceptor withdraws about 7.8mA, so a complete default biped configuration should use about 120mA.

In a worst case scenario the whole host side should be consuming about 1650mA. Not far from the rating of our recommended power bank, but we can leave our concerns in peace by knowing the fact that under normal usage conditions the notochord is just using about 8% of the Raspberry Pi 3 computing capacity which result on a consumption of about 700mA for the SBC.

3.2 Power connection

As stated before, one power connection should always go directly from the supply to the SBC. That being said, there's a couple of option to power the rest of the system.

3.2.1 Using the Hub Buck converter.

In this mode an additional connection from the power source to the Hub's micro USB-B inlet is used. The voltage is converted to 3.3V by a MCP1603 Buck converter and then distributed to the lower part of the hierarchy tree.

Most 2A powerbank come with tho USB-A output connectors. If that's the case then one can be used to power the SBC and the other for the Hub.
If just one is available the Hub provides a 5V power outlet in the form of an USB-A connector, allowing the use of a normal USB cable to give power to the SBC

3.2.1 Using the SBC's 3.3v rail.

If the SBC exposes a pin with a constant 3.3 voltage, then it can be used to power the Hub and the rest of the hierarchy tree overstepping the Hub's Buck converter.
Figure 12 shows the wiring from a Raspberry pi 3 using female jumpers.

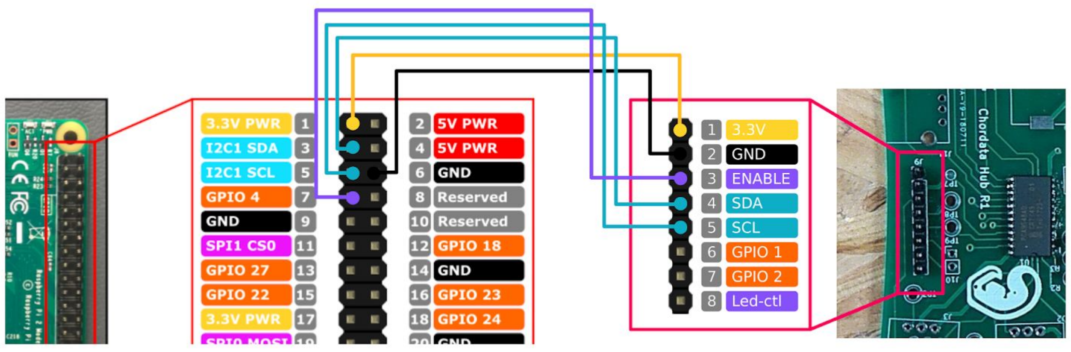


Figure 12. Wiring the Rasberry Pi 3 and the Chordata Hub to use the SBC's 3.3v power rail.

4. How the data from the sensors is used

TODO.

5.Calibration

Sensor calibration

TODO.

In-pose calibration

TODO.

6 .Network configurations

TODO.

7. Communication protocols

TODO.

8. Write your own Chordata client.

TODO.

GLOSARY:

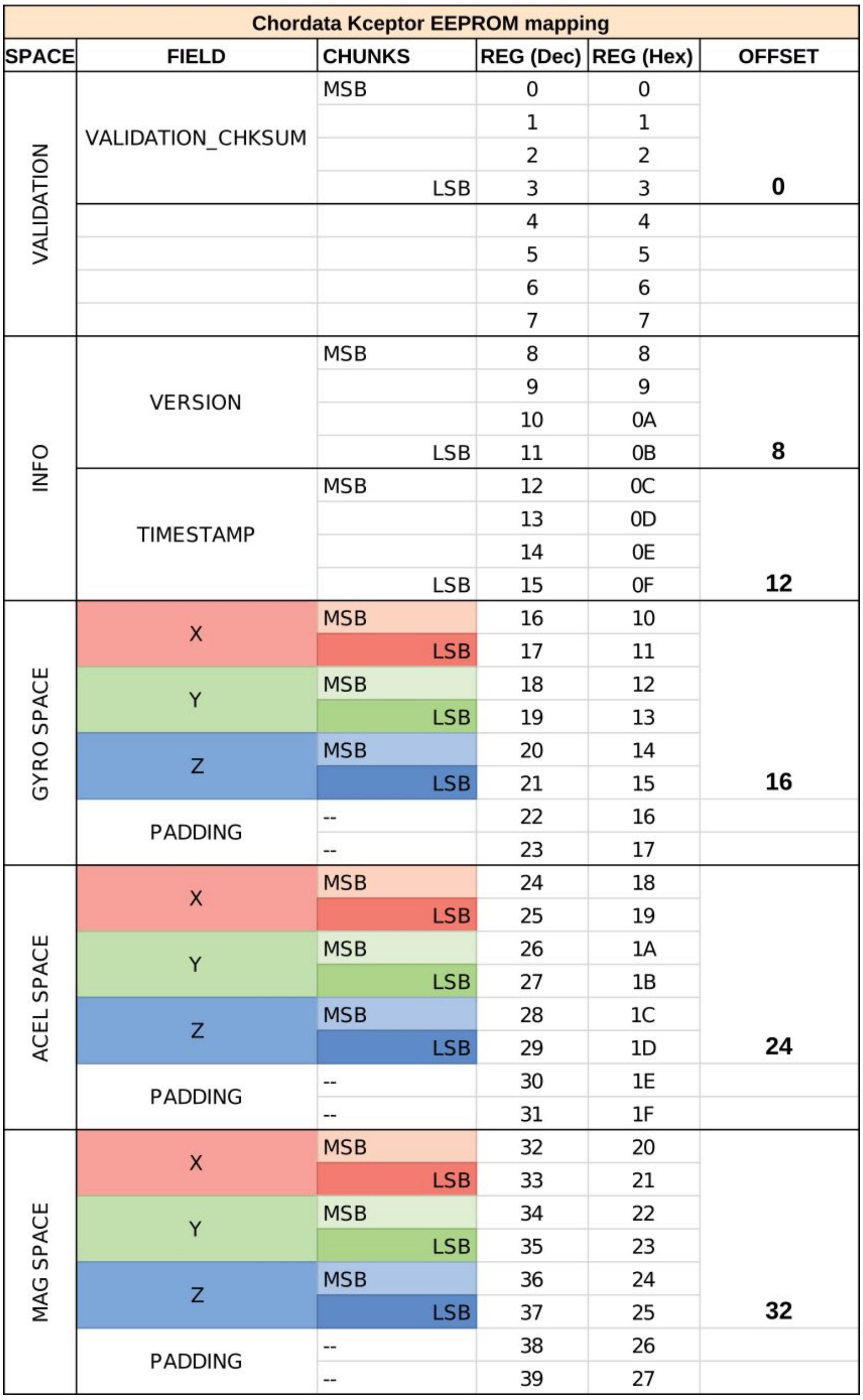
In this section you will find a description of most of the specific terms used in this guide, and in the chordata documentation in general. They are described in the way that they are intended inside the chordata system. For example you will find that the entry for “client” describes the program that receives and visualizes capture data, but doesn’t mention the broader concept of a client software that includes any type of computer program that accesses a service made available by a server.

TODO.

Armature
Armature parsing
Blender
chordata pose protocol
Client
Client side
Core parts
Custom sensing units
default byped configuration
GUI
hierarchy
host side
HTTP
Hub
i2c
Kceptor
Master (i2c)
Notochord
notochord control API
notochord control server
OSC
performer
Periferic parts
Python
Quaternion
SBC
scheduling
skeleton
UDP
Utility parts
Websocket

Appendix A: EEPROM table

The memory mapping table on the K-Ceptor EEPROM. The VALIDATION_CHKSUM is just a 32bits string resulting from the CRC32 digest of the string "chordata". It used for checking presence of data in EEPROM.



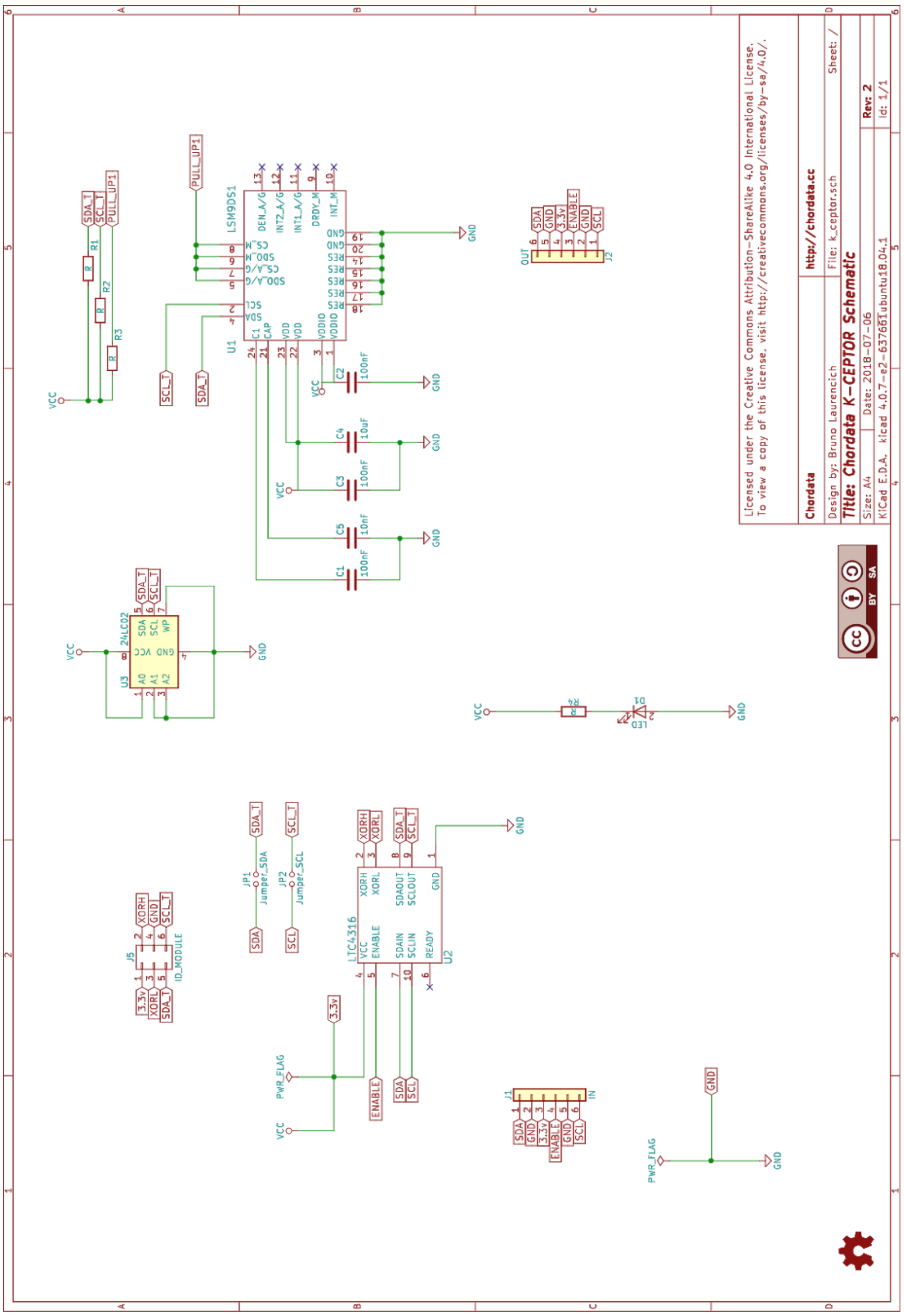
Chordata Kceptor EEPROM mapping

SPACE	FIELD	CHUNKS	REG (Dec)	REG (Hex)	OFFSET
VALIDATION	VALIDATION_CHKSUM	MSB	0	0	
			1	1	
			2	2	
		LSB	3	3	**0**
			4	4	
			5	5	
			6	6	
			7	7	
INFO	VERSION	MSB	8	8	
			9	9	
			10	0A	
		LSB	11	0B	**8**
	TIMESTAMP	MSB	12	0C	
			13	0D	
			14	0E	
		LSB	15	0F	**12**
GYRO SPACE	X	MSB	16	10	
		LSB	17	11	
	Y	MSB	18	12	
		LSB	19	13	
	Z	MSB	20	14	
		LSB	21	15	**16**
	PADDING	--	22	16	
		--	23	17	
ACEL SPACE	X	MSB	24	18	
		LSB	25	19	
	Y	MSB	26	1A	
		LSB	27	1B	
	Z	MSB	28	1C	
		LSB	29	1D	**24**
	PADDING	--	30	1E	
		--	31	1F	
MAG SPACE	X	MSB	32	20	
		LSB	33	21	
	Y	MSB	34	22	
		LSB	35	23	
	Z	MSB	36	24	
		LSB	37	25	**32**
	PADDING	--	38	26	
		--	39	27	

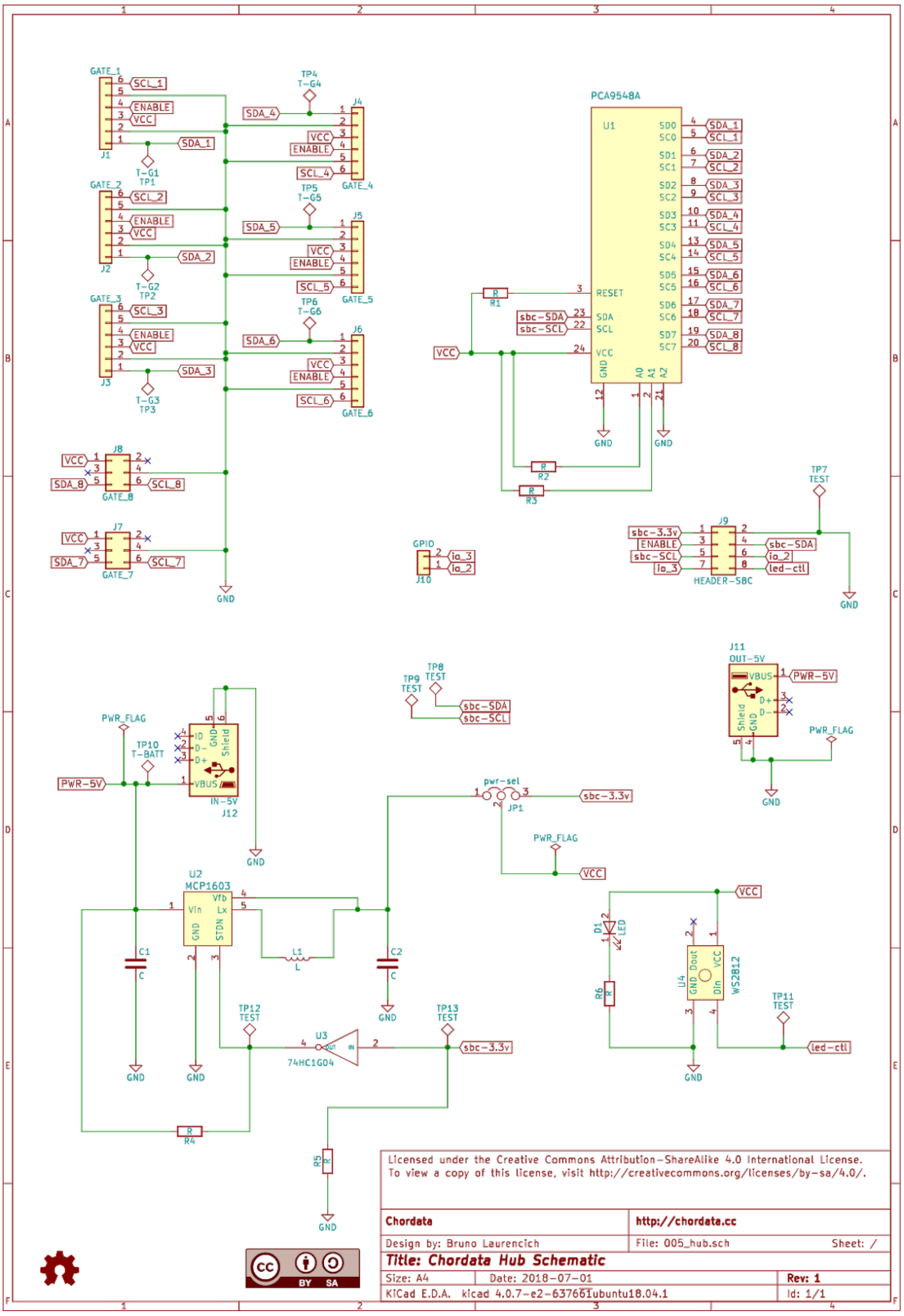
Appendix B: Control protocol commands and syntax

(TODO)

Appendix C: Schematics



Licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.
To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/.
Chordata
http://chordata.cc
Design by: Bruno Laurencich
Sheet: /
File: k_ceptor.sch
Title: Chordata K-CEPTOR Schematic
Size: A4
Date: 2018-07-06
Rev: 2
KiCad E.D.A. kicad 4.0.7-e2-6376~61~ubuntu18.04.1
Id: 1/1
LSM9DS1
U1
24LC02
U3
LTC4316
U2
J1
J2
J5
ID_MODULE
JP1
Jumper_SDA
JP2
Jumper_SCL
R1
R2
R3
R4
D1
LED
C1
C2
C3
C4
C5
100nF
10uF
10nF
VCC
GND
3.3v
SDA
SCL
SDA_T
SCL_T
PULL_UP1
XORH
XORL
ENABLE
PWR_FLAG
IN
OUT



Licensed under the Creative Commons Attribution-ShareAlike 4.0 International License.
To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/.

Chordata	http://chordata.cc	
Design by: Bruno Laurencich	File: 005_hub.sch	Sheet: /
Title: Chordata Hub Schematic		
Size: A4	Date: 2018-07-01	Rev: 1
KiCad E.D.A. kicad 4.0.7-e2-6376~61~ubuntu18.04.1		Id: 1/1