Mobile Robots

Universitat Politècnica de Catalunya

Santi Martínez

June 15, 2014

Index of contents

Images

Chapter 1

Preface

1.1 Introduction

The project I'm introducing here is named LEGO-Pi and the main goal aims to be able to build mobile robots controlling all the peripherals designed for and to LEGO from a Raspberry Pi, the populated credit-card sized board. Specifically, here, we will use the model b revision 2 of the board and, the supported peripherals, are the NXT series from LEGO and a good set of the Hitechnic peripherals too.

1.2 About Raspberry Pi and LEGO NXT

The idea of a Raspberry Pi as we know it nowadays became a reality in 2008, when ARM processors became more affordable. The main concern of the developers was to make a board capable to boot into a proper programming environment with multimedia support and thus try to make the board desirable to kids. In the other hand LEGO Mindstorms NXT project was released in 2006 and the main goal of it was to provide a set of sensors and a PBX capable to control it in a "straightforward" way via NXT-G programming software (however a variety of unofficial languages exists), making the design of simple robots available to kids.

With the above mentioned in mind, it's easy to see that both projects match pretty well, both are focused on kids, and both tries to make the technology more affordable to anyone interested and willing to learn, it seems the next step to develop an interface designed to merge both projects. There is a very good approach to this idea in the web, the BrickPi project, however one of the main goals of the LEGO-Pi project I'm introducing here, as I will expose in the next point of the document, is to develop a low cost platform, affordable and implementable for anyone, thus making the obscure world of electronics a little bit more accessible too.

1.3 Scope and objectives

The LEGO-Pi project is motivated for the "necessity", or the desire of implementing, in top of a free-form platform, an interface able to control all the LEGO-NXT peripherals. The advantages that the project will bring, specially to education are obvious:

- Have a free operating system, in top of what I'll build the software library that will take care of the interaction with the peripherals.
- Have an open source C ANSI library making the code available to anyone interested.
- Have a better comprehension of what peripherals do and how they do it.

Under the writer point of view the above mentioned justifies enough the implementation of the project, nevertheless, it's important to mention the economical part, as one of the main goals of the project is to provide a low cost interface to interact with the LEGO peripherals, so minimize the hardware and simplify the design will be a must at every point of the implementation, thus making the project available to anyone interested. Besides the above mentioned while LEGO updates his devices year after year re-profit the old peripherals can be interesting, so implementing a free low cost interface able to control the LEGO peripherals, or at least set a base point for it seems a good idea, and Raspberry Pi seems a good choice to act as the PBX of the project.

1.4 Organization of the project

We start going deeper here, LEGO-NXT peripherals can be distributed into three big blocks, motors, analog devices and digital devices, so the software library and this document will be organized following this pattern. From the whole set of peripherals of the NXT series the university provides me with a few ones that I need to give support to, these are the following devices:

Motors:

- Standard NXT DC motor. (Servo motors not supported).

Analog Devices:

- Light sensor.
- Push sensor.
- Sound sensor.
- Hitechnic Gyroscope sensor.

Digital Devices:

- Ultrasonic Sensor.
- Hitechnic Acceleration Tilt Sensor.
- Hitechnic Color Sensor.
- Hitechnic Compass Sensor.
- Hitechnic IRSeeker Sensor.

So these are the devices supported for the LEGO-Pi library, however, generic functions have been implemented too to give support to other LEGO design compliant devices. Further on, in the testing chapter, we will discuss on the versions of the sensors that have been tested, and what versions remains untested but supported.

From this point I'll be referring to different parts of the library, you can find the the project at:

https://github.com/szz-dvl/LEGO-Pi

From now on we'll go deeper into the implementation, I'll carry on explaining how the three main blocks of the project has been implemented, the motor interface, the analog interface and the digital interface. All the interfaces will be explained as if it is only one port of each one, this is, we will dimension each interface for just one of the devices that are meant to control, later on we will put it all together and finally dimension the whole project.

Chapter 2

Implementation

2.1 The first approach

The first thing we need to do here is to take a look to what we want to use as the new PBX for the NXT series peripherals, it is, the Raspberry Pi. Concretely we will use here the model B revision 2 of the board that will provide us with 21 GPIO (General Purpose Input Output) pins, including the P5 header. All the signals to control the behavior of the LEGO peripherals are generated by the Raspberry Pi GPIO pins, so USB and ethernet ports remains free for the user benefit. Nevertheless GPIO pins capabilities have certain restrictions:

- It can not deliver more than 50 mA altogether.
- It can not deliver more than 3.3V on each pin.
- It aren't 5V tolerant.
- All GPIO pins are digital.

And LEGO peripherals use to have a working voltage of 5V, so lets see how we solve all this issues and finally control the LEGO peripherals successfully.

2.2 The motor interface

The motor interface is the piece of the project that will take care the motor control, it gathers the hardware design and the software layer, developed to work with this hardware.

2.2.1 The motor hardware interface

The LEGO NXT motors are DC motors, what means that the more voltage you provide to the motor the faster it'll turn, that is, the velocity is proportional to the voltage applied to the pins responsible of the motion, being able to handle up to 9V. Well, the experienced Raspberry Pi user will realize the first problem we are facing here, as mentioned in the section 2.1 our Raspberry Pi is only able to deliver up to 3.3V on each GPIO pin (and about 16mA of current), so we need some hardware able to handle our 3.3V signals and "convert" it to 9V (with enough current capacity) to manage the motor properly, an H-bridge is what we are looking for. An H-bridge is an integrated device consisting in two NPN transistors and two PNP transistors,

2.2. The motor interface

NPN and PNP transistors are very commonly used in electronics to implement switches, or logical gates, in our case the transistors will be arranged this way:

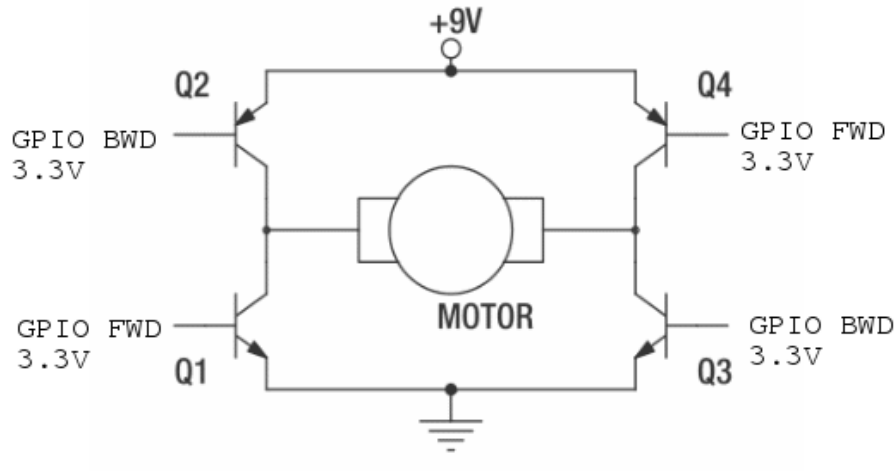


Figure 2.1: H-bridge, stationary motor

So, when we have to set the motor in forward motion, we just need to set the GPIO pin tagged as `GPIO FWD` in high state, then delivering 3.3V to the base of the transistors Q1 and Q4, and the GPIO pin tagged as `GPIO BWD` in low state, letting the current flow this way:

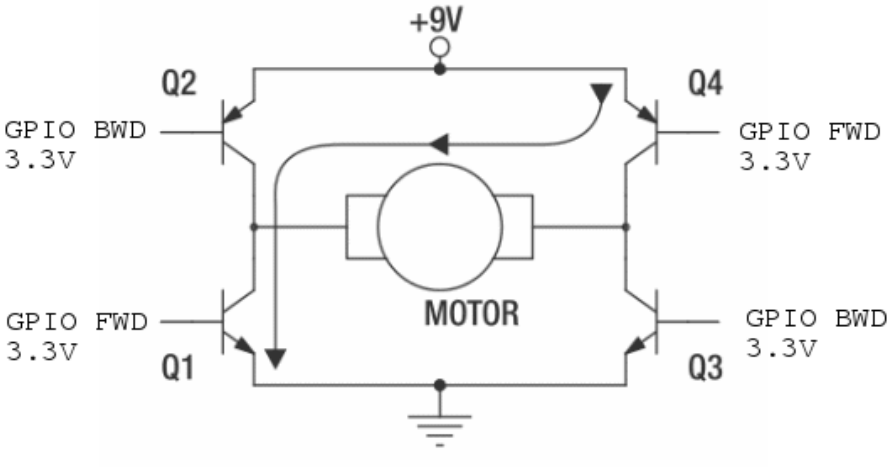


Figure 2.2: H-bridge, forward motion

Then if we have to set the motor in backward motion we just need to set the GPIO tagged as `GPIO BWD` in high state, delivering 3.3V to the base of the transistors Q2 and Q3, and the GPIO pin tagged as `GPIO FWD` in low state, letting the current flow this way:

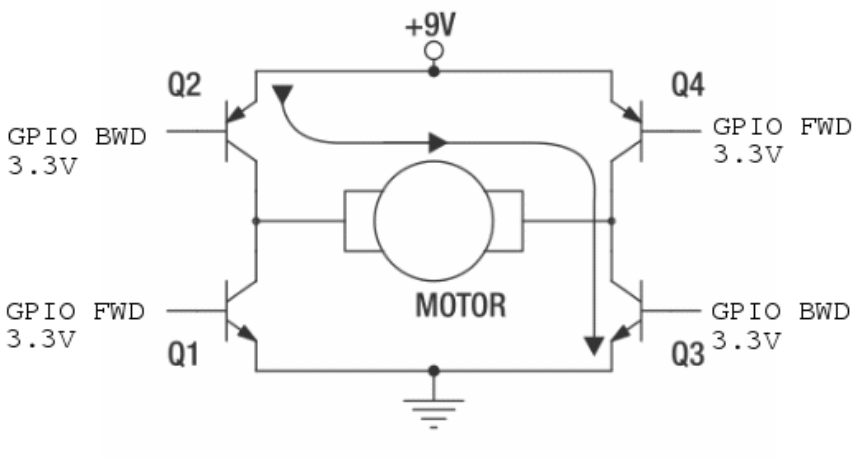


Figure 2.3: H-bridge, backward motion

If we need to stop the motor, we just set both GPIOS connected to the H-Bridge in low state so the current stops flowing thus stopping the motor, as you can see in the figure 2.1.

Fortunately there are several integrated H-Bridges in the market, so we don't need to implement this "manually", the one used for the LEGO-Pi project is SN754410NE of Texas Instruments, which allow us to control up to two motors, the one the NXT PBX use is LB1836M of ON semiconductors, however this last one supports a little less current, so I chose SN754410NE for this reason. Analogously to the transistors this integrated circuits use to build four diodes attached to the motor inputs and from the emitter of the transistors and from GND to motor input to protect the controller, the Raspberry Pi in our case, from the back electromotive forces (back emf) generated by the motor.

At this point the hardware required to handle the motor motion is solved, nevertheless, LEGO NXT motors provides an encoder which will give us feedback of the motor position, so it'll be nice to profit this advantage to develop a curious library.

How LEGO NXT motor encoder works

LEGO NXT motor encoder is an optical encoder, this means it is implemented with a led emitting diode and a photo sensor, but how the quadrature signal is generated?. The encoder itself can be viewed as a gear attached to the motor axis, in our case from the motor to the encoder we have a gear reduction of 10:32, and the motor have a gear train with a 1:48 ratio, so if our encoder gear have 12 slits on it as in the image 2.4:

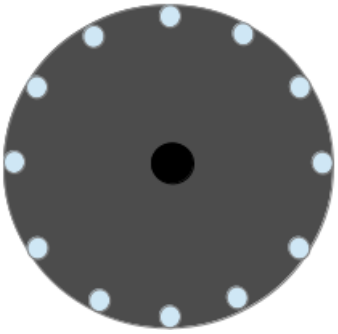


Figure 2.4: Encoder gear front view

For one turn of the output hub the encoder will do $48 \times 10/32 = 15$ turns, for each turn optical detectors will see 12 slits, then we got that for each turn of the output hub the encoder will see $12 \times 15 = 180$ slits, if we use both sides of the encoder gear as you can see (with a little bit of imagination) in the image 2.5:

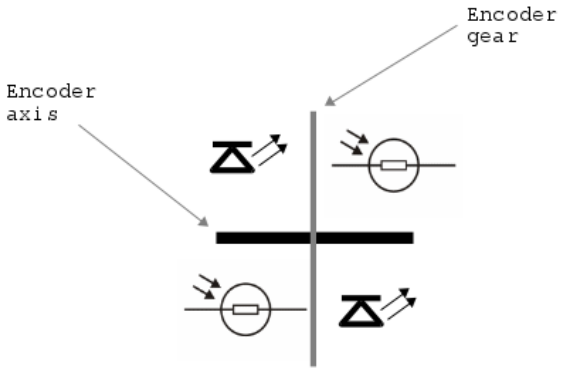


Figure 2.5: Encoder gear lateral view

We have then, $180 \times 2 = 360$ ticks of resolution for every turn of the output hub, since we have a quadrature encoder the maximum resolution will be of 720 ticks per turn, this will be discussed later on in the section 2.2.2.

The graphics above aren't bad to have an idea of how an optical encoder works, however, to have a better comprehension of how it looks a real assembly of the encoder in the NXT motor I let you here a picture of one motor disassembled.

2.2. The motor interface

Figure 2.6: Encoder disassembled

LEGO NXT motor encoder, however, have a working voltage of 5V, then it will deliver a quadrature signal ranging from 0V to 5V and our Raspberry Pi is not 5V tolerant, so this time we need to drop some voltage before we plug the encoder outputs to the board, This can be done with a couple of resistors an a capacitor, the capacitor is added to the circuit to avoid noise in the signals and bad readings. If we decide to use a 4K7 resistor plugged to the encoder output, applying Kirchhoff laws we can find the value of the resistors and capacitors needed to drop enough voltage to make the signal safe for the board, these are:

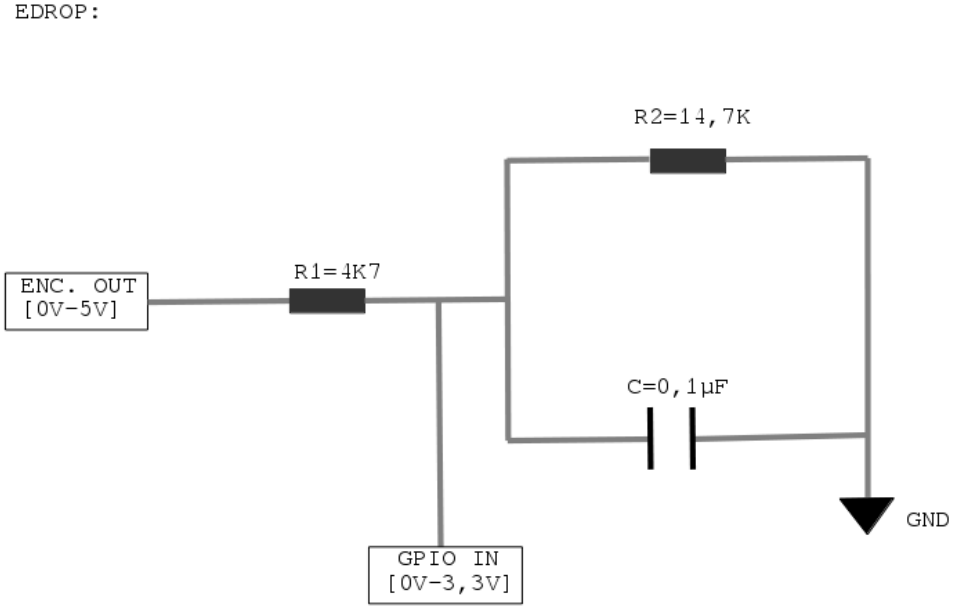


Figure 2.7: EDROP circuit

Motor port final hardware assembly

So now we have the hardware needed to control motors ready, the final assembly to connect one NXT motor to the Raspberry Pi will be something like:

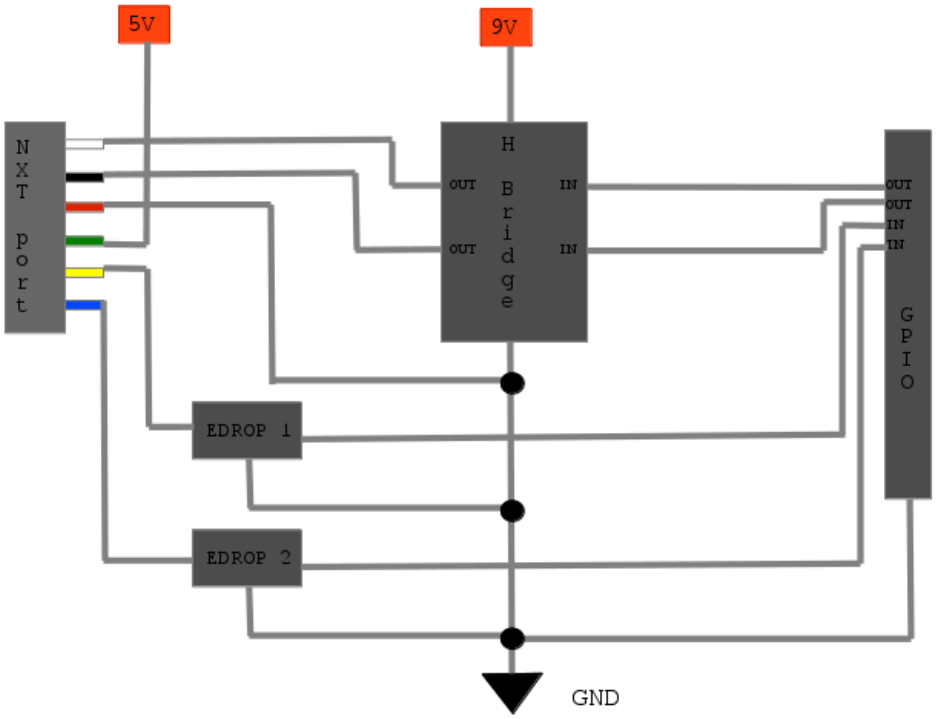


Figure 2.8: Motor final assembly

There are a few more connections to do in the H-Bridge, but we don't care here, this is just a very simple schematic of the block, showing how the things will be arranged, the black circles means junctions, so if no black circle present, wires are crossing. We don't care about the GPIO numbering neither, by the moment, the only important question here is that we need four GPIO pins to control one motor. If you want you can check out the schematics of the project in the appendix A of this document (Project Schematics), or in the `docs/schematics` directory of the LEGO-Pi git-hub previously provided.

2.2.2 The motor software interface

So now we have a nice hardware interface capable to control NXT motors, now is time to specify how we want the motors to behave, the piece of the library that will take care of this is named `lego_motor.c`, you can find the file in the LEGO-Pi git-hub previously provided. The experienced reader will notice here that, with the hardware above mentioned and the raspberry GPIO pins capabilities, we are only able to move the motor forwards and backwards, but there is no way to control the velocity we are moving the motor (we always deliver all the voltage provided to the H-Bridge to the motor), so we need a mechanism to control the motor velocity in both directions, as we are dealing with DC motors it means that we need a mechanism to able to control the voltage that motors will receive. This can be achieved by means of PWM (Pulse

Width Modulation), however our Raspberry Pi have only one GPIO pin (GPIO 18) capable to generate PWM via hardware, this means that we will only be able to control the motor velocity in one of the directions, and only for one motor, so we need to find a way to generate the pulses via software, since we want to be able to control the velocity in both directions.

The LEGO-Pi library make use of the Wiring-Pi library by Gordon Henderson, this is a low level library that will come in handy to control the GPIO pins of our Raspberry Pi, and it provides a way to generate a pulse via software in whatever GPIO pin, however, the way Wiring-Pi generates the pulses will result "expensive" in computational terms, because it is managed via POSIX threads that will update the GPIO pin state depending on the pulse we demand (within a sub cycle time). There is another library named RPIO by Chris Hager, this is a python library, however the developer made the low level code of the library public (which is always the best choice). The particularity of RPIO is that manages the pulses via DMA (Direct Memory Access) channels, this is a very clever and polite way of generate pulses. We have 16 DMA channels available in our Raspberry Pi, RPIO use 15 of them (from 0 to 14), every DMA channel is able to manage multiple GPIO pins, so we just need to configure the pulse we want to generate into a memory control block that will be loaded into the internal registers of the desired DMA channel and the kernel will take care of the pulse generation, consuming almost nothing CPU power.

With the above mentioned in mind I created a patch making available the pulse generation via DMA channels to Wiring-Pi library, the integration with the Wiring-Pi library is poor and only made to fit my needs, you can find the patch in the `install/patch` directory of the provided LEGO-Pi git-hub . If you want more information about DMA channels you can refer to the page 38 of the Raspberry Pi data sheet, you can find it here, Wiring-Pi C sources can be found here, and the files used to create the patch from RPIO can be found here. We will see now what exactly means PWM and how it works.

How PWM works

As explained above, in the section 2.2.2 PWM stands for Pulse Width Modulation and it must help us to control the velocity of the motors somehow, but how exactly PWM works?, how it interacts with the hardware we use?, I will try to solve this questions in the current paragraph.

As mentioned before in the section 2.2.1 we use an H-Bridge to provide the power to the motor, then the GPIO pins will take care of opening or closing the "gates" of the H-Bridge letting the current flow in one way or another depending on the direction we want to move the motor, so without PWM if, for an instance, we want to move forward we have something like this:

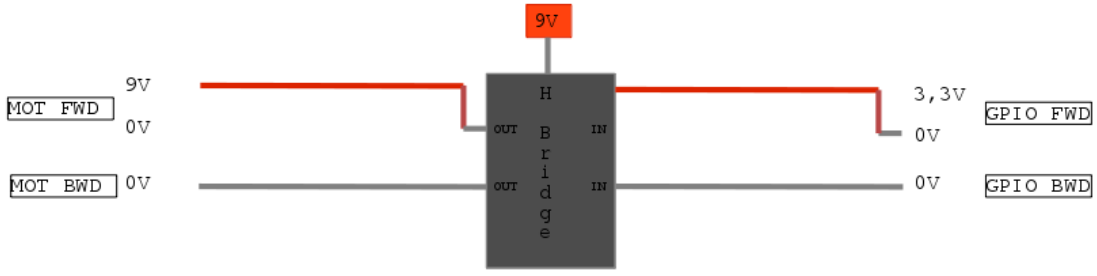


Figure 2.9: Motion without PWM

If we set the `GPIO FWD` in high state the motor will effectively move forward, however as we can see in the image 2.9 all the power provided to the H-Bridge will be delivered to the motor thus making the motor move at maximum speed (if the voltage provided to the H-Bridge is the maximum supported by the motor), or, if we set the `GPIO FWD` in low state, no power will be delivered to the motor thus making the motor remain stationary. Now, how PWM gonna help us?. As the name says what we do is to modulate the width of a generated pulse, so if, as an example, we want to deliver to the motor the 80% of the power we provide to the H-Bridge we generate a pulse like this in the desired GPIO pin:

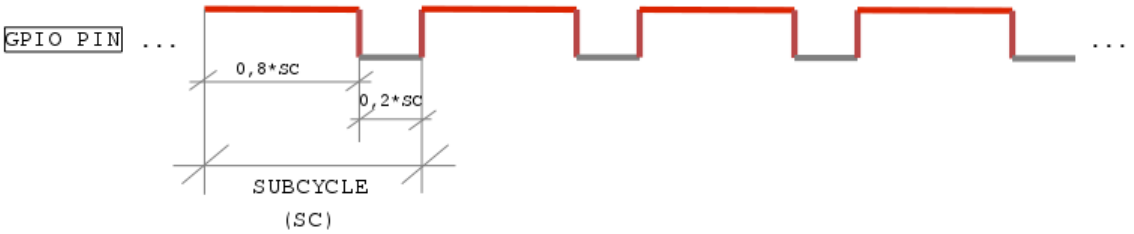


Figure 2.10: PWM, subcycle time

Here I'm introducing a new concept, the sub cycle time. What we gonna set is a sub cycle time, and we gonna split this time between high state and low state proportionally to the power we want to deliver to the motor for the desired GPIO pin, then this sub cycle gonna be repeated indefinitely till we set the GPIO back to low state. So the result is that, if the sub cycle time is small enough,it is, if the frequency of the pulse is big enough, we gonna open and close the H-Bridge "gates" very fast, thus delivering to the motor a proportion of the total voltage provided to the H-Bridge, actually what we do is deliver all the voltage provided to the H-Bridge for a portion of the time depending on the sub cycle that we set, but the motor will behave as if a portion of the voltage provided to the H-Bridge (proportional to the sub cycle high time) is being delivered to it. So following with the example above:

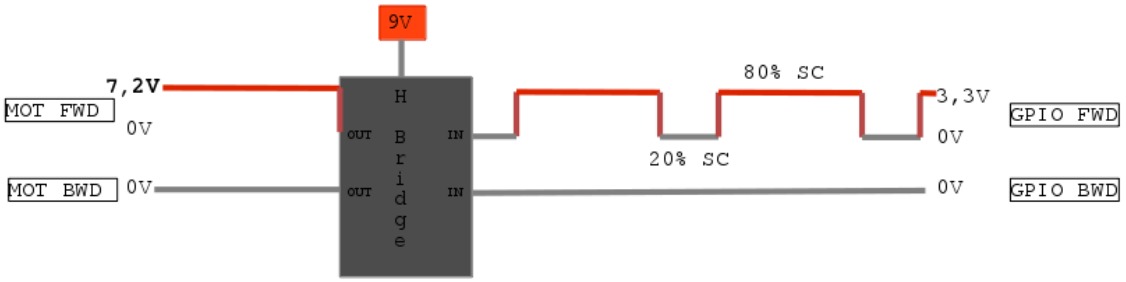


Figure 2.11: PWM, 80% power example

Here we have, if we set a pulse with 80% of the time in high state, the motor will receive the 80% of the total power provided to the H-Bridge, as we are dealing with DC motors, the velocity of the motor is proportional to the voltage applied to it, then we can control the velocity of the

motor controlling the proportion of time in high state within a sub cycle. Then generalizing we have:

$$VELOCITY \propto VOLTAGE\_APPLIED$$

and,

$$VOLTAGE\_APPLIED = VH\_BRIDGE \times \left(\frac{SC\_HIGH\_STATE\_TIME}{SC\_TOTAL\_TIME} \right) ,$$

assuming:

$$VH\_BRIDGE = MAX\_VOLTAGE\_SUPPORTED\_FOR\_MOTORS$$

which is the best scenario, we have :

$MAX\_VELOCITY$ is achieved when voltage provided is VH_BRIDGE ,

then we can say that

$MAX\_VELOCITY$ is achieved when
$SC\_HIGH\_STATE\_TIME = SC\_TOTAL\_TIME$

It means when the GPIO is in high state always, what we had before applying any PWM management, so everything squares pretty well, the generalized graphic (for forward motion) will look something like:

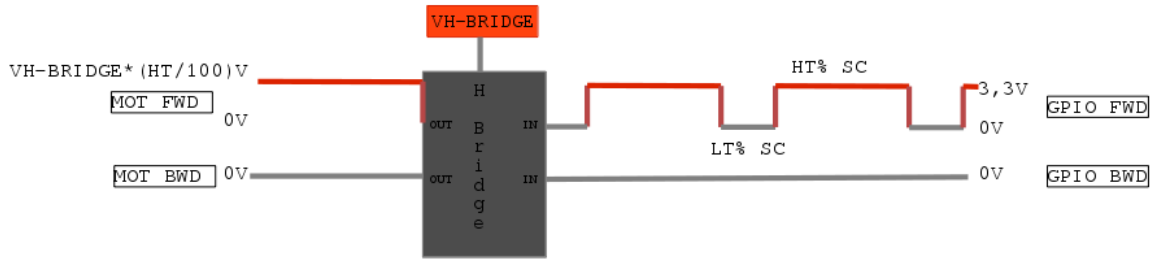


Figure 2.12: PWM, generalized forward motion

Where HT and LT are percentages of the high state time and low state time respectively. Analogously, switching the lines `GPIO FWD` and `GPIO BWD`, and the outputs of the H-Bridge you can get the generalized graphic for backward motion. If you are interested in the NXT motor statistics, you can find a very complete survey here.

Encoder management

At this point we have the hardware necessary for motor controlling and the software arranged to control the velocity too, as mentioned a couple of paragraphs above in the section 2.2.1 LEGO NXT motors provides an encoder that will give us a quadrature signal in the blue an yellow pins of the NXT port, NXT uses both signals to determine the direction of the motor, however we can take advantage here of our Raspberry Pi hence it boots into a proper programming environment, what means that LEGO-Pi have notion of the direction a motor is turning without

the need for the quadrature signal, this means that LEGO-Pi strictly needs just one encoder signal to be able to provide feedback of the motor position fairly enough. This is how the signal will look like for a two degrees forward rotation:

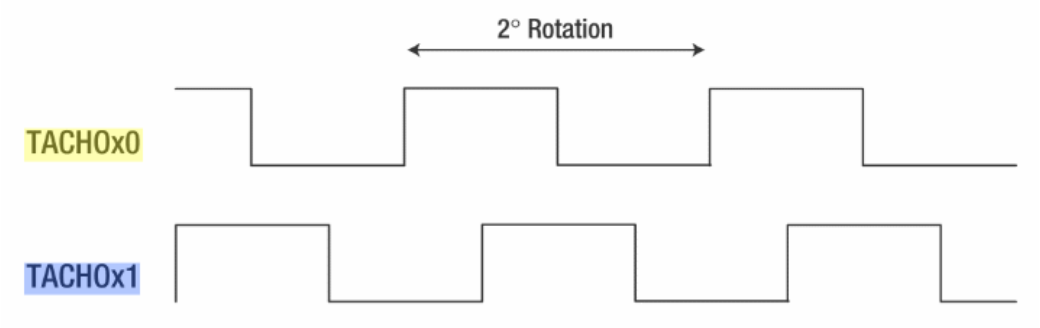


Figure 2.13: Encoder, two degrees forward rotation

As we can see in the figure 2.13 the signals in the blue and yellow wires are shifted, this is how the NXT PBX determines the direction of the motor, the frequency of the signal will determine the velocity at which we are turning, one cycle of the wave corresponds to one degree of rotation, as we can see the yellow wire is used by NXT PBX to detect forward motion, a backward rotation of two degrees will look like:

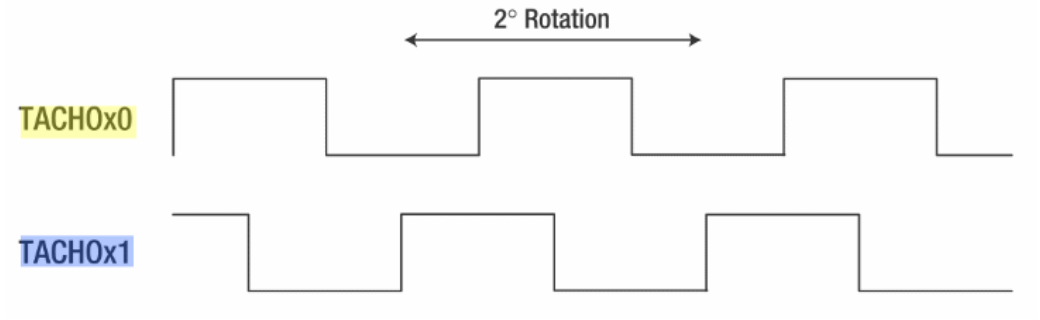


Figure 2.14: Encoder, two degrees backward rotation

Here we see how the blue wire is used by NXT PBX to detect backward rotation, the shift between the two signals is a quarter of the phase, being the phase the fraction of the wave cycle elapsed from the origin, since we are connecting both encoder outputs to the Raspberry Pi yet we have notion of the direction of the motor, we can use the extra signal to have a double precision encoder, what means that we can count the wave cycles on both lines getting then 720 ticks per turn of the motor, since the shift between the signals is stable the readings will be accurate, to be fair we will get a tick more for every reading we do, however it usually means to stop the motor one tick before the desired set point is achieved, which helps to reduce the little error we have when we stop delivering power to the motor, so it is a pretty good solution to me.

Besides of the above mentioned there is another good reason to have both lines connected to the Raspberry Pi. The encoder signals are treated by means of interrupts provided by Wiring-Pi

library, this come in handy for the encoder signals because we can configure an ISR (Interrupt Service Routine) that will be executed every time a tick comes, both encoders are configurable by the user so it is a good way to automatize some procedures, leading to very funny programming designs, so for example we can configure an encoder signal to store the ticks spent while the other take readings from several sensors for every turn of the wheel, actually having the capability to control NXT peripherals from a native C library the possibilities are almost endless. If you want more details of how to use this capabilities at this point you can refer to the appendix B of this document (Library Specification) in its section B.1.3 ,you must look for the `mt_reconf` function.

At this point we are able to control a motor velocity and receive feedback from the encoder too, so we have a close loop, what means that we can set a motor velocity and check if the feedback received from the encoders is the expected, to implement this a P.I.D (Proportional-Integral-Derivative) controller "approach" has been implemented.

How my P.I.D approach works

As you may have realized what I've done is some kind of P.I.D emulation, and I say emulation because the common behavior of a P.I.D controller (applied to motors) is to act at every tick received from the motor, then reset the pulse width if the velocity is not the adequate,it is, if the time between ticks is not the expected, however, this is a very hard to handle for a our Raspberry Pi, so I came up with a version of the P.I.D algorithm you can check out here. What I do is to store the information for a configurable amount of ticks, then apply the P.I.D algorithm to the mean of the times between ticks, so the pulse reconfiguration will only happens if we found an error in the configured amount of ticks, so actually, what I do is make the Proportional part of the algorithm work with the average of data of the last amount of ticks (so it will have a part of an Integral implicit too ...), the Integral will be the accumulated of all the averages (so actually is an "squared" Integral, or something like it ...), and the Derivative will be the difference between the two last averages, then we get a notion of error, the amount of ticks needs to be configured accurately, the faster we turn the bigger must be the amount of ticks.

In top of that the pulses generated when an error is found aren't fully dynamically calculated, what I do is to get a range of the initial pulse (the velocity we actually want) and if a positive error occurs I just reset the pulse to the inferior range (since we are moving a little faster than expected), if a negative error occurs I reset the pulse to the superior range (since we need to move a little faster to recover the lost ticks), independently of the module of the error. This can be viewed as we only have three states when P.I.D is acting, in the correct velocity (P.I.D won't reset the pulse), recovering velocity (P.I.D will set a wider pulse, so we will turn faster) or decreasing velocity (P.I.D will set a narrower pulse, so we will turn slower).

When an error occurs is another matter, a function `mt_calibrate` (B.1.3) is provided in the library, it is in charge of calibrating the motors to have a proper data set to work with P.I.D control, what the function does actually is turn the motors at different velocities (it is, applying different pulses), and store the times between ticks for each velocity, then get the average for the stored sample and get a, hopefully, reliable time between ticks for each velocity, the absolute deviation of the sample is computed too, so we've got a notion of physical error indeed, the amount of velocities for what `mt_calibrate` will compute the expected times and errors is configurable, within a boundaries, the more velocities we compute the more precise will be my P.I.D control, this will be treated thoroughly in the testing section 4.1. So now we have a set of times we expect

between ticks linked to his pulses, and notion of physical error for each pulse too, what my P.I.D algorithm will do is to interpolate the data set to get the expected values of time between ticks and physical error of the demanded velocity, then, if the absolute value of the error computed by the P.I.D algorithm is greater than the interpolated physical error for the demanded velocity an error arise, and it will be treated as explained in the above paragraph.

As you might notice the main problem here is that the times that we get between ticks are very unstable, specially when we've got more than one motor, (because we need more threads to control it), so we need at least a few ticks to get a reliable error value, this is the main reason for the averages "solution", it is probably not the best, however is the better I've been able to do with a single ARM processor board, hence to LEGO-Pi, P.I.D control is considered an experimental functionality. Later on, in the section 3.4 we will discuss on a possible smarter solution to this problem.

2.3 The analog interface

The analog interface is the piece of the project that will take care of the interaction with analog sensors, it gathers the hardware design and the software layer, developed to work with this hardware.

2.3.1 The analog hardware interface

LEGO NXT analog sensors are designed under the same pattern, so do Hitechnic, this means that, fortunately, the connections of the sensors are "standardized", then we can develop a hardware, with several ports available, capable to control whatever of the supported sensors in whatever of the ports we want to plug it. First of all we gonna see which are the standardized wires, and which of them will have a different behavior for each sensor, so the standardized wires for an analog sensor are:

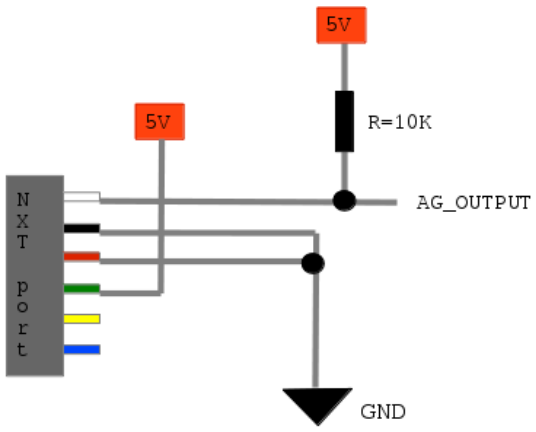


Figure 2.15: Analog sensor standard wires

So, as you can see in the image we always gonna get the analog readings from the white wire, the pull-up resistor is added to widen the range of the analog output from the sensors, so we

can get more precise readings. Red and black are plugged to ground and the green wire will provide voltage to the sensors, so what happens with blue and yellow wires?, each sensor use this wires in a different manner, from the list of supported analog sensors you have from the section 1.4 this are the behaviors of the pins:

Light sensor:

- *Yellow:* Used to deliver voltage to the led that provides extra light.
- *Blue:* Not used.

Push sensor:

- *Yellow:* Not used.
- *Blue:* Not used.

Hitechnic gyroscope sensor:

- *Yellow:* Not used.
- *Blue:* Not used.

Sound sensor:

Here I need to explain a little bit more, for the sound sensor when we deliver 3.3V to the yellow pin and 0V to the blue pin we set the sensor in dB (decibels) mode, thus the analog reading of the white wire must be interpreted as a dB reading, however, if we deliver 3.3V to the blue pin and 0V to the yellow pin we set the sensor in dBA (adjusted decibels) mode, then the readings of the white wire must be interpreted as dBA readings, nevertheless, due to the lack of pins in our Raspberry Pi, LEGO-Pi only allows sound sensor to work in dB mode, so under LEGO-Pi's point of view:

- *Yellow:* Used to set the sensor in dB mode.
- *Blue:* Not used.

This decision is not aleatory, since light sensor needs 3.3V in the yellow pin to power the led, the only way to support any analog sensor in any analog port is to wire up the yellow pin to a GPIO pin and plug the blue pin to ground. This will be clarified later on in the section 3.1.

Another thing to take into account here is the current we provide to the yellow pin, the GPIO pins are able to deliver 50mA of current among all of them, or a maximum of 16mA on each pin not exceeding the 50mA restriction, the sound sensor won't be a problem for this reason, however, a led can consume up to 12mA-16mA easily, in the light sensor the led is connected to the collector of an NPN transistor and the 5V bus trough a 180 ohms resistor, however the resistor connected to the base of the transistor is very big, and after some testing I find out that the current delivered for the Raspberry Pi GPIO pins wasn't enough to power the led (with all the assembly operative). The solution here is to plug a pull-up resistor from the 3.3V bus to the yellow pin and the GPIO pin in charge of controlling the led, I found out that a 4K7 ohms resistor will suffice here. If you want more info about pull-up resistors you can refer to this Wikipedia article. So now we have a our analog port solved, it'll look something like this:

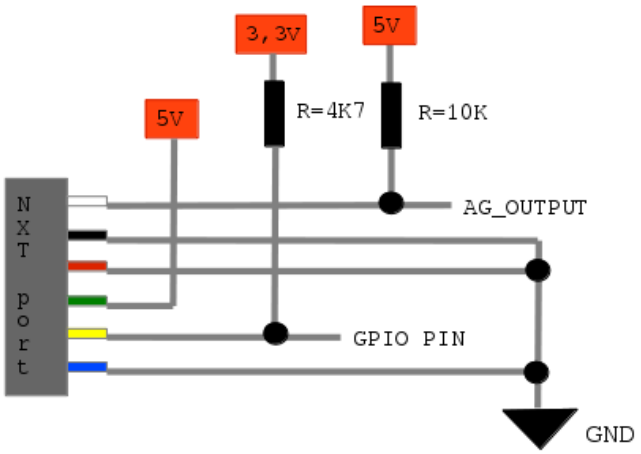


Figure 2.16: Analog sensor standard port

One more time the black circles means junctions, so if no black circle present wires are crossing. Now we have our analog "generic" port ready to use, however the experienced Raspberry Pi user will notice our next problem to solve here, our Raspberry Pi have no analog pins available in the GPIO pins so we need to find a way to get the readings from the analog sensors into our Raspberry Pi, this is solved by means of an analog-digital converter.

The analog-digital converter

An analog-digital converter, A/D further on, is an integrated device able to convert an analog voltage into a digital value, the precision of the A/D will vary depending on the device we choose, for the LEGO-Pi project the device used is MCP3204 from Microchip. This device will provide us 12 bits of precision and four analog channels to wire up the sensor ports, and what is more important, an SPI (Serial Peripheral Interface) interface to communicate with it, this will come in handy with our Raspberry Pi since it supports SPI trough the GPIO 10 (pin 19), 9 (pin 21), 11 (pin 23) , 7 (pin 26) and 8 (pin 24), of the last two pins we will need only one of them, it is the CS (chip select) pin, every SPI device connected to the bus needs its own CS line, as we only need one A/D, and no more SPI devices will be present in the project GPIO 7 will remain free. Another important point to remark here is that the SPI communications with the A/D can be done at 3.3V volts, which are good news because we don't need to worry about the working voltage of the sensors (which is 5V), we just set the reference voltage of the A/D to 5V and read the values from the A/D through our Raspberry Pi SPI interface.

Analog port final hardware assembly

With the above mentioned in mind the schematic to plug one analog sensor to our Raspberry Pi will look something like:

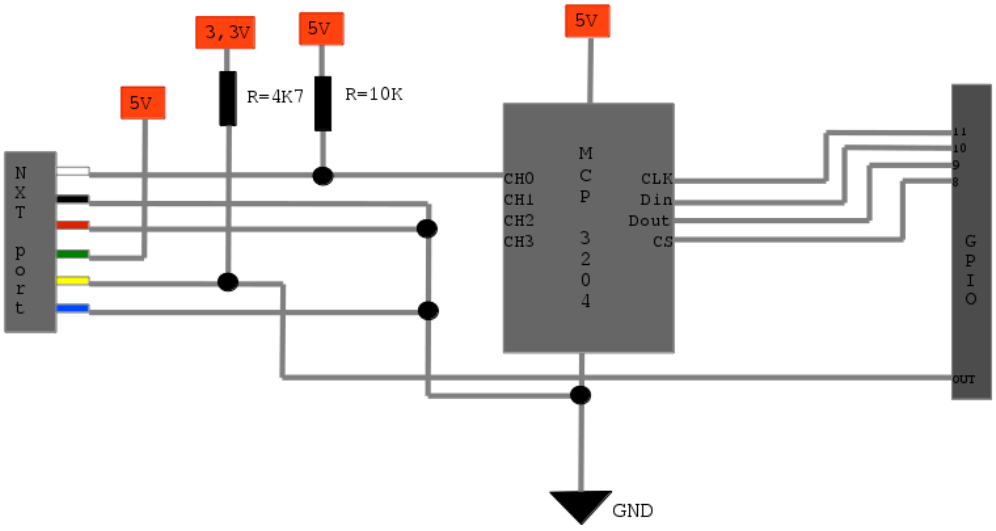


Figure 2.17: Analog port final assembly

Again black circles means junctions, no black circle means wires are crossing. There is a few more connections we need to do in the A/D, but we don't care here, this is just a simple schematic showing how the things must be connected, the important thing to keep in mind here is that to control analog devices we need the GPIOs 8, 9, 10 and 11 to communicate with the A/D and one GPIO more for every analog port we want to give support to (with a maximum of four ports, due to the A/D number of channels). If you want more info about the analog hardware interface you can refer to the appendix A of this document (Project Schematics), or to the `docs/schematics` directory of the LEGO-Pi git-hub previously provided.

2.3.2 The analog software interface

The piece of the library that will take care of the analog sensors control is named `lego_analog.c` you can find it in the provided LEGO-Pi git-hub . There is not much peculiarities to explain about the software interface of the analog sensors, what LEGO-Pi does actually is an SPI transaction with the A/D converter asking for voltage of the desired channel of the A/D, then the 12 bits result will be interpreted in a different manner depending on the sensor attached to the A/D port. If you want more information about low level SPI transactions you can refer to this article on Wikipedia.

As explained above we developed the hardware interface in a way that any supported sensor can be attached to whatever of the available ports, so the port and sensor types will be defined dynamically. When the user request the value from LEGO-Pi he can use any of the specific functions designed for each of the sensors or two of the generic functions provided by the library, this generic functions will return the raw value taken from the A/D, one of them will return the voltage as a floating point value, ranging from 0v to 5V, and the other one will return an integer ranging from 0 to 4095 (the max value since we are getting 12 bits of precision), this functions has been implemented to support other LEGO "compliant" analog devices, it means, that follows the standard pin-out of LEGO devices, the user will be able to control the yellow

wire behavior, and the blue pin won't be accessible. For the supported devices there is a set of functions implemented that will provide more precise info for the desired sensor type, if you want more information about this functions you can refer to the appendix B of this document (Library Specification).

2.4 The digital interface

The digital interface is the piece of the project that will take care of the interaction with digital sensors, it gathers the hardware design and the software layer, developed to work with this hardware.

2.4.1 The digital hardware interface

As happens with analog sensors, LEGO digital sensors are designed under the same pattern, so do Hitechnic, this means that again the pin-out of the devices are standardized, so we can develop an standard port able to handle whatever of the supported sensors, this time, however, all the pins are standardized so we can develop our standard port with the following connections:

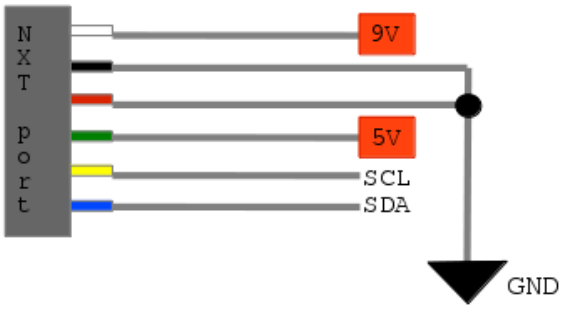


Figure 2.18: Digital standard port

If you are familiarized with I2C protocol the yellow and blue lines tags would be familiar to you too, it are the `SCL` (I2C clock) and `SDA` (I2C data) lines and are used by the sensors to communicate with the master device that controls them, so LEGO sensors use I2C protocol, or, at least, a version of it, this will be clarified later on in the 2.4.2 section. What we need to know now is that there are I2C devices that works with a voltage of 5V, there is another ones that have a working voltage of 3.3V, concretely, in our case we have that Hitechnic devices have a working voltage of 3.3V, however, the ultrasonic sensor works with a voltage of 5V, we know from previous sections that our Raspberry Pi is not 5V tolerant, so we need to find out a solution to this issue. With the I2C protocol the clock line is always driven by the master device, it is the Raspberry Pi in our case, so the pin will always be configured in output mode, since none of the digital sensors supported stretch the clock line, and there is no support for clock stretching (this will be discussed thoroughly in the next section too), in this case we have no problems with the voltage since the ultrasonic sensor have internal pull-ups to 5V that will raise up the 3.3V signals generated by our Raspberry Pi. Nevertheless we need to protect the GPIO pins connected to the `SCL` lines of the digital ports because in the boot process of our Raspberry Pi all the pins are configured in input mode (if the Raspberry is booting and a sensor

is attached to a port 5V will come into the board), and we have to do it without disturbing the I2C communication, so in this case with a big pull-down resistor it is all done. Nevertheless with the SDA lines the workaround is different, SDA lines are driven by the master when we are sending data to the sensor, usually to require data from it, or to configure the sensor, but when we are receiving data the line is driven by the sensor, hence when we receive data from the ultrasonic sensor 5V will come into the board, and we don't want that. This time what I've done is, from the Appendix 1 of the NXT HDK (Hardware Developer Kit) you can find here, take the circuitry of a digital pin, (you can find it in the lower right corner of the third page) and replicate it in LEGO-Pi, the assembly of an SDA line will look something like:

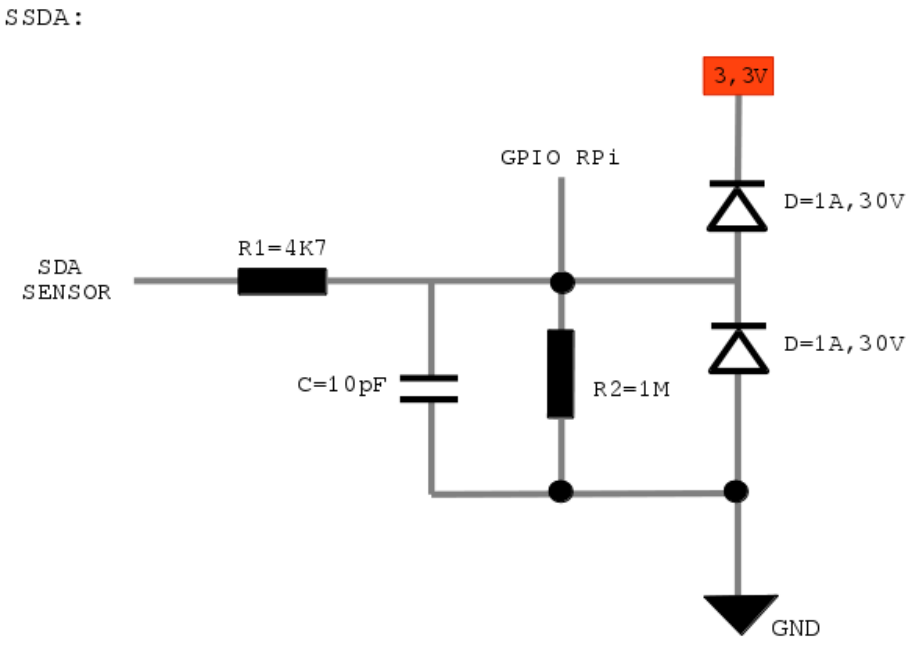


Figure 2.19: Standard SDA circuit

We gonna call this circuit SSDA (standard SDA). The point here are the diodes, and the big resistance added, the capacitor is added to avoid noise in signals. The diodes are schottky diodes, and arranged this way with the big resistor, they will act as a switching diodes, it means, below the specified voltage (3.3V in our case) they will act as an open switch, however, if a higher voltage goes across the circuit they will act as a closed switch, thus dropping some of the voltage (ideally, the difference between the voltage circulating and the specified voltage). Electronics is an analog world, computer science, however, is a digital world, computers only understands signals, so this circuit can be viewed as a "logic one voltage level standardize", this is a clever design from LEGO which I took advantage of.

Digital port final hardware assembly

So with our brand new circuit SSDA we can develop our standard digital port, do not forget the pull-down resistor in the SCL line. The final wire up to connect a digital port to the Raspberry Pi will look something like this:

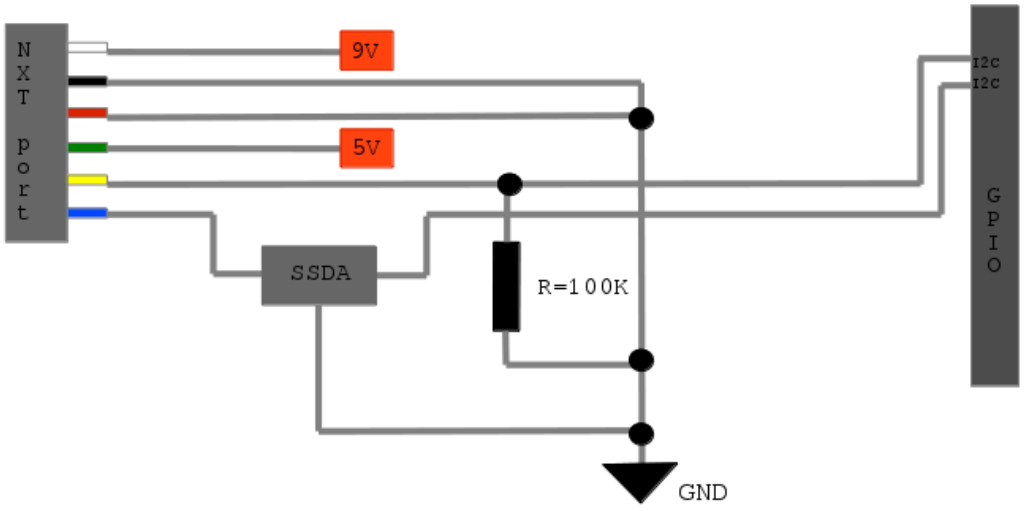


Figure 2.20: Digital port final assembly

As usual black circles means junctions, so if no black circle present wires are crossing. This time however we don't need any integrated device between the sensor and the Raspberry Pi, so the schematic showed in the figure 2.20 is pretty fair with the real assembly of a digital port. As usual you can check out the project schematics of the project in the appendix A of this document or in the `docs/schematics` directory of the LEGO-Pi git-hub .

2.4.2 The digital software interface

The piece of the library that will take care of the digital sensors control is named `lego_digital.c` you can find it in the provided LEGO-Pi git-hub . As mentioned in the last section (2.4.1), LEGO digital sensors use the I2C protocol, to the experienced Raspberry Pi user this would mean good news because our Raspberry Pi supports I2C protocol through the GPIOs 2 (pin 3), 3 (pin 5) and GPIOs 28 (pin 3 [P5 header]), 29 (pin 4 [P5 header]). This pins are used by the BSC (Broadcom Serial Controller) which use the SMB (System Management Bus) bus, this is a bus derived from I2C, however there are some differences between I2C and SMB, the major problem here is the timing, since the less frequency supported for SMB is 10KHz and LEGO digital sensors works at a frequency of 9600 Hz, so we need to find another way to communicate with the sensors.

At this point we need to find another way to expose the I2C driver and link it to some GPIO pins. I found this module from kadamski in the net that will allow us to bit-bang the I2C protocol through two GPIO pins of our desire and at a desired frequency making use of the algorithm `i2c-algo-bit.c` you can find in the Raspberry kernel sources. Nevertheless after some testing I was able to communicate with Hitechnic devices with this module but not with NXT ultrasonic sensor, this is because NXT ultrasonic sensor doesn't seems to be fully I2C compliant, so we need to find yet another solution to achieve a successful communication with all the digital devices we need to support.

Well, at this point panic arise, what can we do if the Linux kernel, one of the greatest achievements of the human being, is not able to handle the communications with our digital sensors.

2.4. The digital interface

To me the first step to take here is try to intercept a communication between the NXT PBX and the ultrasonic sensor, the one causing problems here. To do this a digital oscilloscope would be ideal, however I didn't have one at my disposal so I used our "board for everything" Raspberry Pi to act as an oscilloscope, the montage required was the following:

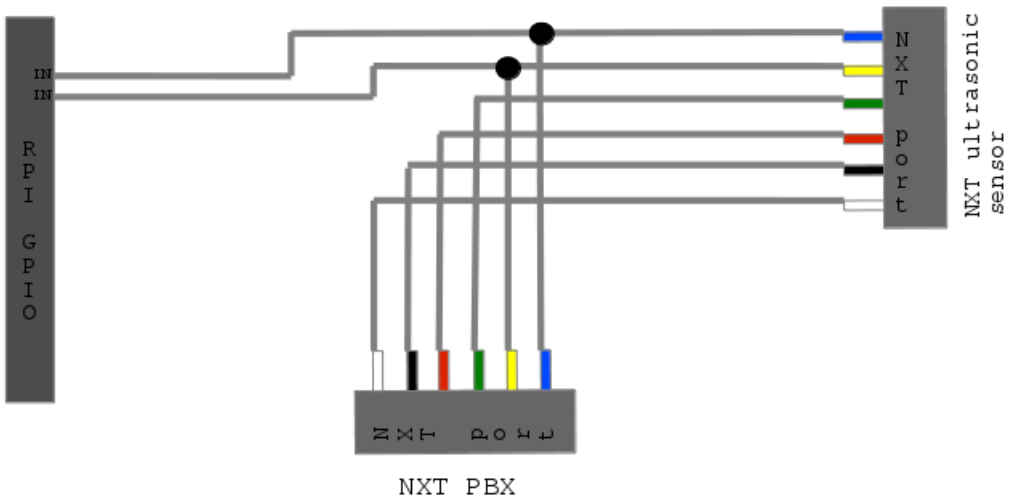


Figure 2.21: Raspberry Pi as oscilloscope

The code to make it work is not populated because it is not part of the project itself, however you can ask for it if you want to see it, the code is very simple, the only thing that it does is put the pins which we plug the NXT PBX and the sensor in input mode and after a configurable amount of time take a sample of the state of the pins until the communication is over, then it dumps the information to a csv formatted file. Here I've to say that I've experimented some problems if the montage is plugged to the I2C GPIO pins (listed above), I guess because of the extra circuitry that this pins have.

Since the I2C frequency of the LEGO devices is very small, the Raspberry Pi was able to handle all the signals successfully and I was able to get a proper graphic of a transference between the NXT PBX and the ultrasonic sensor, the result is the following:

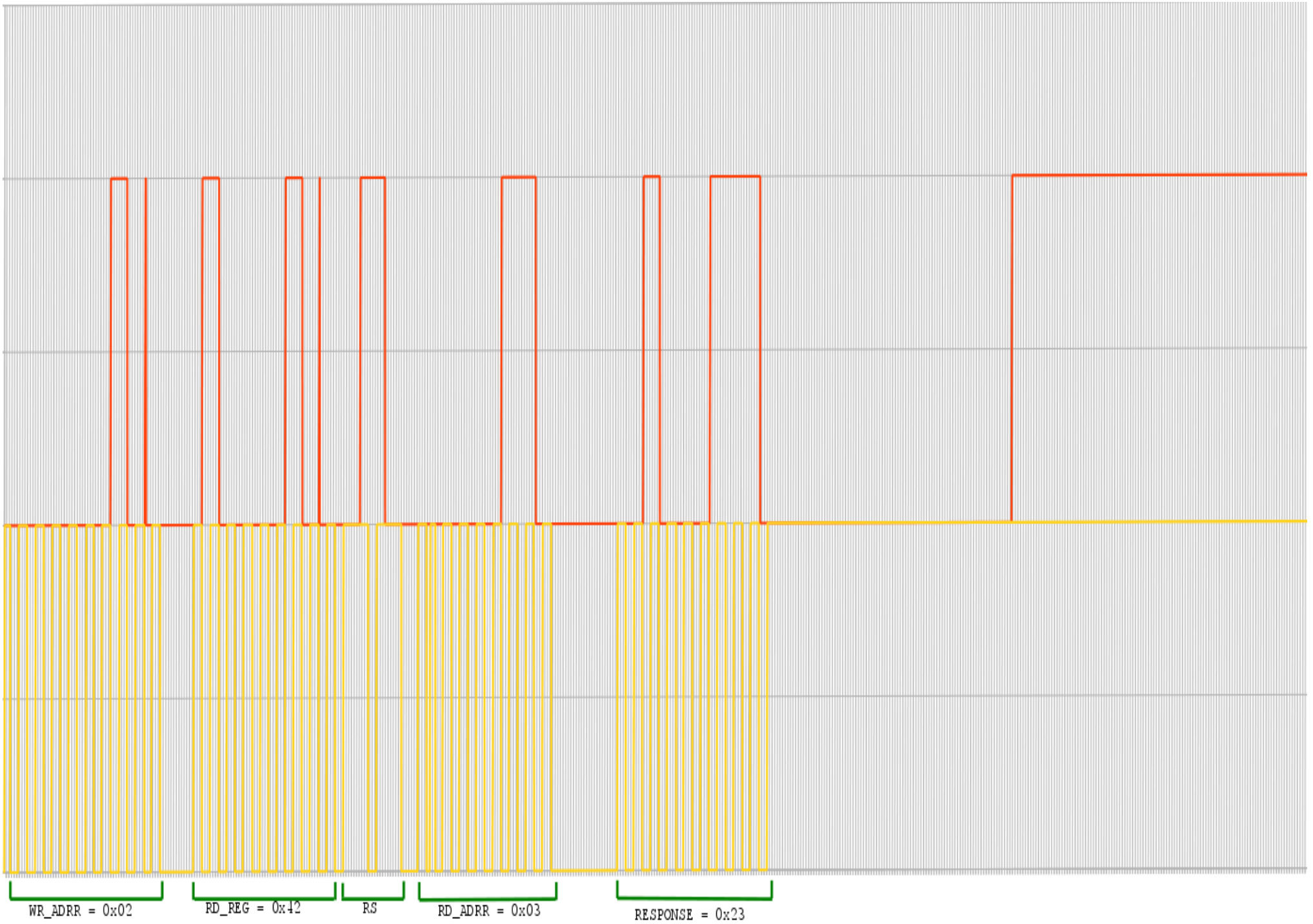


Figure 2.22: Transaction US - NXT PBX

At this point you might want to take a look at this Wikipedia article which explains very well what a standard I2C standard transaction is, it includes a implementation of the bit-banged protocol, here you can find the I2C protocol specification from Philips, what may be useful to you too.

Back to the graphic above (figure 2.22), the red lines represents the SDA line state, while the yellow line represents the SCL line state. As you can see what the I2C master device (the NXT PBX here) does is send clock cycles, depending on the SDA line state on every cycle of the clock line (SCL) we are sending or receiving a 0 or a 1. In the footer of the figure you can see five green colored differentiated blocks, each one represents a byte of data (except the third, which we gonna explain in a jiff), the first four are fully generated for the the NXT PBX, and is a standard sequence generated to request data from the ultrasonic sensor, the last block is the response from the ultrasonic sensor, note here that the clock cycles are still generated by the NXT PBX, however the SDA line is managed by the NXT sensor, this is the way the I2C master device receive data from the slave device. The transaction in the figure 2.22 represents a request from the closest distance the sensor sees, which is stored in the register 0x42 of the sensor, in this particular case it happens to be 35 cm as you can see in the response block of the graphic in hexadecimal format (0x23), the first and fourth blocks are the write address (0x02) and read address (0x03) of the slave device (the ultrasonic sensor here) respectively. So what happens with the third block of the graphic?. As you can see it is tagged as RS, it means repeated start condition, and it consist of a stop condition followed for a start condition, this signals are demanded for the sensor, you can check the ultrasonic sensor I2C protocol in the Appendix 7 of the LEGO HDK (Hardware Developer Kit). The repeated start condition is a well known state to the standard I2C protocol, however, it seems that the timing here matters, as you can see the clock cycles are about four times bigger than a standard clock cycle, I think that this is the main reason why kadamski module and i2c-algo-bit.c weren't able to handle the ultrasonic communications properly. So at this point we need to think about implementing our own I2C driver able to handle the transactions with all the digital devices we need to support.

To develop our I2C driver the best we can do is a modification of the file mentioned above, i2c-algo-bit.c, and try to expose the algorithm with some kind of module similar to what kadamski did with i2-gpio-param, and, probably, modify the ioctl.c file too, letting the kernel take care of the communication with the sensors, at least this would be the more polite and correct solution. However I'm not an experienced kernel module developer, and, besides, it will require a kernel recompilation to the LEGO-Pi user, since we need the option CONFIG_I2C_GPIO enabled in the kernel, which is disabled by default. So I came up with some kind of "emulation" of the I2C driver that will be able to run from user space making use of the mem device, (/dev/mem usually), so we can map a pair of desired GPIO pins and generate the signals needed to communicate with the sensors, the timing is handled by means of the nanosleep C function you can find in the time.h header. The result was a small library that will run under LEGO-Pi, the library is based in the file i2c-algo-bit.c since it was able to communicate with the Hitechnic devices, and we want to standardize the communications with all the digital devices. You can find the file that will take care of the I2C communication (lego_i2c.c) in the directory /lib of the LEGO-Pi git-hub .

There is a few more notes to point here, as you can see in the graphic 2.22, at the end of the transaction the NXT PBX awaits for a while to generate the final stop signal, usually it is done when the slave device needs to compute data, to avoid the master requiring anything

from it, however normally is the slave who stretch the clock line, holding it in low state till the device is ready to handle more petitions. Nevertheless the I2C protocol is partially emulated by the NXT PBX, so this time between the response from the slave and the final stop signal can be viewed as clock stretching emulation, so will do my library to avoid bad readings from the sensor, the times we wait after every transaction has been established by the "empiric" method of test an error, note the irony please. Besides I've to say that my library is not fully I2C compliant, actually is far from it, there is no support for ten bit address, there is no clock stretching support (it just waits a demanded amount of time after every transaction, what can be viewed as a clock stretching emulation), and there is no arbitration support neither, all the LEGO and Hitechnic digital devices use `0x01` seven bit address, so we will need a couple of GPIOs for each port. So as you can see the library was coded only to fit my needs but it handles properly the communication with all the supported digital devices by LEGO-Pi, and, hopefully, with all LEGO compliant digital devices. If you are interested in the functions that this library provide you can check the appendix C of this document (I2C Library Specification).

At the beginning of this section you've been referred to the file `lego_digital.c`, which is the one that will take care of the control of digital sensors. Now we have our new I2C "emulation" library ready to be used by this file. As mentioned in the section 2.4.1 the digital port designed for the LEGO-Pi project is standardized, what means that we can plug any of the supported digital sensors to whatever digital port, so the sensor type plugged into a digital port will be defined dynamically by the user. What LEGO-Pi will do is perform an I2C transaction with the specified sensor, for each sensor there is a set of functions that will return the readings from the sensor in a comfortable format to the user, depending on the sensor type we are dealing with. Besides there are three functions that expose the "low level" I2C library functions to the LEGO-Pi library, these ones are meant to be used by unknown sensor types, it is, LEGO compliant digital sensors that aren't specifically supported by the library. If you are interested in the registers that holds the data for each sensor you can check this site for Hitechnic devices, for the LEGO ultrasonic sensor you may refer to the Appendix 7 of the LEGO HDK. You can check out the specification of the functions provided by the LEGO-Pi library in the appendix B of this document (Library Specification).

Chapter 3

The great gathering

Well, we finally made it, all LEGO and Hitechnic devices from the list in the section 1.4 are fully supported with our implementation, now is time to gather all our stuff and put it all together to see if we can, effectively, build a mobile robot with it. As mentioned in the section 2.1 our Raspberry Pi counts with 21 GPIO pins available, so lets see what can we do with it.

3.1 The official LEGO-Pi assembly

So we have 21 pins to control our peripherals, lets take a look at previous sections to figure out how many pins we need for each kind of port:

- Motor port, from section 2.2.1: 4 pins each
- Analog port, from section 2.3.1: 1 pin each + SPI interface
- Digital port, from section 2.4.1: 2 pins each

As mentioned in section 2.3.1 the SPI interface will be accessed through GPIO 10 (pin 19), 9 (pin 21), 11 (pin 23) , 7 (pin 26) and 8 (pin 24), although the GPIO 7 won't be wired we need to count it here since it can't be used for other purposes while SPI interface is in use, to be honest this is an untested behavior, however the official assembly of LEGO-Pi doesn't make use of the GPIO 7, so it will remain free for the user benefit, making possible to connect another SPI slave device to the montage. So by the moment we have 5 GPIO less available, 16 are unoccupied yet. As explained in section 2.3.1 our A/D converter have four channels, so it will be nice to have four analog ports to profit all the available channels, then we have to take 4 more GPIOs. For the official LEGO-Pi assembly these will be:

- GPIO 3 (pin 5) for the analog port 0, wired to the A/D converter channel 0.
- GPIO 28 (pin 3_P5) for the analog port 1, wired to the A/D converter channel 1.
- GPIO 2 (pin 3) for the analog port 2, wired to the A/D converter channel 2.
- GPIO 29 (pin 4_P5) for the analog port 3, wired to the A/D converter channel 3.

After the analog ports wire up there are 4 GPIO less available, so 12 remains free, if we want to build a mobile robot, at least, we will need 2 motors, actually with one motor and a clever gear train design we can make a robot, however, two motors will lead to a handier robot designs

and the software library was developed thinking in a two motor robot so, if we want two motor ports we need 8 GPIOs, to LEGO-Pi these will be:

Motor port 0:

- GPIO 4 (pin 7), to move the motor forward, it is, wired up to the H-bridge input responsible of the forward motion on motor port 0.
- GPIO 17 (pin 11), to move the motor backward, it is, wired up to the H-bridge input responsible of the backward motion on motor port 0.
- GPIO 27 (pin 13), connected to an encoder output.
- GPIO 22 (pin 15), connected to an encoder output.

Motor port 1:

- GPIO 25 (pin 22), to move the motor forward, it is, wired up to the H-bridge input responsible of the forward motion on motor port 1.
- GPIO 18 (pin 12), to move the motor backward, it is, wired up to the H-bridge input responsible of the backward motion on motor port 1.
- GPIO 24 (pin 18), connected to an encoder output.
- GPIO 23 (pin 16), connected to an encoder output.

So now we have only 4 GPIOs left, since we need two for each digital port all that we can do is to wire up two digital ports for the montage to be ready, to LEGO-Pi these will be:

Digital port 0:

- GPIO 30 (pin 5_P5), for the SDA line.
- GPIO 31 (pin 6_P5), for the SCL line.

Digital port 1:

- GPIO 14 (pin 8), for the SDA line.
- GPIO 15 (pin 10), for the SCL line.

That's it, now we have all our GPIO pins busy, so the final assembly of the project will have 2 motor ports, 4 analog ports and 2 digital ports, not bad after all. At this point you might want to take a look at the project schematics, you can find it in the appendix A of this document (Project Schematics).

3.1.1 Another point of view

So now we have all the montage assembled, however, you may be discontent with the final assembly of the project, you can, for example, think that one encoder for each motor will suffice to you, then you can have an extra digital port, or, maybe you may want to make dBA mode available to 2 of the analog ports while the other 2 will be only capable to handle Hitechnic gyroscope and push sensor, there are several possibilities. However, whatever of the possibilities you choose will probably lead you to a software library rearrangement, which is not a problem since the sources are available, I've to say that I'm not an experienced C developer so you might find the code a little bit disrupted or not optimal, feel free to make any changes.

Another thing to take into account here is that, as briefly mentioned in section 2.4.2, I've been experiencing some problems when wiring the I2C GPIO pins (GPIO 28, 29, 2, 3) to any of the outputs of the ports that generate signals, it is, whatever of the digital lines (SDA, SCL) or any of the encoder outputs, so if you want to rearrange the hardware keep this in mind.

3.2 The power block

Well, we are meant to build a mobile robot with all this stuff we arranged, so we need to find a way to power the whole montage. LEGO batteries will provide us enough power to play with our nearly done robot, lets, however, make some calculations before we assembly the batteries.

For the NXT series there are two kind of batteries, one of them have an storage capacity of 1400 mAh the small one further on, while the other have a capacity of 2200 mAh, we gonna call this one the big one. Now lets try to make an assumption of the minimum time we will be able to work with one or the other battery. First of all we need to think about the voltage that the batteries will deliver, the small one will be able to deliver up to 7.4 V, after some testing I found that some of them even a little bit more when fully loaded, the big one is able to deliver up to 10V, so we can estimate an 9V average more a less. From this 9v we need to divert a 5v bus to power the Raspberry Pi and all the LEGO peripherals and another 3.3V bus for the analog yellow wire ports and the SDA lines of the digital ports, to power the H-bridge and the white wires for the digital sensors the raw voltage from the battery will be taken, note here that the voltage delivered will drop when the batteries discharge, however this won't be a big problem itself, the only thing we will notice is a velocity decrement for the motors, and the readings from the digital sensors may be a little bit less accurate, but we will be able to function anyway.

For the 5V bus we will differentiate between the power of the PCB, it is the Raspberry Pi, and the power of all the periphery. After some testing I've been facing a lot of troubles when powering the peripherals from a switching voltage regulator, the readings from sensors and encoders became totally unstable, so I decided to use a switching regulator to power the Raspberry Pi, what will save us a little battery, and a linear regulator to power all the peripherals, the hardware used is the following.

To power the Raspberry Pi, as mentioned above, a switching voltage regulator is used, it is interesting that the regulator is of the switching kind, since it will provide us an efficiency rate, which means that the extra voltage delivered to it (in reference to the output voltage) is not dissipated into heat, at least not all of it, so it will help us to save a little battery thus making the lifetime of the robot a little bigger. The regulator used in the LEGO-Pi project is the L2596T

from Texas Instruments, to mount it you will require an inductor, a zener diode and a couple of dielectric capacitors, you can find the data sheet of the device here, the montage required for the regulator, to avoid the voltage ripple cause voltage drops on the Raspberry Pi, is the following:

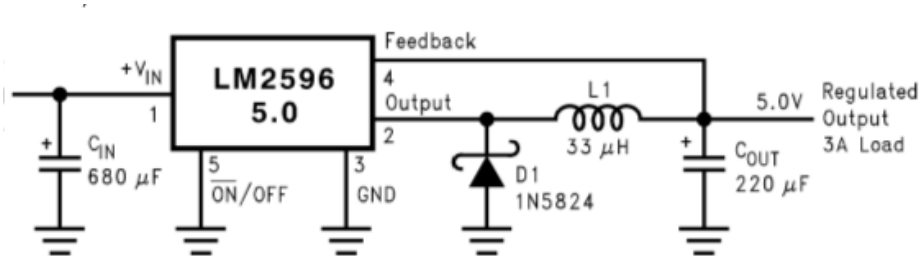


Figure 3.1: Switching voltage regulator assembly

So now we only need to plug the battery to the pin number 1 of the regulator and we will have 5V in the output pin (pin number 2). As you can see in the image 3.1 it is able to deliver up to 3A, what will be more than enough for us. At this point you may want to take a look at the project schematics (you can find it in the appendix A of this document or in the `docs/schematics` of the LEGO-Pi git-hub), since the values for the electronic components indicated in the figure 3.1 vary depending on the input voltage, in the graphic all the electronic components are dimensioned for an input voltage of 12V.

To power all the peripherals a linear voltage regulator is used, linear voltage regulators are more expensive in consumption terms, however it are the best choice if we want to reduce the noise in the output, and it is our main matter here, since the more stable the output from the regulator is the more reliable will be the readings from our peripherals. For the LEGO-Pi project the regulator used to get the 5V bus for our peripherals is the L7805CV from ST Microelectronics, to stabilize the output voltage the montage required will be the following:

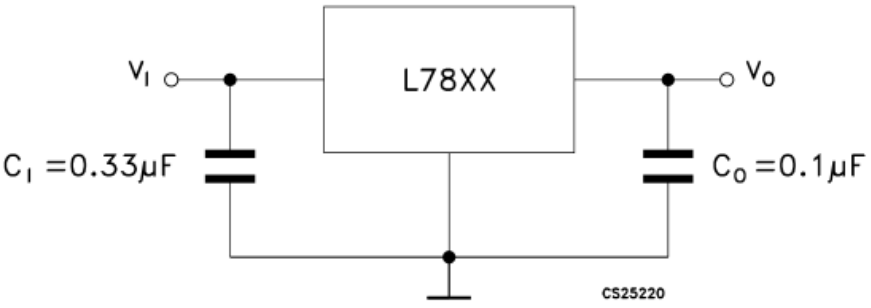


Figure 3.2: Linear voltage regulator assembly

This time the values showed in the image are fair enough with the values of the components needed, this voltage regulator will be able to deliver up to 1.5A, as we will see in the following section it will be more than enough to us. At this point I have to mention that the output voltage from the linear regulator is stable enough to power all the peripherals, the readings from sensors and encoders are fine when powered from the L7805, however, to power the A/D

converter and set the voltage reference of it the voltage must be very accurate, even when getting the voltage from the linear voltage regulator the readings wasn't good enough, so I'm powering the A/D converter from the 5V output of the Raspberry Pi, what solves this issue. You can find a more detailed schematics of the connections in the appendix A of this document or in the `docs/schematics` of the LEGO-Pi git-hub .

With our 5V buses operative is time to build our 3.3V bus, this will be achieved with a very simple circuit, it can be viewed as a very simple fixed input linear voltage regulator, so this time we will drop some of the voltage (the difference between 5V and 3.3V), however to keep the hardware simple I decided to implement it for my own. The circuit we gonna attach to our 5V peripherals bus to get the 3.3V bus will be the following:

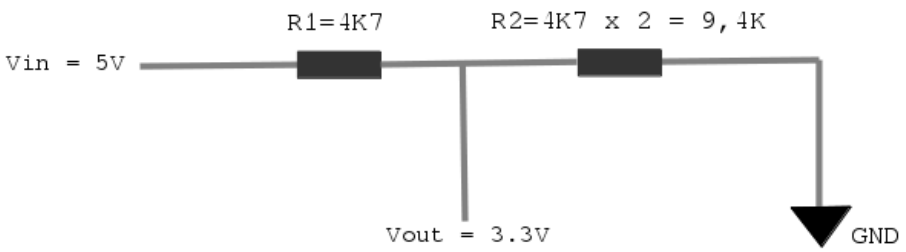


Figure 3.3: 3.3V bus

Note here that 4K7 resistors have been chosen because it are very common, however, what really matters here is the ratio between `R1` and `R2`, you need to use an `R2` resistor with a double resistance of `R1`, then you will have $^2/_3$ of the input voltage between `R1` and `R2`, hence if we apply 5V to the input of the circuit we gonna get 3.3V in the output, what we need. Nevertheless if you have a voltage regulator with a 3.3V output voltage you can use it here.

3.2.1 One probably better design

The reader with a little electronics knowledge will realize here that the montage introduced above is probably not the best. The main problem here was the lack of time to get and test other components that very probably will allow us to build a more efficient power block. The main problem I faced when powering the peripherals from the switching voltage regulator was the instability of the output voltage, so, as explained above that was the main reason to use a second voltage regulator, linear in this case. Nevertheless the voltage is taken from the raw output voltage of the battery in both cases, which isn't optimal at all. The A/D converter will be the main problem here, A/D converters have a sample frequency, it is an A/D will compare the voltage on a specified channel with the reference voltage in several steps, it means that an eventual voltage drop will affect the readings. A switching regulator, in the other hand, is able to provide an efficiency rate opening and closing the voltage output very fast, it is, it have a frequency too. In this article I found what I think is a very good approach to the idea I want to introduce here. There are several dc-dc stabilizer modules in the market, as this one, which is based in the L2596 we use here, that will allow us to adjust the output voltage. If we set the output of the switching voltage regulator a little higher than the 5V we are looking for, it can be done manually dimensioning the components attached to the regulator properly, without needing the module above mentioned, we can drop the voltage of the battery in two steps, it is,

if for example we get 5.5V from the switching voltage regulator we can attach to the output an LDO (Low Dropout) linear voltage regulator such as this one from Texas instruments to get the final 5V bus. With this design we avoid the necessity of having two separated 5V bus, which is not the best, we preserve the maximum the efficiency rate provided by the switching voltage regulator and we get a more stable output from the LDO linear regulator. Nevertheless LDO linear regulators use to have a low current delivery capacity, actually I wasn't able to find one with at least 1.5A, what will be the minimum to ensure a proper working of the montage, but it probably exists, so this is only an idea and it is not implemented and, hence, it is not tested but it might be a good improvement to the power block I did here. In any case, as we will see in the following section, the peripherals consumption on the 5V bus is bounded to 180mA, which represents roughly the 15% of the consumption on the 5V bus, so getting this 5V from a linear regulator attached directly to the battery is not a big tragedy after all.

3.2.2 Lifetime

With all our voltage buses operative is time to get a notion of the lifetime our robot will have, to get a proper assumption we will make a calculation of the consumption of the whole montage operative and for the worst case, it is, with all the ports occupied and consuming the maximum power. First of all lets see what is the maximum consumption for each of the ports, from this site I get what seems a pretty reliable value of the NXT motor consumption, since we gonna assume the motors turns always at the maximum velocity, it is, consuming the maximum power, to get a reliable lower boundary of the lifetime we can say that each motor port consumes 550 mA, if we have 2 motor ports we can say that the consumption will be about 1100 mA. Our new PBX, Raspberry Pi, consumption is between 700 mA and 1000 mA, depending on the devices we plug to the board, HDMI port consumes about 50 mA, however it is disabled in the LEGO-Pi project, since we are using all the GPIO pins, and they are capable to deliver 50 mA altogether we must asses it. LEGO-Pi project doesn't make use of the USB ports, however, the user will possibly want to plug a wifi or a bluetooth dongle, this devices use to consume between 100 mA and 200 mA at most, so for precaution we gonna say that our PBX consumption will be about 1000 mA. From the Extreme NXT book, concretely from the chapter three, I get a reliable consumption value of the NXT Peripherals. NXT PBX have a limit of consumption for the green wire of 180 mA for the eight ports it have, it is a 25 mA average for each port more a less, this boundary its overestimated, as you will find in the book none of the sensors consume 25 mA on the green wire, neither do motor encoders, however, we are interested in a minimal lifetime boundary, so we gonna use this 180 mA limit as a reference since we don't have information about Hitechnic devices consumption. Additionally The NXT PBX have a limit of current in the white wire of 14 mA on each port (for digital sensors), so for the green wire there is a consumption of 180 mA at most for the eight ports (at 5V), additionally we have to take into account the 14 mA for each digital port, this time at 9V. So summarizing we have, for the 5V bus:

- Raspberry Pi: 1000 mA
- Total current consumption: 1000 mA

And for the 9V bus, since we are powering the peripherals from a linear regulator the consumption on the green wire must be accounted here:

- Motors (max. velocity): 1100 mA

- Digital sensors = 28 mA
- LEGO sensors + motor encoders = 180 mA
- Total current consumption: 1308 mA

Now assuming:

BC = Battery capacity,
BV = Battery voltage,
LC_5 = Load current for 5V bus,
LC_9 = Load current for 9V bus,
BV_9 = Bus voltage for 9V bus
BV_5 = Bus voltage for 5V bus, which means 5 volts ...
EFF = Voltage regulator efficiency,

For the 5V bus, given that our voltage regulator have an efficiency rate we can say:

$$Time = \frac{BC \times BV \times EFF}{LC_5 \times BV_5 \times 100},$$

And for the 9V bus, since we are getting the raw voltage from the batteries, it is, no regulator is between the battery and the bus, we can say:

$$Time = \frac{BC \times BV}{LC_9 \times BV_9}, \text{ where } BV = BV_9$$

So, to get a proper lifetime approach the equation to solve is:

$$Time = \frac{BC \times BV}{\left(LC_9 \times BV_9 + \left(\frac{LC_5 \times BV_5 \times 100}{EFF}\right)\right)},$$

Figure 3.4: Lifetime equation

As pointed above there is no regulator between the 9V bus and the batteries, hence, to be fair, we will match the values for BV and BV_9, 9V is an ideal value, ideals in electronics are unlikely to happen. Now substituting the values in the equation 3.4, given that our regulator have about a 75% efficiency rate, we have:

For the big battery:

$$Time = \frac{2200 \times 10}{\left(1308 \times 10 + \left(\frac{1000 \times 5 \times 100}{75}\right)\right)} \approx 1.11 \text{ hours} \approx 67 \text{ minutes},$$

For the small battery:

$$Time = \frac{1400 \times 7.4}{\left(1308 \times 7.4 + \left(\frac{1000 \times 5 \times 100}{75}\right)\right)} \approx 0.63 \text{ hours} \approx 38 \text{ minutes},$$

This is an assumption of the minimum time we will be able to work if our hardware is consuming the maximum current always, without taking into account the voltage dropping of the batteries, however, I've to say that after some testing with the small battery I've been able to work for a roughly three hours, so it is not bad. If you think that this is not enough for you, you can always wire up two batteries in parallel, which will double the capacity, hence, the lifetime.

3.3 Building a support

This time we are very close, we have all the hardware arranged, a software library capable to control all the peripherals and a power block able to provide all the montage independently from any supply, now is time to build a support for our robot.

Since we are dealing with LEGO NXT peripherals, I used LEGO pieces to build my support, it is very rudimentary but it is robust enough to strongly grab the motors, and since LEGO pieces are standardized it guarantees that the motors axis will be aligned, which is the main matter here. I'm using an standard wire wrap board of 16cm x 11cm to build up all the electronic here, so the support will depend on the platform you choose to place all the electronic assembly, however I let you here a couple of pictures of my support.

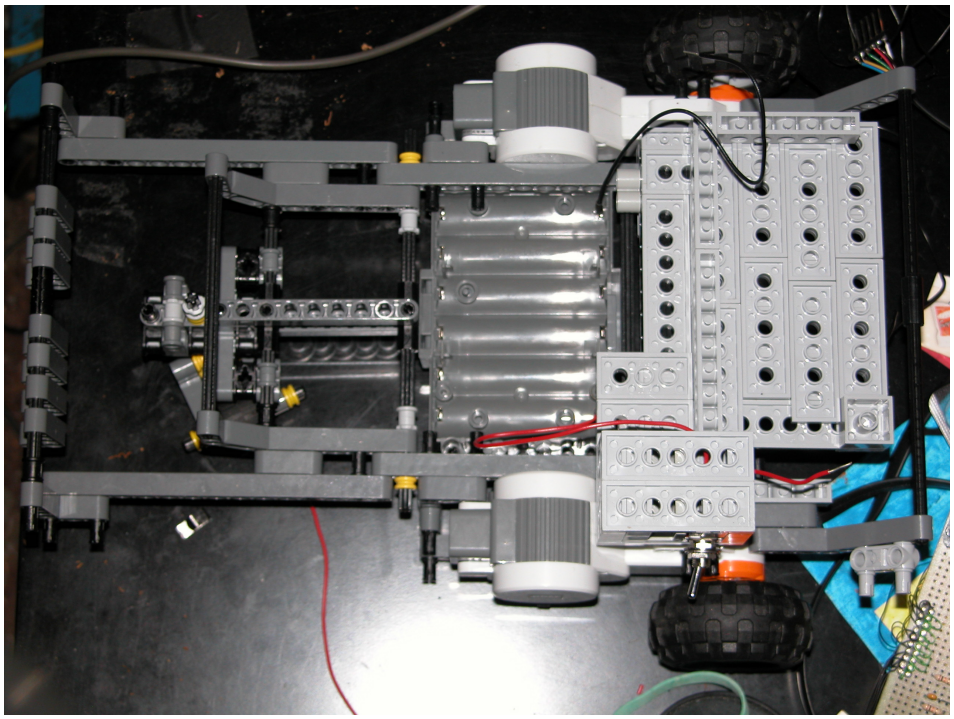


Figure 3.5: Robot support top view

3.3. Building a support

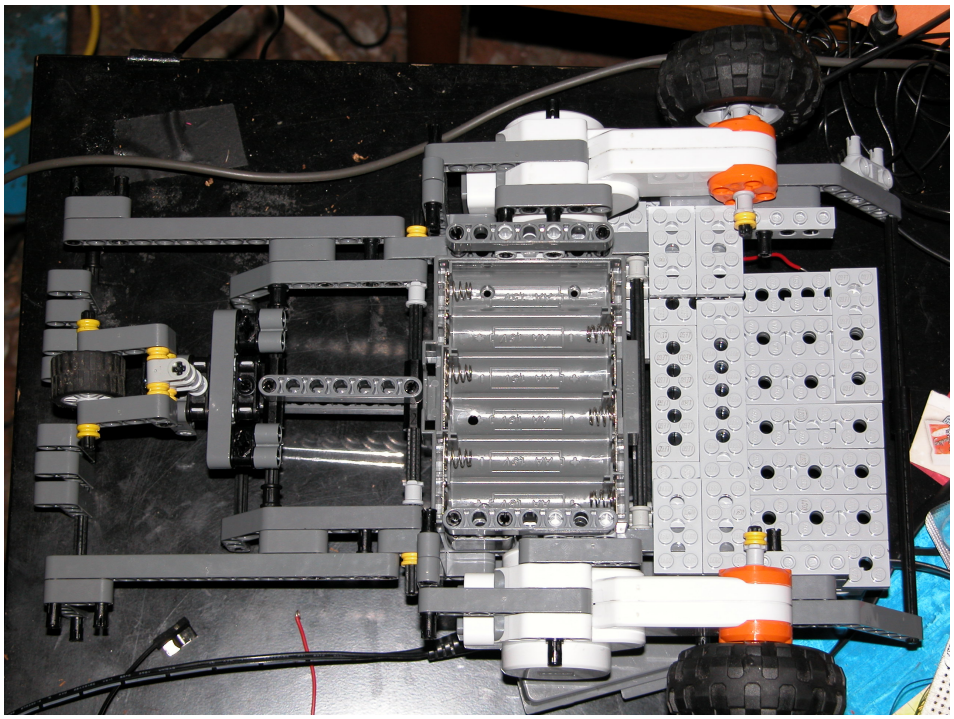


Figure 3.6: Robot support bottom view

And here you can see a picture of the whole montage assembled:

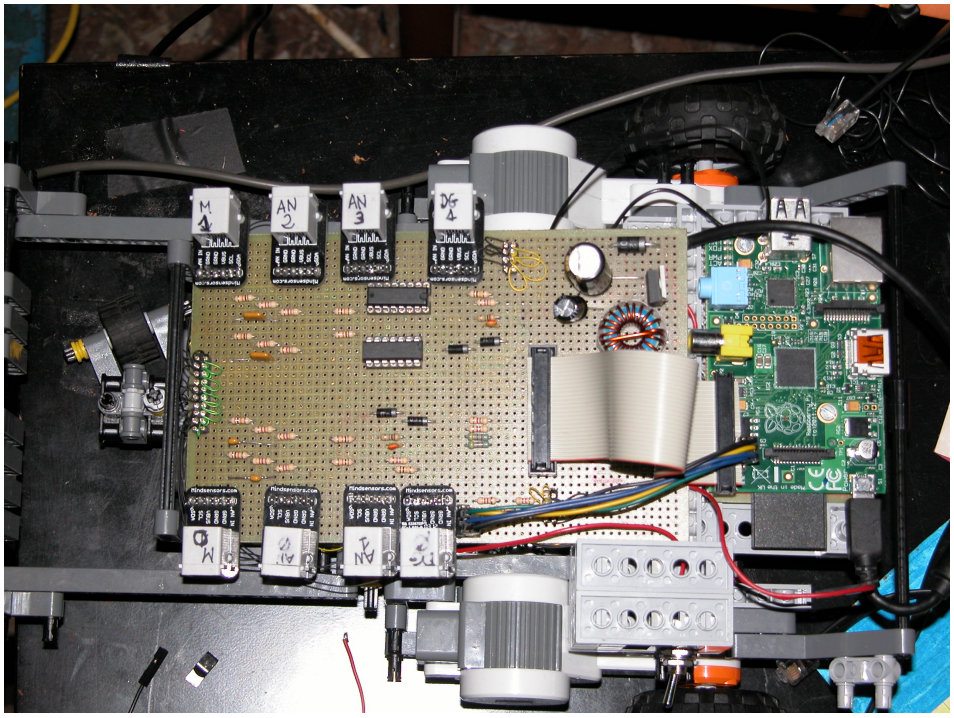


Figure 3.7: Robot assembled

3.4 Improving the model

So now we have our assembly ready to go, nevertheless, as mentioned at the beginning of this document, in the section 1.3, to reduce the budget and simplify the montage, minimizing the hardware was a must at every step of the project, it, however, have its consequences. We delegate all the peripherals management to the Raspberry Pi, which, at the end, is a single ARM core board, for sensors management it is not bad, however motor management turns into a computationally expensive task when we want to implement some "complex" behaviors such as P.I.D control, or motor synchronization. As mentioned in the section 2.2.2 LEGO-Pis P.I.D control approach is far from a real P.I.D control, and actually it does not ensure that we turn at the demanded velocity and, sometimes, because of the software generation of the pulses, which isn't perfect neither, it can even make the motor turn at wrong velocities. As the main problem here is the lack of power of our board a software settlement for this issue is unlikely to be possible, there is however some hardware additions that may help us to improve the motor management.

There is a very interesting project in the net named PiBorg they provide boards to implement robotics with the Raspberry Pi, erstwhile they provide a board named PiBorg (as the project itself), which was developed to control motors in a "professional" environment, however they stop producing it for some reason, the board was using this PIC microcontroller, which will be able to control up to two motors, at least. So with this PIC we can delegate all the motors management to the microcontroller freeing the Raspberry Pi from this heavy tasks, besides we will free 8 GPIO pins, 2 of them must be used to communicate with the microcontroller, however 6 will remain unused, then we can give full support to the analog ports, or, maybe use the MCP3208 A/D converter which is the same we use with 8 channels and have more analog ports, and an extra digital port, there are several possibilities, the important part here is to delegate the motor management to the PIC microcontroller.

Although PiBorg team does not officially produce them PiBorg board anymore, you can find the board info here, with instructions to build your own, and they provide the codes used to implement the closed loop control with the PIC microcontroller above mentioned here too, this code is licensed under the creative commons CC BY-NC-SA licence.

This is a totally conceptual approach, and it is totally untested, however, for a more robust assembly, and, probably for optimal behavior of motors it will be a pretty good improvement.

Chapter 4

Testing

In this chapter I will introduce some tests made to check out the behavior of the peripherals, the methodologies used to implement these tests are mainly functional, it is, the tests tries to show the expected behavior of the library in the major situations a user of the library can face, hence, aims to test all the functionalities provided by the library. Following the pattern of the document I'll begin with the motor testing, and further on I will follow with sensors. If you are interested in the tests sources you can find it in the `tests` directory of the LEGO-Pi git-hub ,to get a briefly usage instructions of each test you can execute any of the tests explained here without parameters. All tests, as well as any program using the LEGO-Pi library, must run with superuser privileges.

All the tests introduced here has been performed with the ported debian version "wheezy raspbian" for Raspberry Pi you can find here, I leave you here the output of `uname -a` in the board:

```
$> uname -a
Linux rpi2 3.10.33+ #1 PREEMPT Mon Mar 24 01:45:46 CET 2014 armv6l GNU/Linux
```

As you see the kernel version used is `3.10.33+`, however `3.10.25+` has been tested too, and everything must work properly, actually LEGO-Pi must be able to run properly with whatever kernel version supporting GPIO interrupt handling, this is any kernel after about June 2012.

4.1 Motor testing

For the motor testing the file containing the sources can be found in the `test` directory of the LEGO-Pi git-hub and it is named `motor_test.c`, in the same directory you will find a `Makefile` file that will take care of the compilation of all the tests in the testing the directory, the present one will produce an executable file named `mt_test`, so, once in the test directory

```
$> make mt_test
```

will produce the executable file in an straightforward way for you.

The `mt_test` will provide 10 different tests, to choose the one we want to run we need to use the `-t / --test` option, this is the only mandatory option for this test, however, additionally we can set several other options that will allow us to test the library all out, all this "optional" settings have their default values you can check out in the usage instructions, I leave you here a printing of the usage instructions for the motor test:

```
$> sudo ./mt_test
```

```
Usage: ./mt_test -t<test_num> [-vpVcsPIDTrebSdCl]
  -t --test      test number to perform [1-10]                    {mandatory}
  -v --vel       velocity [0-200]                                 {60}
  -p --port      motor port [0-1], <2> both                       {0}
  -V --verbose   verbose level [0-3]                              {0}
  -c --calib     calibration samples, [5-20]                      {20 [disabled]}
  -s --step      only apply to some tests                         {10}
  -C --pid       use P.I.D control [no arg]                       {disabled}
  -P --kp        proportional gain for P.I.D control (double)     {0}
  -I --ki        integral gain for P.I.D control (double)         {0}
  -D --kd        derivative gain for P.I.D control (double)       {0}
  -T --ttc       ttc field of P.I.D control [use with caution]    {TTCDEF/36}
  -l --dbg       Set the library in debug mode [no arg]           {disabled}
  -r --turns     turns of the output hub                          {8}
  -e --encod     Encoder lines active per motor [1-2] <3> both    {3}
  -b --psctrl    Position control (double) [0-1]                  {0}
  -S --samples   Samples to store, apply only to some tests       {10}
  -d --dir       direction to move the motor/s 0 = FWD, 1 = BWD   {FWD/0}
```

The default settings are the one you can see at the end of every line between brackets.

Lets begin then with the first test, this is a very basic test that will move the motor first in the direction specified for the `-d / --dir` option for five seconds, forward if no option set, and then in the opposite direction for 5 seconds again, at a velocity specified with the option `-v / --vel` (60 by default), after this the test will move the motor in the demanded direction until the output hub of the motor turns for the number of turns requested with the `-r / --turns` option, which defaults to 8, so from the directory containing the executable file:

```
$> sudo ./mt_test -t1
```

will move the motor plugged to the port 0, which is the default port, at a velocity of 60 (of a maximum of 200), in forward direction for 5 seconds, after this will move the motor backwards 5 seconds more, and, finally will rotate the output hub 8 turns, this is the generated output:

```
Motor 0: ticks received: 3065
Motor 0: ticks received: 3066
Motor 0: ticks received: 5760, turns = 8, enc_active: 2
```

So with this settings everything squares pretty well, the two first lines shows the results of the first two movements for a fixed amount of time, 5 seconds, first rotating in forward direction and later in backward direction, we are getting almost the same number of ticks in both movements, so the behavior is the expected. After the first two rotations the test turned the motor forwards for 8 turns, as we can see both encoder lines are active, so we must get 720 ticks per turn, so $^{5760}/_{720} = 8$, everything goes pretty well then. Now lets run the test again with only the first encoder line active

```
$> sudo ./mt_test -t1 -e1
```

The generated output is the following:

```
Motor 0: ticks received: 1535
Motor 0: ticks received: 1535
Motor 0: ticks received: 2880, turns = 8, enc_active: 1
```

Here we have, again the two first turns return a very close ticks reading, independently of the direction, and, since we are turning at the same velocity in both test executions, but with one encoder less, the ticks amount is more or less the half of the first time, for the last move of the motor this time we will receive 360 ticks per turn so $^{2280}/_{360} = 8$, again everything squares pretty well, we turn the output hub 8 times independently of the encoders we have active. However

to ensure that the assumption of using our quadrature signal as a double precision encoder is right lets run the test with only with the second line of the encoder active this time, the results must be pretty similar to the latter execution

```
$> sudo ./mt_test -t1 -e2
```

The output will be:

```
Motor 0: ticks received: 1537
Motor 0: ticks received: 1538
Motor 0: ticks received: 2880, turns = 8, enc_active: 1
```

The results are pretty fair, as you can see we are getting more a less the same number of ticks for the same amount of time, so seems that we can take profit here of the quadrature signal generated by the LEGO NXT motors and use it as a double precision encoder.
Now lets double the velocity and set back both encoder lines active

```
$> sudo ./mt_test -t1 -v120
```

The output will be the following:

```
Motor 0: ticks received: 5403
Motor 0: ticks received: 5492
Motor 0: ticks received: 5760, turns = 8, enc_active: 2
```

As we can see the two firsts movements nearly doubled the ticks reading from the first execution, while the last remains the same, we seems to lose some ticks here, however, as we will see later on the problem here is that the pulses generated aren't proportional to the velocity number. This is due to the software management of the motors, it is, we don't have any hardware dedicated to treat the encoder feedback from motors and generate the pulses to control the velocity, and this is such an expensive task to our single ARM processor board, lets keep it going to clarify this. Now, as a last check, we will change the motor to the one plugged to the port 1 and run the test with the default settings, since the we are dealing with NXT motors in both cases the results must be similar to those obtained in the first execution, then

```
$> sudo ./mt_test -t1 -p1
```

The results obtained are the following:

```
Motor 1: ticks received: 3079
Motor 1: ticks received: 3083
Motor 1: ticks received: 5760, turns = 8, enc_active: 2
```

Well, pretty fair, so it seems that our library will behave the same for all NXT motors, as expected. At this point you can make more tests with the first test, such as activate P.I.D control or position control, however it will be tested further on, hence, I will carry on with the test number two.

The test number 2 is a very simple test to ensure that the library returns the program flow to the user code when a motor is set, so the user can continue working, performing readings of the sensors or whatever he wants while the motor moves, the library will take care also of stopping the motor when it arrive to the desired set point in this test, however, we have the option to move indefinitely too, and the program flow will be returned to the user anyway, for the user to stop the motor on demand. We will use the default settings first, the test will make the motor on the port 0 rotate 8 turns in forward direction at a velocity 60, while the motor is moving the test will print each second the amount of ticks stored till this moment.

```
$> sudo ./mt_test -t2
```

The output produced is the following:

```
moving 0
moving 565
moving 1191
moving 1817
moving 2444
moving 3070
moving 3695
moving 4321
moving 4947
moving 5572
pthread_create says: 1, ticks received: 5760, ticks expected: 5760
```

As we can see, for 8 turns we receive 5760 ticks, which is the correct with both encoder lines enabled, now if we set just one encoder line, but let the speed in the default value (60) we will see that the test output the same number of lines, but the ticks will be more a less the half of the first case for each line, as we saw above both encoder lines behave the same, so we gonna let the first line active on the port 0, however the test can be run with `-e2` option and it will behave the same.

```
$> sudo ./mt_test -t2 -e1
```

The output will be:

```
moving 0
moving 286
moving 599
moving 913
moving 1226
moving 1539
moving 1851
moving 2165
moving 2478
moving 2791
pthread_create says: 1, ticks received: 2880, ticks expected: 2880
```

Almost exact, so again we are receiving the expected ticks and, now we can say, at the expected time. Now, as a final test, we will set back both encoder lines, however we will double the velocity, hence the number of lines printed must be reduced to the half of the latter cases (because the delay between printings is fixed).

```
$> sudo ./mt_test -t2 -v120
```

The output will be:

```
moving 0
moving 985
moving 2080
moving 3181
moving 4284
moving 5386
pthread_create says: 1, ticks received: 5760, ticks expected: 5760
```

So, as you can see, everything is working as expected, The lines have reduced to the half and the values for the same number of line are almost the double of the first execution, pretty fair. Again you can do more tests with P.I.D and position control here if you want.

I will jump to test number 6 now, this is a very basic test to ensure that we can change the default ISRs of the library that will be executed every time a tick comes, and disable and re-enable encoder lines on demand. As the latter tests this one only allows one motor at a time, I will run the test only for the motor plugged to port 0, the default, since this time there is no hardware testing, is just a software issue. The default settings for turns and velocity will be fine too. for this test two auxiliary ISRs will be used to change the defaults, the shape of it will be:

```
/* < Global scope > */

static void isr_print_X(void){
  if(mt->moving){
    mt->encX->tics++;
    if ((mt->enc1->tics%300) == 0)
      printf("Hi, I'm the ISR X\n");
  }
  clock_gettime(CLK_ID, &mt->encX->tmp);
}
```

Substituting the X values for the number of the encoder line of the motor you will get the name of the ISR that will be used for each encoder line. Then:

```
$> sudo ./mt_test -t6
```

will produce an output like:

```
both encoders active, m_e1: 27, m_e2: 22
ticks received: 5760, ticks expected: 5760, tics e1: 2879, tics e2: 2881
disabling encoder 2 ...
m_e2 disabled , m_e1: 27, m_e2: -1,
ticks received: 2880, ticks expected: 2880, tics e1: 2880, tics e2: 0
re-enabling encoder 2 custom ISR ...

both encoders active, m_e1: 27, m_e2: 22
Hi, I'm the ISR 2
Hi, I'm the ISR 2
Hi, I'm the ISR 2
Hi, I'm the ISR 2
Hi, I'm the ISR 2
Hi, I'm the ISR 2
Hi, I'm the ISR 2
ticks received: 5760, ticks expected: 5760, tics e1: 2880, tics e2: 2880
disabling encoder 1 ...

m_e1 disabled , m_e1: -1, m_e2: 22,
Hi, I'm the ISR 2
Hi, I'm the ISR 2
Hi, I'm the ISR 2
Hi, I'm the ISR 2
Hi, I'm the ISR 2
Hi, I'm the ISR 2
Hi, I'm the ISR 2
ticks received: 2880, ticks expected: 2880, tics e1: 0, tics e2: 2880
re-enabling encoder 1 custom ISR ...

both encoders active, m_e1: 27, m_e2: 22,
Hi, I'm the ISR 1
Hi, I'm the ISR 2
Hi, I'm the ISR 1
Hi, I'm the ISR 2
Hi, I'm the ISR 1
Hi, I'm the ISR 2
Hi, I'm the ISR 1
Hi, I'm the ISR 1
Hi, I'm the ISR 2
Hi, I'm the ISR 1
Hi, I'm the ISR 2
Hi, I'm the ISR 1
Hi, I'm the ISR 2
Hi, I'm the ISR 1
Hi, I'm the ISR 1
Hi, I'm the ISR 2
ticks received: 5760, ticks expected: 5760, tics e1: 2879, tics e2: 2881
back to defaults ...
```

```
both encoders active, (default ISRs) m_e1: 27, m_e2: 22,
ticks received: 5760, ticks expected: 5760, tics e1: 2879, tics e2: 2881

trying to disable both encoders at a time ...
mt_reconf: Configuration let motor 0 without encoders.
```

As you can see, first the test will turn for the turns demanded, 8 in this case, with both encoder lines active, then it will disable the second encoder line of the motor plugged to the requested port, 0 in this case, as you can see the pin of the encoder line will became -1 (ENULL) when it is disabled, after it will rotate again for the amount of turns requested, with just one encoder line enabled, hence, receiving the half of ticks. Once the latter movement is done the test will re-enable the second encoder line, this time however, it will set as the ISR of the line the custom ISR made to this test, as explained above, and, again move for the requested turns, as you can see the ISR is printing the "Hello World" message. Then it will disable the first encoder line, but the second will remain enabled and untouched, so the custom ISR will remain active, as you can see the second ISR keeps printing the message, however the ticks received are the half of the latter movement. After this the test will re-enable the encoder line one with the custom ISR made for this test, as you can see now both lines are printing the message and the ticks received are the double of the last rotation. After both custom ISRs have been active the test will let both encoder lines active but with the default ISRs and make a last rotation, as you can see the ticks received are the expected. Finally the test will try to disable both encoder lines, as you can see the library prints an advice message to alert the user about the situation, the function `mt_reconf` will return false, however the program won't terminate, but all the functions designed to move till a set point will be useless and the library won't allow the user to call it with this configuration.

Now I will continue with the test number 5, from now on the tests will be less functional and will be more analytic, it is, we won't test library functionalities here, instead we will try to get some statistics to really ensure that the motors behavior is the expected,in this test we will turn a motor to a desired velocity, the turns will increase from the amount requested trough the `-s / --step` option to 40 (MAXM) turns, the times between ticks spent will be computed in two different ways, we will store the time between ticks via the ISRs of the encoders and we will compute the total time of the rotation externally and divide the amount of time for the ticks received, if the to values match and remain stables for the different amount of turns we will know that the ticks we are receiving are stables and the times correct. To store the delay between ticks we will use a set of ISRs designed to do the job, the ISRs you will find in the test source file will be something like:

```
typedef struct timespec TSPEC;  /* < From time.h > */

/* < Global scope > */

TSPEC tXY;
double *acumZ = NULL

extern void dbg_isr_XY(void){
  if(mtX->moving){
    clock_gettime(CLK_ID, &tXY);
    acumZ[mtX->encY->tics] = difft(&mtX->encY->tmp, &tXY);
    pthread_mutex_lock(&mtX->encY->mtx);
    mtX->encY->tics++;
    pthread_mutex_unlock(&mtX->encY->mtx);
    clock_gettime(CLK_ID, &mtX->encY->tmp);
  }
}
```

4.1. Motor testing

There will be four functions with the "shape" like above exposed, one for each encoder line of each motor, you can figure out what of them will be used for each motor and encoder line by substituting the `x` values for the motor port+1 and the `y` values for the encoder line of the motor, so, as an instance, motor port 0 encoder line 1 will use the ISR `dbg_isr_11`, and so on. Additionally to compute the time difference between ticks an auxiliary `TSPEC` struct is needed for each ISR, following the pattern of the function naming you can get the names of this auxiliary variables substituting the `x` value for the motor port+1 and the `y` values for the encoder line of the motor that will make use of it. Finally to store the delays an array of floating point values is needed for each line, in this case you can get the name of the arrays by substituting the values `z` for 1 and 2 for the motor port 0, and 3 and 4 for the motor port 1, so, `dbg_isr_11` will use `acum1`, and `dbg_isr_22` will use `acum4`. The times, in microseconds, will be arranged as following, the time between tick `n` and tick `n+1` will be stored in the `n` position of the array.

Now that we have our interrupt handling explained we can carry on with the test number 5, for this first run we will use the default settings for the velocity, we will set the step to 5 so each increment of the rotation will be of 5 turns, it is, we will rotate from 5 turns to 40 turns, I will run the test for both motors at a time, and with both encoders enabled. Actually here the important setting is the velocity for us to know if the ticks per turn at a certain velocity are accurate. I will set, however, the verbose level to 1, then at every loop, it is, for each rotation, we will get a brief result printing.

```
$> sudo ./mt_test -t5 -s5 -p2 -V1
```

has, this time, produced this output:

```
MOTOR 0:
turns: 5, telapsed: 5.86686 sec
txturn: 1.17337 sec
tbticks_total: 1629.68194 micros
ticks/sec: 613.61656

enc1,
mean:       2972.280379
tbticks_e1: 3270.00000
range_min:  2711.800000
range_max:  3282.000000

enc2,
mean:       2946.965651
tbticks_e2: 3248.00000
range_min:  2706.800000
range_max:  3265.000000

-----------------------------------------------------------------------------------------

MOTOR 1:
turns: 5, telapsed: 5.86686 sec
txturn: 1.17337 sec
tbticks_total: 1629.68194 micros
ticks/sec: 613.61656

enc1,
mean:       2821.069367
tbticks_e1: 3255.00000
range_min:  2444.000000
range_max:  3206.800000

enc2,
mean:       2852.737486
tbticks_e2: 3262.00000
range_min:  2606.400000
```

```
range_max:  3175.000000

-------------------------------------------------------------------------------------------

MOTOR 0:
turns: 10, telapsed: 11.66037 sec
txturn: 1.16604 sec
tbticks_total: 1619.49444 micros
ticks/sec: 617.47638

enc1,
mean:       2961.851739
tbticks_e1: 3243.00000
range_min:  2641.800000
range_max:  3270.000000

enc2,
mean:       2947.969209
tbticks_e2: 3234.00000
range_min:  2692.000000
range_max:  3261.000000

-------------------------------------------------------------------------------------------

MOTOR 1:
turns: 10, telapsed: 11.66037 sec
txturn: 1.16604 sec
tbticks_total: 1619.49444 micros
ticks/sec: 617.47638

enc1,
mean:       2819.463245
tbticks_e1: 3230.00000
range_min:  2492.600000
range_max:  3202.000000

enc2,
mean:       2874.218881
tbticks_e2: 3247.00000
range_min:  2661.000000
range_max:  3177.000000

-------------------------------------------------------------------------------------------

MOTOR 0:
turns: 15, telapsed: 17.43276 sec
txturn: 1.16218 sec
tbticks_total: 1614.14306 micros
ticks/sec: 619.52331

enc1,
mean:       2967.066926
tbticks_e1: 3231.00000
range_min:  2749.600000
range_max:  3276.000000

enc2,
mean:       2935.691767
tbticks_e2: 3224.00000
range_min:  2643.400000
range_max:  3257.000000

-------------------------------------------------------------------------------------------

MOTOR 1:
turns: 15, telapsed: 17.43276 sec
txturn: 1.16218 sec
tbticks_total: 1614.14306 micros
```

```
ticks/sec: 619.52331

enc1,
mean:       2820.970398
tbticks_e1: 3224.00000
range_min:  2444.800000
range_max:  3213.200000

enc2,
mean:       2859.625255
tbticks_e2: 3232.00000
range_min:  2587.800000
range_max:  3183.000000

-------------------------------------------------------------------------------------------

MOTOR 0:
turns: 20, telapsed: 23.37971 sec
txturn: 1.16898 sec
tbticks_total: 1623.59028 micros
ticks/sec: 615.91870

enc1,
mean:       2978.513841
tbticks_e1: 3252.00000
range_min:  2714.000000
range_max:  3291.000000

enc2,
mean:       2956.474115
tbticks_e2: 3242.00000
range_min:  2683.000000
range_max:  3275.200000

-------------------------------------------------------------------------------------------

MOTOR 1:
turns: 20, telapsed: 23.37971 sec
txturn: 1.16898 sec
tbticks_total: 1623.59028 micros
ticks/sec: 615.91870

enc1,
mean:       2834.453069
tbticks_e1: 3246.00000
range_min:  2446.200000
range_max:  3228.000000

enc2,
mean:       2884.228119
tbticks_e2: 3248.00000
range_min:  2644.000000
range_max:  3201.000000

-------------------------------------------------------------------------------------------

MOTOR 0:
turns: 25, telapsed: 29.50481 sec
txturn: 1.18019 sec
tbticks_total: 1639.15556 micros
ticks/sec: 610.07012

enc1,
mean:       3013.498553
tbticks_e1: 3284.00000
range_min:  2720.600000
range_max:  3331.000000
```

```
enc2,
mean:       2984.637202
tbticks_e2: 3272.00000
range_min:  2702.800000
range_max:  3313.000000

-------------------------------------------------------------------------------------

MOTOR 1:
turns: 25, telapsed: 29.50481 sec
txturn: 1.18019 sec
tbticks_total: 1639.15556 micros
ticks/sec: 610.07012

enc1,
mean:       2874.985119
tbticks_e1: 3274.00000
range_min:  2461.000000
range_max:  3268.000000

enc2,
mean:       2926.964961
tbticks_e2: 3281.00000
range_min:  2684.800000
range_max:  3257.000000

-------------------------------------------------------------------------------------

MOTOR 0:
turns: 30, telapsed: 34.78040 sec
txturn: 1.15935 sec
tbticks_total: 1610.20278 micros
ticks/sec: 621.03940

enc1,
mean:       2956.710736
tbticks_e1: 3224.00000
range_min:  2684.000000
range_max:  3269.000000

enc2,
mean:       2946.399352
tbticks_e2: 3216.00000
range_min:  2712.000000
range_max:  3252.000000

-------------------------------------------------------------------------------------

MOTOR 1:
turns: 30, telapsed: 34.78040 sec
txturn: 1.15935 sec
tbticks_total: 1610.20278 micros
ticks/sec: 621.03940

enc1,
mean:       2828.514854
tbticks_e1: 3218.00000
range_min:  2432.800000
range_max:  3222.000000

enc2,
mean:       2880.922550
tbticks_e2: 3222.00000
range_min:  2653.000000
range_max:  3198.000000

-------------------------------------------------------------------------------------
```

4.1. Motor testing

```
MOTOR 0:
turns: 35, telapsed: 40.47035 sec
txturn: 1.15630 sec
tbticks_total: 1605.96528 micros
ticks/sec: 622.67816

enc1,
mean:       2949.013830
tbticks_e1: 3216.00000
range_min:  2627.000000
range_max:  3262.000000

enc2,
mean:       2929.137723
tbticks_e2: 3207.00000
range_min:  2663.000000
range_max:  3250.000000

-----------------------------------------------------------------------------------------

MOTOR 1:
turns: 35, telapsed: 40.47035 sec
txturn: 1.15630 sec
tbticks_total: 1605.96528 micros
ticks/sec: 622.67816

enc1,
mean:       2813.518842
tbticks_e1: 3208.00000
range_min:  2387.800000
range_max:  3215.000000

enc2,
mean:       2871.223070
tbticks_e2: 3214.00000
range_min:  2637.000000
range_max:  3191.000000

-----------------------------------------------------------------------------------------

MOTOR 0:
turns: 40, telapsed: 46.37959 sec
txturn: 1.15949 sec
tbticks_total: 1610.40139 micros
ticks/sec: 620.96281

enc1,
mean:       2954.673966
tbticks_e1: 3224.00000
range_min:  2632.800000
range_max:  3270.000000

enc2,
mean:       2941.534537
tbticks_e2: 3217.00000
range_min:  2685.800000
range_max:  3258.000000

-----------------------------------------------------------------------------------------

MOTOR 1:
turns: 40, telapsed: 46.37959 sec
txturn: 1.15949 sec
tbticks_total: 1610.40139 micros
ticks/sec: 620.98437

enc1,
mean:       2831.634224
```

```
tbticks_e1: 3216.00000
range_min:  2458.000000
range_max:  3221.000000

enc2,
mean:       2872.116882
tbticks_e2: 3224.00000
range_min:  2628.200000
range_max:  3200.000000

-------------------------------------------------------------------------------------------

MOTOR: 0, step: 5, vel: 60

                          AVERAGE                ABS_DEV

secs  x turn:             1.16574                0.00641
ticks x sec :             617.66338              3.39294

enc: 1,
average:                  2969.20125             14.17226
t_btwn_ticks:             3243.00000             19.25000
range_min:                2685.20000             38.80000
range_max:                3281.37500             14.96875
min_rmin:                 2627.00000
max_rmax:                 3331.00000

enc: 2,
average:                  2948.60119             10.97723
t_btwn_ticks:             3232.50000             16.50000
range_min:                2686.10000             17.30000
range_max:                3266.40000             13.85000
min_rmin:                 2643.40000
max_rmax:                 3313.00000


MOTOR: 1, step: 5, vel: 60

                          AVERAGE                ABS_DEV

secs  x turn:             1.16574                0.00641
ticks x sec :             617.66338              3.39294

enc: 1,
average:                  2830.57614             12.33600
t_btwn_ticks:             3233.87500             18.34375
range_min:                2445.90000             18.55000
range_max:                3222.00000             13.00000
min_rmin:                 2387.80000
max_rmax:                 3268.00000

enc: 2,
average:                  2877.75465             14.71292
t_btwn_ticks:             3241.25000             18.25000
range_min:                2637.77500             22.92500
range_max:                3197.75000             16.25000
min_rmin:                 2587.80000
max_rmax:                 3257.00000
```

As you can see for each amount of turns the test prints a brief result for each motor and all the active encoders it have. After all the loops are done it'll print the final results that will be the average of all the computed values. The important fields in this test are the `average` (`mean` in the partial printings), that corresponds to the time between ticks computed by the ISRs, and the `t_btwn_ticks` field (`tbticks_eX` in the partial printings) which is the time between ticks computed externally, the `range_min` and `range_max` fields are the 20-quantile and the 80-quantile of

the sample taken by the ISRs at each loop respectively. The header fields `secs x turn` and `ticks x sec` are there to give a notion of "real" velocity so we can identify quickly if several executions for the same LEGO-Pi velocity are behaving properly, as you can see the values of `t_btwn_ticks` and `average` match pretty well, at every step of the execution, so we can say that the ticks per turn are fair enough with the reality, lets now, however, make a second execution of the test for 120 velocity, to see if the header fields `secs x turn` and `ticks x sec` are about the half and the double respectively from the latter execution. This time, once you saw how the test behave, I will disable verbose.

```
$> sudo ./mt_test -t5 -s5 -p2 -v120
```

has produced the following output:

```
MOTOR: 0, step: 5, vel: 120

                        AVERAGE                 ABS_DEV

secs  x turn:           0.66246                 0.00381
ticks x sec :           1086.93060              6.22584

enc: 1,
average:                1684.40504              10.88589
t_btwn_ticks:           1847.37500              11.56250
range_min:              1489.05000              10.00000
range_max:              2027.75000              5.25000
min_rmin:               1475.00000
max_rmax:               2039.00000

enc: 2,
average:                1694.39898              6.26270
t_btwn_ticks:           1832.12500              10.87500
range_min:              1523.37500              7.87500
range_max:              2008.25000              5.93750
min_rmin:               1508.00000
max_rmax:               2019.00000


MOTOR: 1, step: 5, vel: 120

                        AVERAGE                 ABS_DEV

secs  x turn:           0.66246                 0.00381
ticks x sec :           1086.93060              6.22584

enc: 1,
average:                1678.60306              2.45453
t_btwn_ticks:           1855.62500              10.43750
range_min:              1514.72500              5.27500
range_max:              2033.37500              2.78125
min_rmin:               1506.00000
max_rmax:               2039.00000

enc: 2,
average:                1672.51324              6.23470
t_btwn_ticks:           1823.87500              13.12500
range_min:              1572.37500              8.12500
range_max:              1916.15000              2.11250
min_rmin:               1559.00000
max_rmax:               1921.00000
```

As you can see everything squares pretty well, so we can say that our velocities are fair with reality, as you can see the external times are indifferently of the velocity, so here don't seems that we are losing any tick, why 120 velocity isn't exactly the double of 60 velocity then?, well, as it seems the pulses generated aren't exactly what are expected to be, it is, the velocity number

won't be exactly proportional to the power delivered to the motor, here, however, I've an ace in the hole, if we set the velocity to 200, the maximum allowed, the library won't use any PWM management, it just sets the GPIO pin involved in the movement to high state, thus opening the "gates" of the H-Bridge constantly, hence delivering all the power to the motors, then we can get a reliable value of the statistics when all the power is being delivered to the motors. Here I've to say that this tests are done with a transformer directly plugged to the 220V standard output of any installation, so the values showed here may vary a bit when working with batteries, actually it will vary for sure when batteries discharge, however the ratio between the maximum velocity (when we set the GPIO pin to high state) and the rest of velocities (that we generate via software PWM) must remain stable, so to get a notion of the values for a maximum velocity we will run the test separately on each motor since the little gap between stops of both motors can affect a little bit the external times computation and we are interested in the more precise values here.

```
$> sudo ./mt_test -t5 -v200 -p0 -s5
```

Has produced the following output:

```
MOTOR: 0, step: 5, vel: 200

                        AVERAGE                 ABS_DEV

secs  x turn:           0.41840                 0.00380
ticks x sec :           1721.05588              15.50947

enc: 1,
average:                1151.60543              4.71652
t_btwn_ticks:           1166.50000              8.87500
range_min:              854.75000               1.87500
range_max:              1437.62500              1.87500
min_rmin:               851.00000
max_rmax:               1442.00000

enc: 2,
average:                1151.51290              5.18686
t_btwn_ticks:           1156.75000              12.18750
range_min:              872.30000               1.87500
range_max:              1422.12500              1.90625
min_rmin:               868.00000
max_rmax:               1426.00000
```

And for the motor on the port 1

```
$> sudo ./mt_test -t5 -v200 -p1 -s5
```

This output was produced:

```
MOTOR: 1, step: 5, vel: 200

                        AVERAGE                 ABS_DEV

secs  x turn:           0.39664                 0.00414
ticks x sec :           1815.55077              18.68734

enc: 1,
average:                1144.03828              3.51361
t_btwn_ticks:           1111.25000              9.81250
range_min:              862.75000               4.37500
range_max:              1436.12500              3.87500
min_rmin:               857.00000
max_rmax:               1443.00000

enc: 2,
average:                1137.94890              3.60682
```

```
t_btwn_ticks:           1091.37500              14.96875
range_min:              941.50000               5.62500
range_max:              1358.25000              1.43750
min_rmin:               934.00000
max_rmax:               1360.00000
```

Now, after some testing, I've been able to locate what seems a good notion of the middle velocity on both ports for a value of 92, then if we run

```
$> sudo ./mt_test -t5 -v92 -p0 -s5
```

For the motor port 0 the values found are:

```
MOTOR: 0, step: 5, vel: 92

                        AVERAGE                 ABS_DEV

secs  x turn:           0.81160                 0.01365
ticks x sec :           887.66990               14.29990

enc: 1,
average:                2222.17640              38.19141
t_btwn_ticks:           2254.37500              38.06250
range_min:              2031.25000              43.43750
range_max:              2406.75000              35.06250
min_rmin:               2003.00000
max_rmax:               2547.00000

enc: 2,
average:                2224.25509              38.18670
t_btwn_ticks:           2253.50000              38.00000
range_min:              2055.75000              42.81250
range_max:              2385.37500              34.40625
min_rmin:               2029.00000
max_rmax:               2523.00000
```

And for the motor port 1

```
$> sudo ./mt_test -t5 -v92 -p1 -s5
```

the values found are:

```
MOTOR: 1, step: 5, vel: 92

                        AVERAGE                 ABS_DEV

secs  x turn:           0.79598                 0.00266
ticks x sec :           904.56252               3.01321

enc: 1,
average:                2195.62268              3.11911
t_btwn_ticks:           2206.87500              6.87500
range_min:              1996.62500              0.78125
range_max:              2399.75000              1.43750
min_rmin:               1996.00000
max_rmax:               2404.00000

enc: 2,
average:                2198.45887              3.73756
t_btwn_ticks:           2214.37500              8.31250
range_min:              2123.00000              1.00000
range_max:              2265.37500              1.21875
min_rmin:               2121.00000
max_rmax:               2268.00000
```

Ok then, will the velocities follow that ratio?, with a simple script I've been able to compute a couple of graphics, one for each motor port that will be self explaining for how the velocities will evolve related to the maximum velocity, the one we generate without PWM, for the motor

port 0 the values for the graphic has been found with the following line

```
$> for vel in `seq 15 10 200`; do sudo ./mt_test -t5 -p0 -s10 -v"$vel" >> outs/graph0; done
```

As you can see the step has been increased to 10 to speed up the test, the values found will correspond to the velocities from 15 to 195, and we will use the values found for max velocity in the above executions, the graphics that this values let are the following:

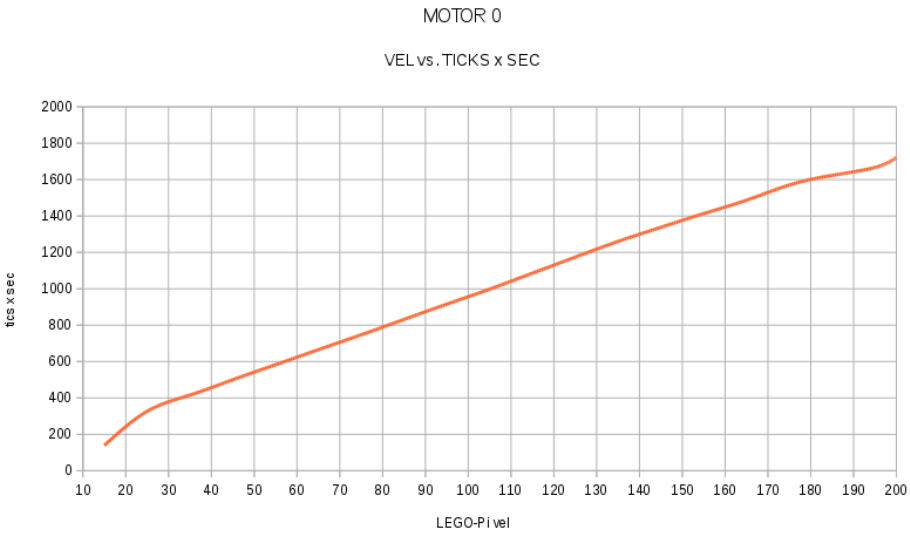


Figure 4.1: Motor 0: ticks x turn

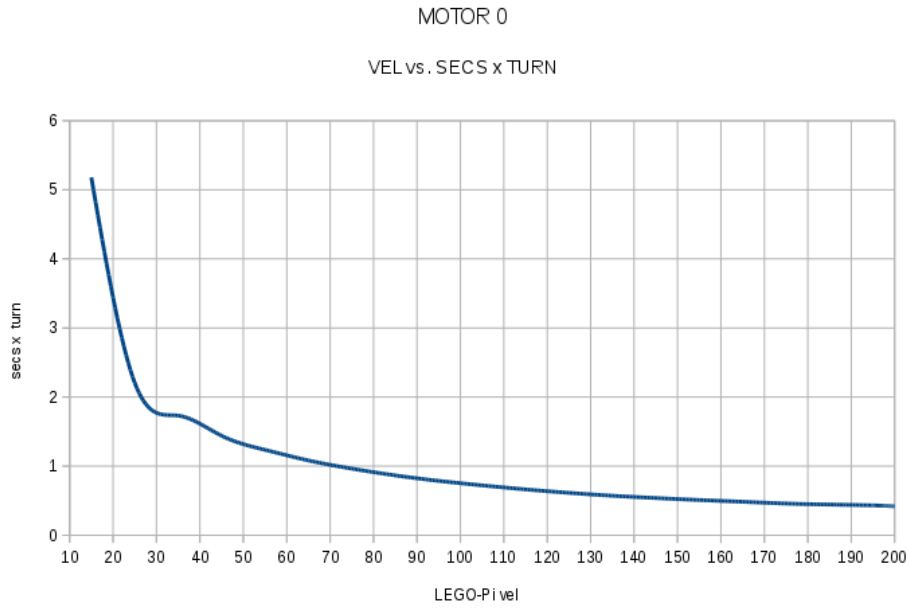


Figure 4.2: Motor 0: seconds x turn

Analogously for port 1, the values has been found with the following line

```
$> for vel in `seq 15 10 200`; do sudo ./mt_test -t5 -p1 -s10 -v"$vel" >> outs/graph1; done
```

And the graphics that the values let are:

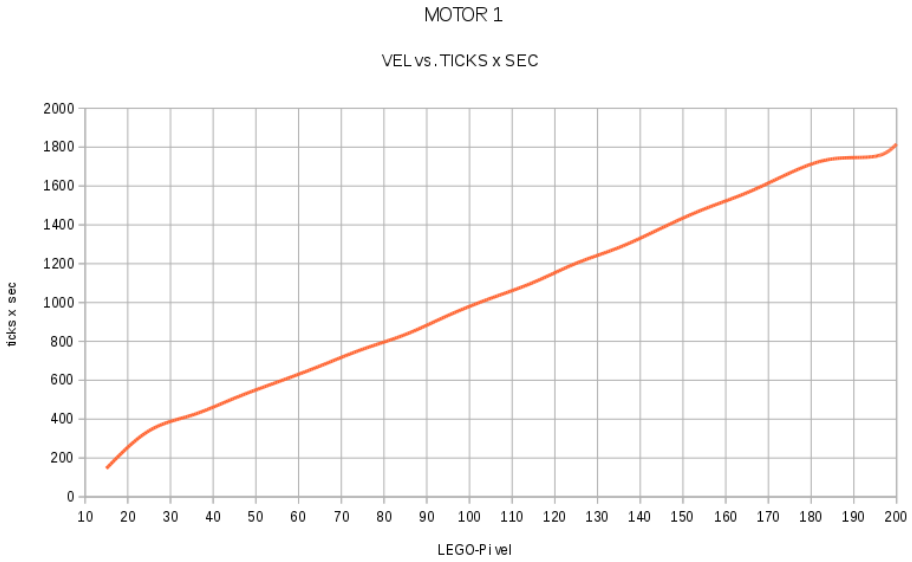


Figure 4.3: Motor 1: ticks x turn

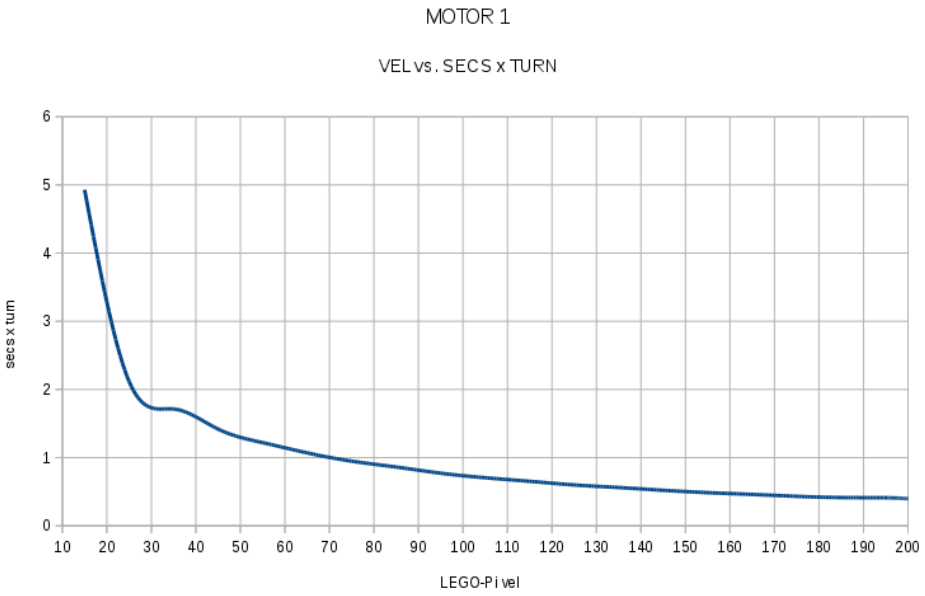


Figure 4.4: Motor 1: seconds x turn

With this values computed, I computed the average of both motors to get some kind of "generalized" graphics, ideally this ones must be computed with more than two motors readings, however, I don't have more motors, and, to compute the power we deliver to a NXT motor this values are fine enough, the generalized graphics will look like:

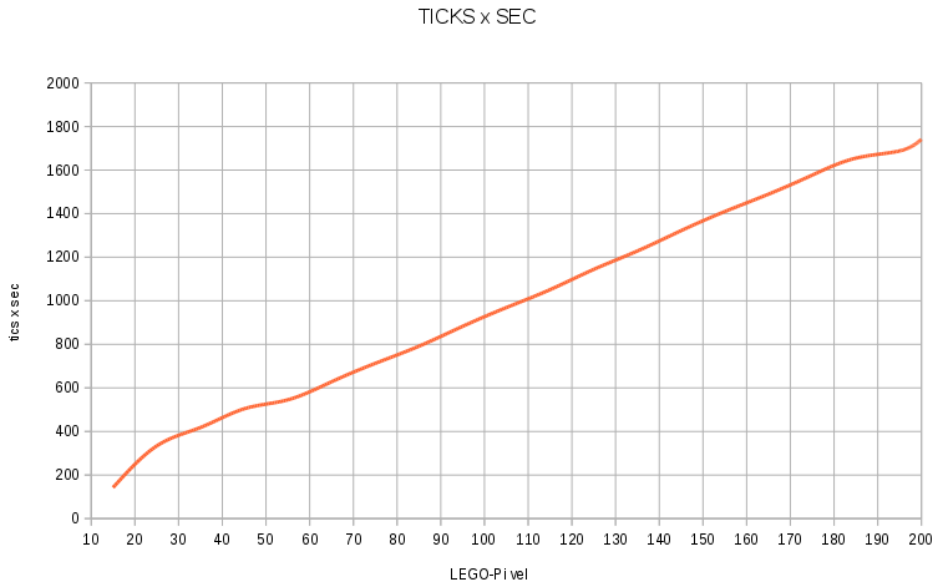


Figure 4.5: NXT Motor: ticks x turn

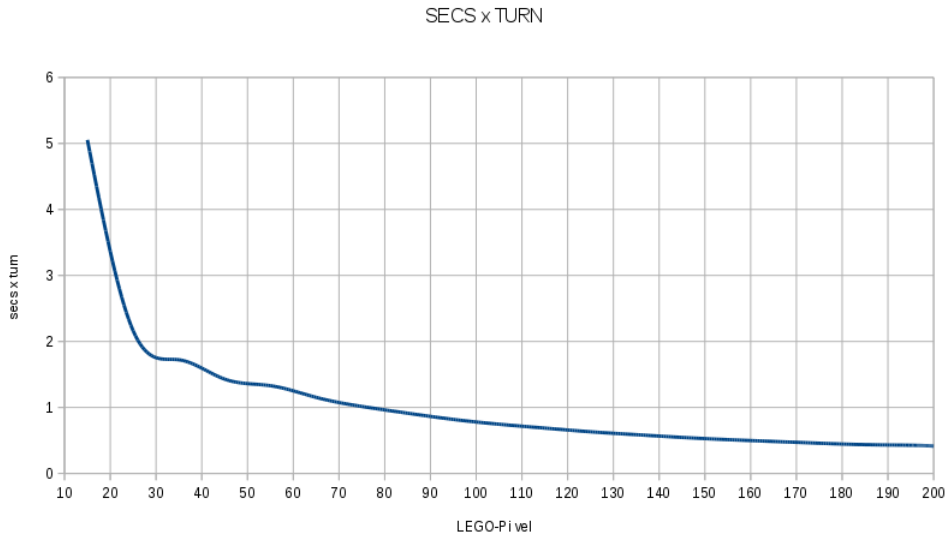


Figure 4.6: NXT Motor: seconds x turn

Now, if we reference the computed ticks per second value with the ticks per second value of the maximum velocity, the only one we generate without PWM, we can make an assumption of what percentage of the power we give to the H-Bridge we will deliver to the motor for every LEGO-Pi velocity number, since we are dealing with DC motors, it is, velocity will be proportional to the voltage applied. The following graphic will show the percentage of power delivered to the motor, this values has been computed with the ticks per second values, however it can be computed based in the seconds per turn values, the results will be the same, the important part is reference the values taken for the velocities generated via PWM and the values obtained for the max velocity, generated without PWM, the graphic of the percentage of power is:

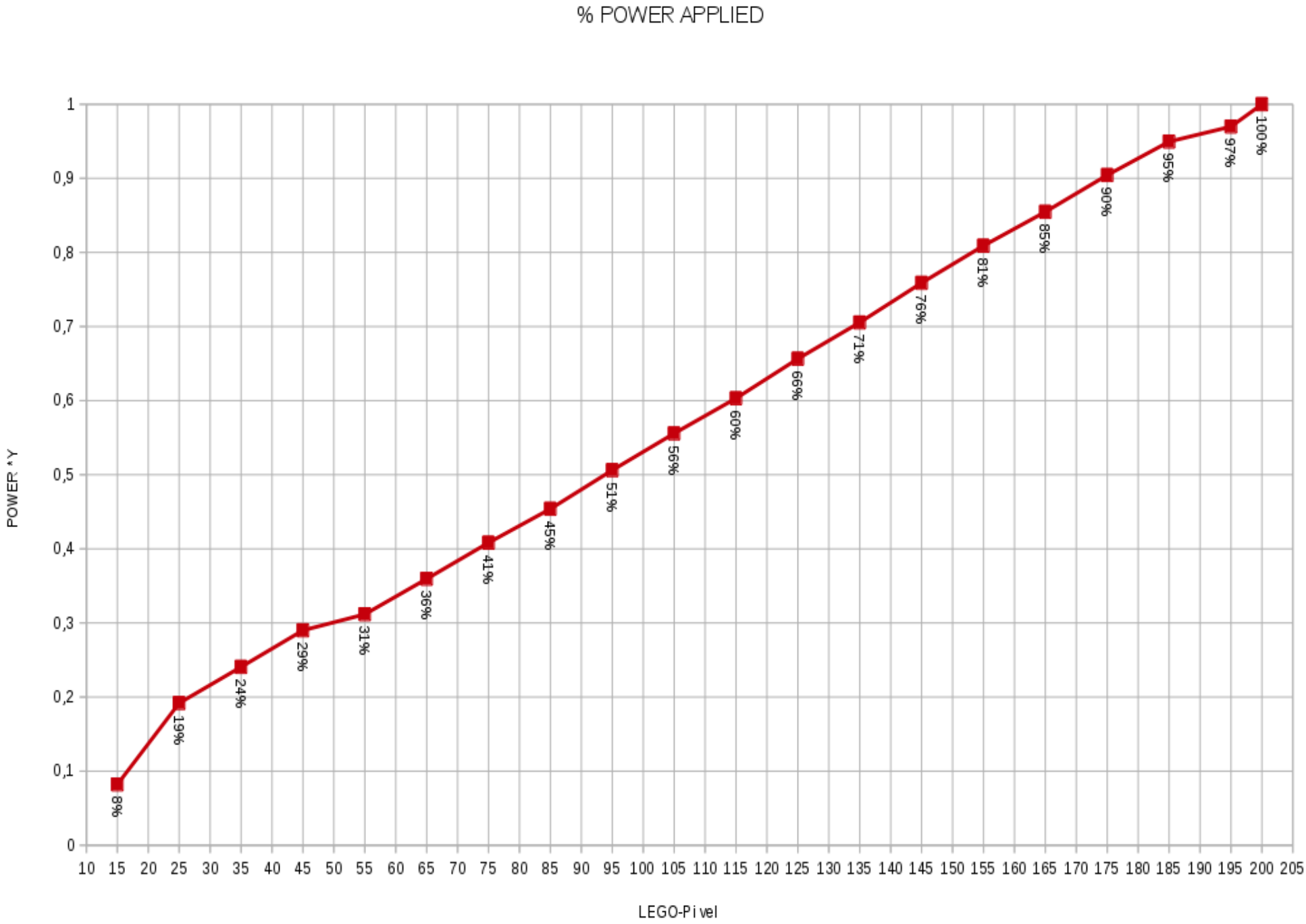


Figure 4.7: LEGO-PI vel vs. %power applied

So if you take a quick look to the latter graphic you will realize that the velocity number and the power applied aren't totally linearly related, as an example velocity 60 will be about the 33% of the power delivered to the H-bridge, as you can see the 66% of the power corresponds to the velocity number 125, which is a little bit more than the expected double (120), and it is the main reason why in the latter test, test number 2, we seems to miss some ticks. However we don't actually need a notion of "real" velocity to control our motors, it will suffice to us that the ticks per turn are stable at any velocity, what this last test proved.

Now that we know that we are receiving an stable amount of ticks for each turn independently of the velocity we can continue with test number 3, in this test we will turn a motor a desired amount of turns and we will store the delay spent between every tick, after this we will try to get some statistics of the resulting time between ticks array, which will be of the size of the ticks received. We will use the default settings for the first execution so we will rotate the motor plugged to the port 0 at a velocity of 60, it is applying the 30% of the power, for 8 turns in forward direction, in this case we will set the verbose option `-V / --verb` to 2, it will make the test print all the array stored for each encoder before the stats computed, I will paste here only the interesting parts of such a huge output, then

```
$> sudo ./mt_test -t3 -V2
```

this time the output produced for the encoder line 1 is:

```
ticks received: 2880
mean: 3152.73785
variance: 87496.77912
standard deviation: 295.79854
median: 3188.50000
quantil superior: 3325.00000
quantil inferior: 3050.00000
rmax: 3663.00000
rmin: 2849.00000
autocorrelation: 0.24701
min: 663.00000
max: 4070.00000
absolute_deviation: 163.25130


weights:
wmean: 3162.39344
wvariance: 66101.53456
wstandard_deviation: 257.10219
wabs_dev: 147.20916

smart weights:
swmean: 3347.36540
swvariance: 43567.82422
swstandard_deviation: 208.72907
swabs_dev: 190.96633

uq: 0.980, lq: 0.020, pop: 0.960, ini: 0.7, fi: 0.9
POP_PERC: 0.00000, 0.00139, 0.00174, 0.00764, 0.00799, 0.00868, 0.01042, 0.62743,
    0.33299, 0.00139,
```

And for the encoder line 2:

```
ticks received: 2880
mean: 3142.64757
variance: 106933.88235
standard deviation: 327.00747
median: 3194.00000
quantil superior: 3300.00000
quantil inferior: 3079.00000
```

```
rmax: 3900.00000
rmin: 2600.00000
autocorrelation: 0.58441
min: 712.00000
max: 6500.00000
absolute_deviation: 163.44313

weights:
wmean: 3158.32177
wvariance: 80648.89728
wstandard_deviation: 283.98749
wabs_dev: 139.97434

smart weights:
swmean: 3321.09786
swvariance: 36434.77114
swstandard_deviation: 190.87894
swabs_dev: 171.84320

uq: 0.978, lq: 0.022, pop: 0.957, ini: 0.4, fi: 0.6
POP_PERC: 0.00000, 0.00764, 0.01493, 0.01771, 0.62986, 0.32708, 0.00000, 0.00000,
    0.00000, 0.00000,
```

For each response we can see four differentiated blocks, the first one prints some stats of the raw data stored in the array for the ISR, to be honest, since we are interested in getting the more stable times for a velocity, the data will be a little bit rearranged for the test. If nonsense values are found (smallest time between ticks than the expected when turning at maximum speed) they will be substituted by the pre-computed average of the array, and the first and lasts positions of the array will be set to the average too, in order to avoid the readings of the acceleration and, if present, the breaking time. Once this little arrangements are done the test will compute the values for the first block, the `quantil superior` field stands for the 80th-percentile of the data set, it is the 80% of the population have values under the one in this field, while the `quantil inferior` stands for the 20th-percentile of the data set, it is the 80% of the population have values greater than the one in this field. The values `rmin` and `rmax` are more related with the 4th block. The `autocorrelation` is very unstable as the variance, which shows that the data is not very related to itself, which are bad news because we are turning at a, hopefully, stable velocity. The `min` and `max` values are the maximum and minimum of all the sample. The `absolute deviation` are the good news here since, after some testing, I found it is a very stable value for each velocity so we can use it as our reference to get the physical error for our pseudo P.I.D control.

The two next blocks are the same computations but applying weights to the sample taken. The first of both weights, corresponding to the results showed in the block titled `weights`, are computed taking into account the position of the array, the more centered in the array the value is the more weighted it will be, this was implemented because I found that the velocity tends to stabilize this way, actually the idea here was to totally invalidate the acceleration and break times. In the next block titled `smart weights` the values are computed sorting the data, then I take the maximum value as a reference, then we distribute the values in the array in groups depending on the value of each element, if the element value is between `j%` and `j+10%` of the maximum value it goes into the same group, being `j` a percentage ranging from 0 to 90, after this the more members the group of an element have the more weighted it will be. In the last block we can see how the population distributes related to the maximum value for a sample, in the encoder one line we can see the 96 of the population between the 70% and 90% of the maximum value while in the second encoder line we find the same 96% of the population between the 40% and 60%, as you can see the percentages of the maximum value where the population resides differs a lot for the two encoder lines, it is due to the instability of the maximum value, hence we can

not rely on it for our pseudo P.I.D control, however for the test is not bad to have this values computed.

The following values are taken from the first encoder line array of data, and as you can see, the readings can go really unstable:

```
...

3150,  3428,  3215,  3337,  3312,  3208,  3394,  3291,  3116,  3444,  3263,  3222,
3437,  3284,  3321,  3381,  3334,  3487,  3185,  3591,  3381,  3444,  3349,  3270,
3393,  3404,  3129,  3391,  3287,  3398,  3079,  3450,  3159,  3231,  3140,  3536,
3053,  3497,  3217,  3406,  3148,  3535,  3301,  3539,  3336,  3308,  3349,  3383,
3247,  3385,  3343,  3330,  3197,  3416,  3159,  3284,  3174,  3400,  3262,  3222,
3339,  3394,  3085,  3462,  3254,  3280,  3473,  3410,  3330,  3433,  3286,  3384,
3110,  3543,  3313,  3451,  3178,  3486,  3058,  3377,  3135,  3368,  3325,  3389,
3174,  3207,  3311,  3365,  3296,  3396,  3246,  3412,  3291,  3342,  3467,  3231,
3415,  3459,  1336,    26,  1704,  3431,   108,  3238,   176,  1373,  3166,    54,
2698,  1257,  1295,  2111,   217,   621,  2738,  1897,   460,  2106,  1310,  2040,
1163,  2700,  2085,    26,  1898,  1899,  3092,    27,  1769,  1425,  3229,    51,
3204,   222,  3090,  1949,    28,  1913,  1319,  3148,  1819,   181,   727,  3261,
3075,  3330,  3121,  3125,  3209,  3365,  3045,  3238,  3319,  3220,  3224,  3553,
3152,  3496,  3154,  3416,  3157,  3552,  3317,  3320,  3383,  3312,  3262,  3336,
2995,  3423,  2978,  3372,  3216,  3362,  3051,  3284,  3158,  3211,  3279,  3130,
3168,  3307,  3086,  3421,  3076,  3474,  3213,  3355,  3254,  3412,  3124,  3527,
3255,  3409,  3202,  3478,  3093,  3536,  3017,  3441,  3375,  3304,  3287,  3136,
3334,  3411,  3007,  3428,  3161,  3459,  3045,  3413,  3173,  3262,  3080,  3465,
3090,  3424,  2981,  3501,  3121,  3315,  3188,  3513,  3174,  3383,  3330,  3392,
3303,  3415,  3364,  3301,  3399,  3313,  3400,  3351,  3334,  3263,  3281,  3337,
3005,  3362,  2935,  3294,  3098,  3440,  3029,  3419,  2943,  3458,  3145,  3344

...
```

As mentioned in the section 2.2.2 a P.I.D algorithm must act at every sample taken from the encoders, however, if we want to implement P.I.D control with this values of time between ticks we need to find a solution, since we will treat the data in "real time", this is the main reason for the bizarre P.I.D control implementation of LEGO-Pi. And the things goes even wrong when velocity increments, it is, the values taken depends on the velocity, the smaller the time between ticks is the more executions of the threads responsible of the encoders lines we will have, this is our main problem, since we can lose ticks or count the ticks some time after it comes because another tick is being treated, due to the lack of power of our Raspberry Pi. In top of that, there is even another problem, we are generating the pulses via software, and the allocation time of the structures used to generate pulses is critical to P.I.D control, besides the minimum sub cycle time the software pwm library will allow us to configure is 3000µs, it is 3ms, which is equivalent to a 334 Hz frequency, and it is simply not enough, the velocity control is fair enough, however a P.I.D control will be very hard to achieve via software. The next test will perform several readings for a configurable set of velocities, however, as an instance I leave here the output of this test for a velocity of 160, it is applying the 80% of the power so we can see how unstable the time readings goes when more velocity is requested.

```
$> sudo ./mt_test -t3 -V2 -v160
```

The output is the following, for the encoder line 1:

```
ticks received: 2900
mean: 1381.65931
variance: 73300.21090
standard deviation: 270.74012
median: 1328.00000
quantil superior: 1670.00000
quantil inferior: 1117.00000
rmax: 1676.80000
```

```
rmin: 1048.00000
autocorrelation: -0.84125
min: 651.00000
max: 2096.00000
absolute_deviation: 258.18289

weights:
wmean: 1388.91433
wvariance: 72686.63386
wstandard_deviation: 269.60459
wabs_dev: 261.52019

smart weights:
swmean: 1524.37997
swvariance: 67323.00431
swstandard_deviation: 259.46677
swabs_dev: 235.14111

uq: 0.865, lq: 0.135, pop: 0.730, ini: 0.5, fi: 0.8
POP_PERC: 0.00000, 0.00000, 0.00000, 0.00793, 0.03276, 0.43931, 0.06103, 0.29103,
    0.16517, 0.00172,
```

And for the encoder line 2:

```
ticks received: 2860
mean: 1379.92168
variance: 72796.84031
standard deviation: 269.80890
median: 1348.00000
quantil superior: 1659.00000
quantil inferior: 1130.00000
rmax: 1701.60000
rmin: 1134.40000
autocorrelation: -0.77670
min: 651.00000
max: 2836.00000
absolute_deviation: 252.75067

weights:
wmean: 1390.52233
wvariance: 70321.70833
wstandard_deviation: 265.18241
wabs_dev: 254.58040

smart weights:
swmean: 1511.98429
swvariance: 62896.81654
swstandard_deviation: 250.79238
swabs_dev: 242.35990

uq: 0.859, lq: 0.141, pop: 0.719, ini: 0.4, fi: 0.6
POP_PERC: 0.00000, 0.00000, 0.01608, 0.20140, 0.32168, 0.39720, 0.06189, 0.00035,
    0.00000, 0.00070,
```

Here we can see how the population distribution is totally velocity dependent, and the autocorrelation of the data goes even worst than the first execution. The only thing that seems to remain stable after some testing is the average time between ticks and the absolute deviation, so, if we want some kind of closed loop control we must count on this values.

In any case, there is even one more thing to test here, if we saw that the main problem here was the lack of power of the board, what will happens to our values if two motors run at a time each of them with its encoders enabled, a closed loop control is meant to be independent from one motor to another. To put our board really against the wall I will run this test at a high velocity, we saw previously that in it situation we can lose some ticks or get more bad readings, lets see

```
$> sudo ./mt_test -t3 -V2 -v160 -p2
```

will produce an output like, for the motor plugged to port 0:

```
for the first encoder line:

ticks received: 2876
mean: 1261.07893
variance: 84620.33638
standard deviation: 290.89575
median: 1164.00000
quantil superior: 1617.00000
quantil inferior: 1105.00000
rmax: 1795.20000
rmin: 897.60000
autocorrelation: -0.13021
min: 651.00000
max: 2992.00000
absolute_deviation: 234.38154

weights:
wmean: 1273.74333
wvariance: 84801.98744
wstandard_deviation: 291.20781
wabs_dev: 238.75459

smart weights:
swmean: 1323.10281
swvariance: 47142.94956
swstandard_deviation: 217.12427
swabs_dev: 169.00482

uq: 0.886, lq: 0.114, pop: 0.773, ini: 0.3, fi: 0.6
POP_PERC: 0.00000, 0.00000, 0.08275, 0.49513, 0.13873, 0.27747, 0.00104, 0.00174,
    0.00035, 0.00243,

and for the second encoder line:

ticks received: 2884
mean: 1273.33356
variance: 99429.93101
standard deviation: 315.32512
median: 1163.00000
quantil superior: 1635.00000
quantil inferior: 1096.60000
rmax: 1791.60000
rmin: 895.80000
autocorrelation: -0.17294
min: 651.00000
max: 4479.00000
absolute_deviation: 252.45107

weights:
wmean: 1283.11542
wvariance: 98250.86771
wstandard_deviation: 313.44994
wabs_dev: 255.47368

smart weights:
swmean: 1524.79869
swvariance: 109420.03233
swstandard_deviation: 330.78699
swabs_dev: 308.20700

uq: 0.948, lq: 0.052, pop: 0.895, ini: 0.2, fi: 0.4
POP_PERC: 0.00000, 0.09466, 0.58426, 0.31103, 0.00451, 0.00035, 0.00485, 0.00000,
    0.00000, 0.00000,
```

And for the motor plugged to port 1:

```
for the first encoder line:

ticks received: 2846
mean: 1257.02987
variance: 116177.47011
standard deviation: 340.84816
median: 1121.00000
quantil superior: 1672.00000
quantil inferior: 1067.00000
rmax: 1732.20000
rmin: 866.10000
autocorrelation: -0.16801
min: 651.00000
max: 2887.00000
absolute_deviation: 277.31529

weights:
wmean: 1270.72801
wvariance: 116907.71812
wstandard_deviation: 341.91771
wabs_dev: 282.52858

smart weights:
swmean: 1290.22868
swvariance: 59386.49354
swstandard_deviation: 243.69344
swabs_dev: 198.57051

uq: 0.891, lq: 0.109, pop: 0.781, ini: 0.3, fi: 0.6
POP_PERC: 0.00000, 0.00000, 0.09452, 0.52178, 0.08468, 0.25931, 0.02670, 0.00351,
    0.00141, 0.00773,

and for the second encoder line:

ticks received: 2914
mean: 1244.61633
variance: 78040.06559
standard deviation: 279.35652
median: 1205.00000
quantil superior: 1546.00000
quantil inferior: 1098.00000
rmax: 0.00000
rmin: 0.00000
autocorrelation: 0.05207
min: 651.00000
max: 2941.00000
absolute_deviation: 206.29235

weights:
wmean: 1255.66830
wvariance: 76425.27020
wstandard_deviation: 276.45121
wabs_dev: 206.10045

smart weights:
swmean: 1345.87810
swvariance: 44269.64010
swstandard_deviation: 210.40352
swabs_dev: 164.65109

uq: 0.850, lq: 0.150, pop: 0.700, ini: 0.0, fi: 0.0
POP_PERC: 0.00000, 0.00000, 0.08373, 0.34454, 0.31606, 0.24537, 0.00137, 0.00172,
    0.00000, 0.00686,
```

So, as you can see, our previously reliable values, the mean, and, specially, the absolute deviation are affected when more than one motor is moving at a time, however, the values remains stable

for two motors, so we've got to take a decision here, since a closed loop control must count with the more precise values we can get, the algorithm must behave different when two motors are initialized, this is the main reason for the implementation of the function `mt_calibrate`, if only one motor is present it will automatically calibrate the motor in question, otherwise it will calibrate both motors at a time, storing then the, hopefully, more close to reality values we can get. With an array of values of time between ticks and another of absolute deviation both linked to its velocities, given that the times will be proportional to the velocities, and, after some testing, I figured out that the deviation will remain stable for each velocity but it will vary depending on the number of motors we can interpolate the values of time between ticks and absolute deviation, our notion of physical error, for any velocity we want. With a value of time between ticks and a physical error for a given velocity we can try to implement a closed loop control, based on the P.I.D algorithm, but, instead of correct the velocity at each reading of the difference between times, it will wait for a few ticks to come and will get an error notion for the travel computing the mean time between ticks for the ticks we wait for, this way we avoid bad readings, and, hopefully, the partial means will be fair with the values computed for the `mt_calibrate` function, since we turn at a allegedly stable velocity. Since the `mt_calibrate` is an expensive function, in time and computation, the library have a pre-computed values for the user to be able of use the **experimental** P.I.D control without the needing of a calibration, this values are computed for two motors, what seems the more natural scenario for a mobile robot.

Despite the efforts done the closed loop control is not actually helping the motors, and, in fact, it consumes a considerable amount of CPU, in top of that, as mentioned earlier, the software generation of the pulses is not helping either, so the control of the motors will be far from a "professional" environment motor control, however I've to say that the velocity control, it is, the pulses generated, are pretty good (even at the low frequency it works), and I've been able to make the robot move in an straight direction without much problems, and, maybe, a closed loop control running independently on each motor is too ambitious without extra hardware, however motor synchronization seems to be a more affordable task as we will see later on. For what I think is a good hardware addition to improve this behaviors you can check the section 3.4.

Now I'll carry on with test number 4, this test is actually several loops of the latter test explained, test number 3, for a set of velocities we request, when the computations are over the test will print all the statistics for our set of velocities, for each velocity we will take a few samples, each sample will be equivalent to a test 3 execution, so we have a proper data set to work with. The library only allows the user to run P.I.D with a minimum velocity of 15, since we need to interpolate the data set, the library needs both boundary velocities defined, and 15 velocity seems small enough, actually if we apply too many weight the robot will barely move at this velocity, and with too many weight I mean a simple support, the board and a few sensors with its wires. So the test will get the statistics from this minimum velocity to the max velocity (200) incrementing the velocity `step` units at each change, for each velocity the test will take `samples` samples. To set the step parameter we will use the `-s / --step` option, and to set the samples we will use the `-S / --samples` option, each sample will be of `turns` turns which defaults to 8, and it will be fine for this test, however you can always use the `-r / --turns` option to set the samples the size you like. Well, lets execute the test then, this time to avoid a huge output we will leave the verbose mode in its default, to 0, so we only get the results of the test itself, we will take 10 samples for each velocity and we will increment the velocity 10 units every step, additionally we will run the test for both motors at the same time and with both encoder lines enabled since we are interested in the values when the board is the more stressed:

4.1. Motor testing

```
$> sudo ./mt_test -t4 -s10 -S10 -p2
```

Has produced the following output:

```
MOTOR 0, STEP: 10, SAMPLES 10 x step:

                    ENC 1:
                    AVERAGE         VARIANCE        STND_DEV        ABS_DEV

vel_15_average:     7128.34633,     2770.89996,     52.63934,       39.15372,
vel_15_variance:    1143756.48237,  7687720883.272, 87679.64920,    73267.30011,
vel_15_std_dev:     1068.76371,     1667.34451,     40.83313,       34.23279,
vel_15_abs_dev:     732.37239,      222.50782,      14.91670,       11.46760,
vel_15_acorr:       -0.03372,       0.00163,        0.04036,        0.03393,
vel_15_wavg:        7106.05823,     3859.21250,     62.12256,       41.45285,
vel_15_wvari:       1086138.82206,  6421694505.357, 80135.47595,    67075.34290,
vel_15_w_std_dev:   1041.54131,     1478.35045,     38.44932,       32.05774,
vel_15_w_abs_dev:   725.74405,      177.51129,      13.32334,       9.87707,
vel_15_sw_avg:      7668.07511,     6105.51641,     78.13780,       54.81366,
vel_15_sw_var:      637243.78108,   723190958.673,  26892.21000,    19559.81035,
vel_15_sw_std_dev:  798.11709,      280.99305,      16.76285,       12.19660,
vel_15_sw_abs_dev:  676.52436,      1433.74810,     37.86487,       22.51914,
vel_15_upperq:      7866.32000,     4253.01511,     65.21514,       38.44800,
vel_15_lowerq:      6515.80000,     6926.87111,     83.22783,       70.64000,
vel_15_rmax:        6515.80000,     6926.87111,     83.22783,       70.64000,
vel_15_rmin:        6515.80000,     6926.87111,     83.22783,       70.64000,
vel_15_0%-10%:      0.00000,        0.00000,        0.00000,        0.00000,
vel_15_10%-20%:     0.00306,        0.00008,        0.00883,        0.00501,
vel_15_20%-30%:     0.05512,        0.00517,        0.07190,        0.04878,
vel_15_30%-40%:     0.22587,        0.03925,        0.19811,        0.14243,
vel_15_40%-50%:     0.46283,        0.02802,        0.16741,        0.09666,
vel_15_50%-60%:     0.24482,        0.01816,        0.13477,        0.10092,
vel_15_60%-70%:     0.00382,        0.00001,        0.00255,        0.00202,
vel_15_70%-80%:     0.00139,        0.00000,        0.00148,        0.00125,
vel_15_80%-90%:     0.00073,        0.00000,        0.00084,        0.00067,
vel_15_90%-100%:    0.00170,        0.00000,        0.00121,        0.00088,
pop_avg_sum:        0.99934

vel_25_average:     5012.48803,     205.18219,      14.32418,       11.16924,
vel_25_variance:    742088.39924,   3834594852.343, 61924.10558,    49199.21035,
vel_25_std_dev:     860.75941,      1312.93384,     36.23443,       28.66529,
vel_25_abs_dev:     605.44808,      535.17219,      23.13379,       19.57973,
vel_25_acorr:       -0.00266,       0.00023,        0.01512,        0.01234,
vel_25_wavg:        5002.94407,     201.65040,      14.20037,       10.97869,
vel_25_wvari:       736382.51754,   3733417575.331, 61101.69863,    48055.69082,
vel_25_w_std_dev:   857.44588,      1298.97907,     36.04135,       28.15058,
vel_25_w_abs_dev:   610.02219,      491.19230,      22.16286,       18.61246,
vel_25_sw_avg:      5384.94694,     4453.94215,     66.73786,       51.24191,
vel_25_sw_var:      324169.44287,   1250877888.852, 35367.75210,    24047.87926,
vel_25_sw_std_dev:  568.55249,      1019.44884,     31.92881,       21.35328,
vel_25_sw_abs_dev:  500.87603,      1220.55925,     34.93650,       24.15261,
vel_25_upperq:      5534.00000,     47.12889,       6.86505,        5.44000,
vel_25_lowerq:      4446.12000,     7003.54844,     83.68721,       68.33600,
vel_25_rmax:        4446.12000,     7003.54844,     83.68721,       68.33600,
vel_25_rmin:        4446.12000,     7003.54844,     83.68721,       68.33600,
vel_25_0%-10%:      0.00226,        0.00000,        0.00115,        0.00073,
vel_25_10%-20%:     0.00111,        0.00000,        0.00103,        0.00078,
vel_25_20%-30%:     0.06532,        0.00094,        0.03072,        0.02174,
vel_25_30%-40%:     0.14134,        0.00041,        0.02018,        0.01286,
vel_25_40%-50%:     0.58786,        0.00803,        0.08960,        0.06959,
vel_25_50%-60%:     0.19531,        0.01476,        0.12150,        0.09985,
vel_25_60%-70%:     0.00209,        0.00000,        0.00077,        0.00056,
vel_25_70%-80%:     0.00097,        0.00000,        0.00043,        0.00035,
vel_25_80%-90%:     0.00097,        0.00000,        0.00112,        0.00088,
vel_25_90%-100%:    0.00202,        0.00000,        0.00132,        0.00110,
pop_avg_sum:        0.99927

vel_35_average:     4153.98197,     11636.22996,    107.87136,      59.46641,
```

4.1. Motor testing

```
vel_35_variance:        595286.20150,   2168023014.314, 46562.03404,    34647.19820,
vel_35_std_dev:         771.03891,      872.44352,      29.53715,       22.15540,
vel_35_abs_dev:         559.96105,      409.34938,      20.23238,       16.09309,
vel_35_acorr:           0.03532,        0.00092,        0.03026,        0.02699,
vel_35_wavg:            4146.66808,     10830.42467,    104.06933,      57.14898,
vel_35_wvari:           598596.56712,   1755013447.336, 41892.88063,    33368.81909,
vel_35_w_std_dev:       773.27829,      708.06116,      26.60942,       21.23901,
vel_35_w_abs_dev:       566.77853,      297.00435,      17.23381,       14.26889,
vel_35_sw_avg:          4498.18040,     11864.38599,    108.92376,      59.11876,
vel_35_sw_var:          262941.00533,   414704942.912,  20364.30561,    17418.22435,
vel_35_sw_std_dev:      512.42910,      397.13845,      19.92833,       17.01560,
vel_35_sw_abs_dev:      459.43520,      244.88854,      15.64892,       12.41467,
vel_35_upperq:          4602.20000,     9993.06667,     99.96533,       55.56000,
vel_35_lowerq:          3538.46000,     30333.74267,    174.16585,      121.26000,
vel_35_rmax:            3538.46000,     30333.74267,    174.16585,      121.26000,
vel_35_rmin:            3538.46000,     30333.74267,    174.16585,      121.26000,
vel_35_0%-10%:          0.00017,        0.00000,        0.00025,        0.00021,
vel_35_10%-20%:         0.01212,        0.00002,        0.00442,        0.00306,
vel_35_20%-30%:         0.05596,        0.00013,        0.01123,        0.00885,
vel_35_30%-40%:         0.14729,        0.00003,        0.00575,        0.00400,
vel_35_40%-50%:         0.59047,        0.00694,        0.08333,        0.07002,
vel_35_50%-60%:         0.18733,        0.00861,        0.09278,        0.07952,
vel_35_60%-70%:         0.00174,        0.00000,        0.00049,        0.00042,
vel_35_70%-80%:         0.00038,        0.00000,        0.00035,        0.00026,
vel_35_80%-90%:         0.00038,        0.00000,        0.00042,        0.00033,
vel_35_90%-100%:        0.00300,        0.00000,        0.00143,        0.00105,
pop_avg_sum:            0.99885

vel_45_average:         3504.90856,     205.73053,      14.34331,       12.03678,
vel_45_variance:        475559.96215,   375647037.519,  19381.61597,    15129.83031,
vel_45_std_dev:         689.47734,      201.06707,      14.17981,       11.01217,
vel_45_abs_dev:         495.38871,      152.69097,      12.35682,       10.53446,
vel_45_acorr:           0.02115,        0.00032,        0.01783,        0.01391,
vel_45_wavg:            3500.20230,     215.18844,      14.66930,       11.84559,
vel_45_wvari:           474796.43758,   834443801.790,  28886.74093,    22624.12402,
vel_45_w_std_dev:       688.76064,      450.24674,      21.21902,       16.56826,
vel_45_w_abs_dev:       497.80118,      222.62785,      14.92072,       11.34765,
vel_45_sw_avg:          3781.08211,     3158.12471,     56.19719,       43.28031,
vel_45_sw_var:          199793.59285,   156474286.217,  12508.96823,    9746.62259,
vel_45_sw_std_dev:      446.78734,      194.07075,      13.93093,       10.88626,
vel_45_sw_abs_dev:      394.81660,      285.63464,      16.90073,       13.10696,
vel_45_upperq:          3888.64000,     35.95378,       5.99615,        4.32000,
vel_45_lowerq:          2966.62000,     4536.55511,     67.35395,       55.82000,
vel_45_rmax:            2966.62000,     4536.55511,     67.35395,       55.82000,
vel_45_rmin:            2966.62000,     4536.55511,     67.35395,       55.82000,
vel_45_0%-10%:          0.00153,        0.00000,        0.00085,        0.00058,
vel_45_10%-20%:         0.01671,        0.00001,        0.00265,        0.00201,
vel_45_20%-30%:         0.07263,        0.00130,        0.03603,        0.02615,
vel_45_30%-40%:         0.13346,        0.00047,        0.02176,        0.01648,
vel_45_40%-50%:         0.62984,        0.00530,        0.07280,        0.06132,
vel_45_50%-60%:         0.13837,        0.00732,        0.08554,        0.07148,
vel_45_60%-70%:         0.00174,        0.00000,        0.00109,        0.00091,
vel_45_70%-80%:         0.00066,        0.00000,        0.00064,        0.00046,
vel_45_80%-90%:         0.00080,        0.00000,        0.00128,        0.00098,
vel_45_90%-100%:        0.00293,        0.00000,        0.00185,        0.00151,
pop_avg_sum:            0.99868

vel_55_average:         3064.36879,     96.33487,       9.81503,        7.75052,
vel_55_variance:        348027.94599,   368302259.195,  19191.20265,    14987.56640,
vel_55_std_dev:         589.73471,      267.67990,      16.36093,       12.82590,
vel_55_abs_dev:         413.19298,      76.85994,       8.76698,        7.12461,
vel_55_acorr:           -0.01102,       0.00066,        0.02577,        0.02117,
vel_55_wavg:            3060.32264,     115.21386,      10.73377,       7.88850,
vel_55_wvari:           346937.42704,   484423502.667,  22009.62296,    18075.37485,
vel_55_w_std_dev:       588.74294,      354.64807,      18.83210,       15.46390,
vel_55_w_abs_dev:       414.59889,      95.69180,       9.78222,        7.60891,
vel_55_sw_avg:          3325.89720,     86.20908,       9.28488,        7.32092,
vel_55_sw_var:          153964.01204,   82279422.260,   9070.80053,     6597.90611,
```

vel_55_sw_std_dev:	392.23102,	132.04137,	11.49093,	8.36525,
vel_55_sw_abs_dev:	348.02420,	85.20068,	9.23042,	7.12221,
vel_55_upperq:	3396.06000,	26.19600,	5.11820,	4.04800,
vel_55_lowerq:	2677.16000,	2644.52267,	51.42492,	40.08000,
vel_55_rmax:	2677.16000,	2644.52267,	51.42492,	40.08000,
vel_55_rmin:	2677.16000,	2644.52267,	51.42492,	40.08000,
vel_55_0%-10%:	0.00063,	0.00000,	0.00049,	0.00042,
vel_55_10%-20%:	0.01163,	0.00000,	0.00160,	0.00121,
vel_55_20%-30%:	0.06134,	0.00008,	0.00914,	0.00774,
vel_55_30%-40%:	0.13047,	0.00008,	0.00873,	0.00707,
vel_55_40%-50%:	0.60397,	0.00300,	0.05477,	0.04829,
vel_55_50%-60%:	0.18295,	0.00306,	0.05532,	0.04758,
vel_55_60%-70%:	0.00212,	0.00000,	0.00070,	0.00045,
vel_55_70%-80%:	0.00080,	0.00000,	0.00029,	0.00024,
vel_55_80%-90%:	0.00045,	0.00000,	0.00037,	0.00029,
vel_55_90%-100%:	0.00393,	0.00000,	0.00121,	0.00101,
pop_avg_sum:	0.99829			
vel_65_average:	2707.87651,	81.65393,	9.03626,	6.85568,
vel_65_variance:	263461.48591,	266290026.996,	16318.39536,	12506.63050,
vel_65_std_dev:	513.06571,	250.07305,	15.81370,	12.06270,
vel_65_abs_dev:	359.08576,	51.56913,	7.18116,	5.51623,
vel_65_acorr:	-0.01744,	0.00046,	0.02135,	0.01740,
vel_65_wavg:	2703.13596,	81.58601,	9.03250,	6.64895,
vel_65_wvari:	261770.68463,	401727736.969,	20043.14688,	15589.55606,
vel_65_w_std_dev:	511.30327,	377.39252,	19.42659,	15.11700,
vel_65_w_abs_dev:	359.55917,	74.08860,	8.60747,	5.69927,
vel_65_sw_avg:	2942.98972,	805.71671,	28.38515,	16.99734,
vel_65_sw_var:	129943.39281,	69632077.381,	8344.58372,	5564.78910,
vel_65_sw_std_dev:	360.30367,	138.51152,	11.76909,	7.76030,
vel_65_sw_abs_dev:	315.23914,	443.75481,	21.06549,	11.83361,
vel_65_upperq:	3021.12000,	13.64622,	3.69408,	2.92000,
vel_65_lowerq:	2398.80000,	3564.34667,	59.70215,	47.28000,
vel_65_rmax:	2398.80000,	3564.34667,	59.70215,	47.28000,
vel_65_rmin:	2398.80000,	3564.34667,	59.70215,	47.28000,
vel_65_0%-10%:	0.00010,	0.00000,	0.00033,	0.00019,
vel_65_10%-20%:	0.01205,	0.00032,	0.01785,	0.01013,
vel_65_20%-30%:	0.07846,	0.00097,	0.03122,	0.01734,
vel_65_30%-40%:	0.18945,	0.04047,	0.20116,	0.11445,
vel_65_40%-50%:	0.50866,	0.03351,	0.18306,	0.10485,
vel_65_50%-60%:	0.20287,	0.00716,	0.08463,	0.06118,
vel_65_60%-70%:	0.00275,	0.00000,	0.00106,	0.00073,
vel_65_70%-80%:	0.00080,	0.00000,	0.00055,	0.00040,
vel_65_80%-90%:	0.00028,	0.00000,	0.00036,	0.00028,
vel_65_90%-100%:	0.00344,	0.00000,	0.00164,	0.00116,
pop_avg_sum:	0.99885			
vel_75_average:	2424.04946,	84.12771,	9.17212,	6.91863,
vel_75_variance:	227131.46068,	126864125.832,	11263.39761,	8572.81042,
vel_75_std_dev:	476.45154,	139.32137,	11.80345,	9.02316,
vel_75_abs_dev:	326.90731,	38.27507,	6.18669,	5.21331,
vel_75_acorr:	-0.01491,	0.00076,	0.02752,	0.02337,
vel_75_wavg:	2418.93092,	92.40175,	9.61258,	7.60041,
vel_75_wvari:	226136.56957,	164039783.177,	12807.80165,	10224.79464,
vel_75_w_std_dev:	475.36813,	179.68234,	13.40456,	10.73730,
vel_75_w_abs_dev:	327.41186,	31.05965,	5.57312,	4.42361,
vel_75_sw_avg:	2667.49688,	138.50656,	11.76888,	8.35378,
vel_75_sw_var:	122832.43844	186658214.775,	13662.29171,	9922.44394,
vel_75_sw_std_dev:	349.95970,	400.72065,	20.01801,	14.44894,
vel_75_sw_abs_dev:	310.13248,	439.95288,	20.97505,	14.79748,
vel_75_upperq:	2741.86000,	13.48489,	3.67218,	2.63200,
vel_75_lowerq:	2238.86000,	2256.32933,	47.50084,	33.08800,
vel_75_rmax:	2238.86000,	2256.32933,	47.50084,	33.08800,
vel_75_rmin:	2238.86000,	2256.32933,	47.50084,	33.08800,
vel_75_0%-10%:	0.00000,	0.00000,	0.00000,	0.00000,
vel_75_10%-20%:	0.01051,	0.00002,	0.00421,	0.00324,
vel_75_20%-30%:	0.09273,	0.00106,	0.03252,	0.02438,
vel_75_30%-40%:	0.17692,	0.02415,	0.15542,	0.11679,

vel_75_40%-50%:	0.51880,	0.00928,	0.09632,	0.06914,
vel_75_50%-60%:	0.19105,	0.01067,	0.10327,	0.07612,
vel_75_60%-70%:	0.00247,	0.00000,	0.00104,	0.00066,
vel_75_70%-80%:	0.00157,	0.00000,	0.00150,	0.00091,
vel_75_80%-90%:	0.00059,	0.00000,	0.00116,	0.00067,
vel_75_90%-100%:	0.00341,	0.00000,	0.00216,	0.00162,
pop_avg_sum:	0.99805			
vel_85_average:	2164.91037,	727.15389,	26.96579,	22.99422,
vel_85_variance:	223167.92388,	598834277.731,	24471.09065,	19099.38339,
vel_85_std_dev:	471.75361,	684.95156,	26.17158,	20.18551,
vel_85_abs_dev:	344.26568,	331.35056,	18.20304,	14.46023,
vel_85_acorr:	-0.06545,	0.00873,	0.09345,	0.07889,
vel_85_wavg:	2158.19151,	860.86452,	29.34049,	24.91114,
vel_85_wvari:	221703.47771,	583858851.089,	24163.17138,	18324.99528,
vel_85_w_std_dev:	470.21057,	672.77271,	25.93786,	19.44023,
vel_85_w_abs_dev:	346.25610,	366.35252,	19.14034,	15.31536,
vel_85_sw_avg:	2420.26670,	963.44260,	31.03937,	17.76418,
vel_85_sw_var:	142978.29584,	338163053.412,	18389.21024,	14089.52793,
vel_85_sw_std_dev:	377.43852,	576.06775,	24.00141,	18.58412,
vel_85_sw_abs_dev:	335.63678,	643.01743,	25.35779,	20.80397,
vel_85_upperq:	2530.08000,	66.99733,	8.18519,	7.49600,
vel_85_lowerq:	1823.36000,	9475.18933,	97.34058,	87.08000,
vel_85_rmax:	1823.36000,	9475.18933,	97.34058,	87.08000,
vel_85_rmin:	1823.36000,	9475.18933,	97.34058,	87.08000,
vel_85_0%-10%:	0.00000,	0.00000,	0.00000,	0.00000,
vel_85_10%-20%:	0.03070,	0.00096,	0.03097,	0.02330,
vel_85_20%-30%:	0.09785,	0.00037,	0.01917,	0.01606,
vel_85_30%-40%:	0.16585,	0.02703,	0.16441,	0.10414,
vel_85_40%-50%:	0.44505,	0.01298,	0.11395,	0.06608,
vel_85_50%-60%:	0.25312,	0.02101,	0.14496,	0.11849,
vel_85_60%-70%:	0.00286,	0.00000,	0.00153,	0.00118,
vel_85_70%-80%:	0.00136,	0.00000,	0.00129,	0.00086,
vel_85_80%-90%:	0.00038,	0.00000,	0.00031,	0.00025,
vel_85_90%-100%:	0.00217,	0.00001,	0.00334,	0.00241,
pop_avg_sum:	0.99934			
vel_95_average:	2002.44730,	25.88669,	5.08790,	3.53766,
vel_95_variance:	186860.05376,	43267726.663,	6577.82081,	4841.52078,
vel_95_std_dev:	432.21410,	56.69926,	7.52989,	5.54856,
vel_95_abs_dev:	302.57762,	20.30255,	4.50584,	3.32151,
vel_95_acorr:	-0.09244,	0.00032,	0.01798,	0.01343,
vel_95_wavg:	1996.14600,	42.49859,	6.51909,	5.21635,
vel_95_wvari:	182826.96647,	61218581.382,	7824.23040,	6363.66846,
vel_95_w_std_dev:	427.49626,	82.12460,	9.06226,	7.39392,
vel_95_w_abs_dev:	300.37413,	24.26855,	4.92631,	3.76501,
vel_95_sw_avg:	2217.11370,	545.02620,	23.34580,	19.24069,
vel_95_sw_var:	111413.45531,	40592413.654,	6371.21760,	4392.16679,
vel_95_sw_std_dev:	333.66666,	88.90796,	9.42910,	6.52615,
vel_95_sw_abs_dev:	288.21665,	217.41920,	14.74514,	11.88059,
vel_95_upperq:	2336.70000,	1.78889,	1.33749,	1.10000,
vel_95_lowerq:	1784.26000,	760.38267,	27.57504,	20.16800,
vel_95_rmax:	1784.26000,	760.38267,	27.57504,	20.16800,
vel_95_rmin:	1784.26000,	760.38267,	27.57504,	20.16800,
vel_95_0%-10%:	0.00000,	0.00000,	0.00000,	0.00000,
vel_95_10%-20%:	0.01670,	0.00024,	0.01538,	0.01006,
vel_95_20%-30%:	0.09625,	0.00057,	0.02379,	0.01746,
vel_95_30%-40%:	0.21570,	0.03022,	0.17385,	0.12661,
vel_95_40%-50%:	0.46722,	0.01702,	0.13046,	0.08813,
vel_95_50%-60%:	0.19231,	0.02209,	0.14861,	0.13734,
vel_95_60%-70%:	0.00321,	0.00000,	0.00130,	0.00105,
vel_95_70%-80%:	0.00209,	0.00000,	0.00109,	0.00077,
vel_95_80%-90%:	0.00153,	0.00000,	0.00124,	0.00111,
vel_95_90%-100%:	0.00398,	0.00001,	0.00246,	0.00195,
pop_avg_sum:	0.99899			
vel_105_average:	1855.15125,	80.16171,	8.95331,	6.98461,
vel_105_variance:	164994.33008,	84874708.109,	9212.74704,	7151.98472,

```
vel_105_std_dev:        406.05313,      127.98386,      11.31299,       8.79504,
vel_105_abs_dev:        291.34870,      19.11048,       4.37155,        3.47824,
vel_105_acorr:          -0.13588,       0.00054,        0.02319,        0.01816,
vel_105_wavg:           1849.26282,     83.76833,       9.15250,        7.36844,
vel_105_wvari:          160986.67238,   119990952.834,  10954.03820,    7787.69911,
vel_105_w_std_dev:      401.02545,      183.62650,      13.55089,       9.63575,
vel_105_w_abs_dev:      288.93127,      28.68378,       5.35572,        4.24417,
vel_105_sw_avg:         2073.21092,     1088.63360,     32.99445,       18.85335,
vel_105_sw_var:         107007.30852,   107269022.815,  10357.07598,    6808.40374,
vel_105_sw_std_dev:     326.75284,      266.54553,      16.32622,       10.46783,
vel_105_sw_abs_dev:     288.85956,      525.60232,      22.92602,       13.29700,
vel_105_upperq:         2189.70000,     37.34444,       6.11101,        4.90000,
vel_105_lowerq:         1657.72000,     147.92178,      12.16231,       8.21600,
vel_105_rmax:           1657.72000,     147.92178,      12.16231,       8.21600,
vel_105_rmin:           1657.72000,     147.92178,      12.16231,       8.21600,
vel_105_0%-10%:         0.00003,        0.00000,        0.00011,        0.00006,
vel_105_10%-20%:        0.04324,        0.00157,        0.03963,        0.03559,
vel_105_20%-30%:        0.18758,        0.03089,        0.17575,        0.13337,
vel_105_30%-40%:        0.31027,        0.05236,        0.22883,        0.19905,
vel_105_40%-50%:        0.27514,        0.04076,        0.20189,        0.17717,
vel_105_50%-60%:        0.17452,        0.03267,        0.18075,        0.17145,
vel_105_60%-70%:        0.00362,        0.00000,        0.00201,        0.00174,
vel_105_70%-80%:        0.00223,        0.00000,        0.00173,        0.00125,
vel_105_80%-90%:        0.00052,        0.00000,        0.00062,        0.00056,
vel_105_90%-100%:       0.00244,        0.00001,        0.00287,        0.00244,
pop_avg_sum:            0.99958

vel_115_average:        1714.32786,     44.00726,       6.63380,        5.26914,
vel_115_variance:       147927.63644,   27790128.873,   5271.63436,     4154.31723,
vel_115_std_dev:        384.55797,      47.55899,       6.89630,        5.42768,
vel_115_abs_dev:        285.50934,      14.93140,       3.86412,        3.15941,
vel_115_acorr:          -0.17702,       0.00052,        0.02274,        0.01845,
vel_115_wavg:           1709.26064,     42.61574,       6.52807,        5.36773,
vel_115_wvari:          145256.95853,   27064253.175,   5202.33151,     4094.07880,
vel_115_w_std_dev:      381.07061,      46.82967,       6.84322,        5.37690,
vel_115_w_abs_dev:      283.34707,      15.73605,       3.96687,        3.11500,
vel_115_sw_avg:         1917.45091,     301.00009,      17.34935,       12.51309,
vel_115_sw_var:         97972.31635,    22577350.620,   4751.56297,     3313.84189,
vel_115_sw_std_dev:     312.92140,      58.35081,       7.63877,        5.32804,
vel_115_sw_abs_dev:     278.35436,      155.80868,      12.48233,       8.69456,
vel_115_upperq:         2049.52000,     10.11733,       3.18078,        2.04800,
vel_115_lowerq:         1513.20000,     72.16000,       8.49470,        6.48000,
vel_115_rmax:           1513.20000,     72.16000,       8.49470,        6.48000,
vel_115_rmin:           1513.20000,     72.16000,       8.49470,        6.48000,
vel_115_0%-10%:         0.00000,        0.00000,        0.00000,        0.00000,
vel_115_10%-20%:        0.01419,        0.00048,        0.02185,        0.01374,
vel_115_20%-30%:        0.09472,        0.00170,        0.04119,        0.02328,
vel_115_30%-40%:        0.24572,        0.03879,        0.19694,        0.16550,
vel_115_40%-50%:        0.37956,        0.01280,        0.11316,        0.06332,
vel_115_50%-60%:        0.25297,        0.02469,        0.15715,        0.13627,
vel_115_60%-70%:        0.00407,        0.00000,        0.00068,        0.00049,
vel_115_70%-80%:        0.00243,        0.00000,        0.00079,        0.00056,
vel_115_80%-90%:        0.00202,        0.00000,        0.00216,        0.00156,
vel_115_90%-100%:       0.00372,        0.00001,        0.00265,        0.00221,
pop_avg_sum:            0.99941

vel_125_average:        1599.03547,     99.47585,       9.97376,        7.63516,
vel_125_variance:       140167.59007,   83215487.552,   9122.25233,     7329.77453,
vel_125_std_dev:        374.21428,      145.84547,      12.07665,       9.67797,
vel_125_abs_dev:        284.76351,      32.02632,       5.65918,        4.48140,
vel_125_acorr:          -0.21946,       0.00074,        0.02724,        0.02008,
vel_125_wavg:           1592.89487,     66.77488,       8.17159,        6.07879,
vel_125_wvari:          136931.71536,   63145623.66764, 7946.42207,     5776.58361,
vel_125_w_std_dev:      369.90602,      112.50446,      10.60681,       7.71190,
vel_125_w_abs_dev:      282.16393,      21.77271,       4.66612,        3.66254,
vel_125_sw_avg:         1783.87774,     719.01133,      26.81439,       18.49984,
vel_125_sw_var:         92759.62102,    75080005.846,   8664.87195,     6472.11105,
vel_125_sw_std_dev:     304.25394,      210.18087,      14.49762,       10.74350,
```

```
vel_125_sw_abs_dev:     268.86050,      446.25992,      21.12486,       15.62313,
vel_125_upperq:         1937.90000,     29.65556,       5.44569,        4.08000,
vel_125_lowerq:         1389.80000,     116.76444,      10.80576,       6.84000,
vel_125_rmax:           1445.10000,     29739.131,      172.45037,      97.98000,
vel_125_rmin:           1389.80000,     116.76444,      10.80576,       6.84000,
vel_125_0%-10%:         0.00000,        0.00000,        0.00000,        0.00000,
vel_125_10%-20%:        0.02145,        0.00083,        0.02887,        0.02210,
vel_125_20%-30%:        0.11461,        0.00716,        0.08462,        0.05743,
vel_125_30%-40%:        0.28073,        0.03045,        0.17450,        0.15636,
vel_125_40%-50%:        0.32456,        0.01484,        0.12181,        0.08981,
vel_125_50%-60%:        0.24511,        0.02463,        0.15693,        0.13413,
vel_125_60%-70%:        0.00466,        0.00001,        0.00257,        0.00184,
vel_125_70%-80%:        0.00289,        0.00000,        0.00151,        0.00115,
vel_125_80%-90%:        0.00167,        0.00000,        0.00195,        0.00124,
vel_125_90%-100%:       0.00369,        0.00001,        0.00299,        0.00212,
pop_avg_sum:            0.99937

vel_135_average:        1503.41583,     91.82639,       9.58261,        7.91649,
vel_135_variance:       128652.20107,   24532878.993,   4953.06763,     3505.88630,
vel_135_std_dev:        358.62246,      46.81537,       6.84218,        4.84210,
vel_135_abs_dev:        280.87318,      18.90447,       4.34793,        3.27827,
vel_135_acorr:          -0.25592,       0.00070,        0.02640,        0.01974,
vel_135_wavg:           1497.25513,     108.92233,      10.43659,       8.71370,
vel_135_wvari:          125728.17161,   20338282.181,   4509.79846,     3249.12636,
vel_135_w_std_dev:      354.53084,      40.06432,       6.32964,        4.56291,
vel_135_w_abs_dev:      278.01879,      15.80275,       3.97527,        2.80638,
vel_135_sw_avg:         1651.34703,     179.88190,      13.41201,       10.15606,
vel_135_sw_var:         81285.16358,    24154673.285,   4914.74041,     3993.80446,
vel_135_sw_std_dev:     284.98550,      76.03363,       8.71973,        7.09333,
vel_135_sw_abs_dev:     249.78101,      153.23922,      12.37898,       10.06855,
vel_135_upperq:         1838.00000,     31.07556,       5.57455,        4.00000,
vel_135_lowerq:         1286.96000,     95.06489,       9.75012,        7.16000,
vel_135_rmax:           1562.44000,     84294.06933,    290.33441,      275.36000,
vel_135_rmin:           1286.96000,     95.06489,       9.75012,        7.16000,
vel_135_0%-10%:         0.00000,        0.00000,        0.00000,        0.00000,
vel_135_10%-20%:        0.00530,        0.00003,        0.00573,        0.00401,
vel_135_20%-30%:        0.06975,        0.00005,        0.00719,        0.00569,
vel_135_30%-40%:        0.24573,        0.00863,        0.09288,        0.06385,
vel_135_40%-50%:        0.32236,        0.00473,        0.06881,        0.05062,
vel_135_50%-60%:        0.33749,        0.00077,        0.02781,        0.01633,
vel_135_60%-70%:        0.00694,        0.00000,        0.00198,        0.00164,
vel_135_70%-80%:        0.00310,        0.00000,        0.00124,        0.00095,
vel_135_80%-90%:        0.00202,        0.00000,        0.00160,        0.00109,
vel_135_90%-100%:       0.00621,        0.00000,        0.00191,        0.00127,
pop_avg_sum:            0.99892

vel_145_average:        1425.11412,     28.84086,       5.37037,        4.63424,
vel_145_variance:       114067.77813,   25515515.622,   5051.28851,     4472.02615,
vel_145_std_dev:        337.66474,      55.88926,       7.47591,        6.62041,
vel_145_abs_dev:        271.36353,      9.76786,        3.12536,        2.72842,
vel_145_acorr:          -0.26133,       0.00098,        0.03133,        0.02653,
vel_145_wavg:           1417.95151,     31.05698,       5.57288,        4.82248,
vel_145_wvari:          110236.903,     14802954.687,   3847.46081,     3490.73041,
vel_145_w_std_dev:      331.97395,      33.55551,       5.79271,        5.25657,
vel_145_w_abs_dev:      267.58885,      6.21191,        2.49237,        2.16992,
vel_145_sw_avg:         1562.97795,     1882.61692,     43.38913,       24.50455,
vel_145_sw_var:         77228.22508,    129150776.811,  11364.45233,    6770.75513,
vel_145_sw_std_dev:     277.27749,      383.79751,      19.59075,       11.85491,
vel_145_sw_abs_dev:     245.06328,      613.42013,      24.76732,       14.88474,
vel_145_upperq:         1751.44000,     11.18044,       3.34372,        2.64800,
vel_145_lowerq:         1205.20000,     10.14222,       3.18469,        2.40000,
vel_145_rmax:           1587.52000,     69348.944,      263.34188,      228.95200,
vel_145_rmin:           1205.20000,     10.14222,       3.18469,        2.40000,
vel_145_0%-10%:         0.00000,        0.00000,        0.00000,        0.00000,
vel_145_10%-20%:        0.01506,        0.00060,        0.02440,        0.02109,
vel_145_20%-30%:        0.16462,        0.03213,        0.17924,        0.14437,
vel_145_30%-40%:        0.28797,        0.00349,        0.05905,        0.05084,
vel_145_40%-50%:        0.27353,        0.01101,        0.10492,        0.07305,
```

vel_145_50%-60%:	0.24028,	0.02650,	0.16279,	0.14151,
vel_145_60%-70%:	0.00855,	0.00001,	0.00355,	0.00263,
vel_145_70%-80%:	0.00353,	0.00000,	0.00192,	0.00122,
vel_145_80%-90%:	0.00150,	0.00000,	0.00114,	0.00092,
vel_145_90%-100%:	0.00440,	0.00001,	0.00327,	0.00268,
pop_avg_sum:	0.99944			
vel_155_average:	1367.83175,	149.06124,	12.20906,	8.88631,
vel_155_variance:	94604.98902,	9251288.588,	3041.59310,	2406.90375,
vel_155_std_dev:	307.54354,	24.40182,	4.93982,	3.92010,
vel_155_abs_dev:	258.15656,	7.19202,	2.68179,	1.75660,
vel_155_acorr:	-0.31498,	0.00154,	0.03923,	0.02810,
vel_155_wavg:	1361.59565,	177.42156,	13.31997,	9.29693,
vel_155_wvari:	91008.99646,	9638545.020,	3104.60062,	2316.45852,
vel_155_w_std_dev:	301.63761,	26.38965,	5.13709,	3.83879,
vel_155_w_abs_dev:	254.42304,	9.48839,	3.08032,	1.91959,
vel_155_sw_avg:	1499.45818,	2577.65080,	50.77057,	40.31281,
vel_155_sw_var:	70541.08902,	64324696.325,	8020.26785,	7040.66916,
vel_155_sw_std_dev:	265.21476,	224.68852,	14.98961,	13.23178,
vel_155_sw_abs_dev:	237.52724,	374.47788,	19.35143,	15.48790,
vel_155_upperq:	1683.24000,	28.18489,	5.30894,	4.36000,
vel_155_lowerq:	1143.56000,	25.34933,	5.03481,	3.88800,
vel_155_rmax:	1522.12000,	66819.592,	258.49486,	224.71200,
vel_155_rmin:	1143.56000,	25.34933,	5.03481,	3.88800,
vel_155_0%-10%:	0.00000,	0.00000,	0.00000,	0.00000,
vel_155_10%-20%:	0.00447,	0.00007,	0.00844,	0.00556,
vel_155_20%-30%:	0.06244,	0.00587,	0.07660,	0.04346,
vel_155_30%-40%:	0.38862,	0.01376,	0.11729,	0.09896,
vel_155_40%-50%:	0.26214,	0.00487,	0.06979,	0.05009,
vel_155_50%-60%:	0.26048,	0.01356,	0.11645,	0.09209,
vel_155_60%-70%:	0.01027,	0.00006,	0.00764,	0.00569,
vel_155_70%-80%:	0.00409,	0.00000,	0.00198,	0.00167,
vel_155_80%-90%:	0.00258,	0.00001,	0.00224,	0.00184,
vel_155_90%-100%:	0.00370,	0.00001,	0.00315,	0.00275,
pop_avg_sum:	0.99881			
vel_165_average:	1318.78340,	113.35413,	10.64679,	8.27786,
vel_165_variance:	91688.76252,	11974821.427,	3460.46549,	2843.53297,
vel_165_std_dev:	302.75342,	32.36307,	5.68885,	4.67528,
vel_165_abs_dev:	256.22329,	6.20696,	2.49138,	2.01686,
vel_165_acorr:	-0.33764,	0.00023,	0.01514,	0.01135,
vel_165_wavg:	1312.01296,	116.93332,	10.81357,	8.63755,
vel_165_wvari:	88573.94832,	12732643.399,	3568.28298,	3081.29313,
vel_165_w_std_dev:	297.55970,	35.75132,	5.97924,	5.15865,
vel_165_w_abs_dev:	252.58837,	5.44875,	2.33426,	1.91773,
vel_165_sw_avg:	1426.31377,	3716.70254,	60.96476,	36.97281,
vel_165_sw_var:	63970.66910,	154947101.188,	12447.77495,	8206.40468,
vel_165_sw_std_dev:	252.00172,	517.56076,	22.74996,	15.15661,
vel_165_sw_abs_dev:	222.90018,	824.69895,	28.71757,	19.24296,
vel_165_upperq:	1626.68000,	87.20178,	9.33819,	8.41600,
vel_165_lowerq:	1082.84000,	113.45600,	10.65157,	9.32800,
vel_165_rmax:	1190.92000,	52786.92622,	229.75406,	174.19200,
vel_165_rmin:	1082.84000,	113.45600,	10.65157,	9.32800,
vel_165_0%-10%:	0.00000,	0.00000,	0.00000,	0.00000,
vel_165_10%-20%:	0.00348,	0.00008,	0.00882,	0.00557,
vel_165_20%-30%:	0.10395,	0.03355,	0.18318,	0.12459,
vel_165_30%-40%:	0.37687,	0.00073,	0.02710,	0.02146,
vel_165_40%-50%:	0.22123,	0.00741,	0.08610,	0.04887,
vel_165_50%-60%:	0.27055,	0.01956,	0.13985,	0.10604,
vel_165_60%-70%:	0.01113,	0.00009,	0.00933,	0.00675,
vel_165_70%-80%:	0.00449,	0.00001,	0.00229,	0.00132,
vel_165_80%-90%:	0.00190,	0.00000,	0.00136,	0.00117,
vel_165_90%-100%:	0.00594,	0.00001,	0.00331,	0.00251,
pop_avg_sum:	0.99954			
vel_175_average:	1252.11512,	123.15885,	11.09770,	8.57467,
vel_175_variance:	86235.71768,	15739692.046,	3967.32807,	3056.62655,
vel_175_std_dev:	293.59046,	44.84658,	6.69676,	5.18863,

4.1. Motor testing

```
vel_175_abs_dev:        252.82495,      12.74302,       3.56974,        2.86513,
vel_175_acorr:          -0.36929,       0.00121,        0.03474,        0.02297,
vel_175_wavg:           1246.16855,     141.03904,      11.87599,       9.08049,
vel_175_wvari:          83309.34747,    14835137.624,   3851.64090,     2421.86315,
vel_175_w_std_dev:      288.56519,      43.86499,       6.62307,        4.18236,
vel_175_w_abs_dev:      249.36916,      11.80358,       3.43563,        2.39083,
vel_175_sw_avg:         1355.62703,     1337.86856,     36.57689,       24.11070,
vel_175_sw_var:         62089.40594,    32869814.521,   5733.22026,     3496.91922,
vel_175_sw_std_dev:     248.95436,      123.48222,      11.11226,       6.87432,
vel_175_sw_abs_dev:     219.12399,      219.96868,      14.83134,       8.62865,
vel_175_upperq:         1559.70000,     113.78889,      10.66719,       8.04000,
vel_175_lowerq:         1018.50000,     74.19778,       8.61381,        6.60000,
vel_175_rmax:           1288.08000,     80946.99733,    284.51186,      269.72000,
vel_175_rmin:           1018.50000,     74.19778,       8.61381,        6.60000,
vel_175_0%-10%:         0.00000,        0.00000,        0.00000,        0.00000,
vel_175_10%-20%:        0.00053,        0.00000,        0.00167,        0.00095,
vel_175_20%-30%:        0.05577,        0.01052,        0.10255,        0.05819,
vel_175_30%-40%:        0.42313,        0.01127,        0.10616,        0.08937,
vel_175_40%-50%:        0.20917,        0.00776,        0.08811,        0.06845,
vel_175_50%-60%:        0.27574,        0.01015,        0.10074,        0.06609,
vel_175_60%-70%:        0.02213,        0.00031,        0.01755,        0.01545,
vel_175_70%-80%:        0.00415,        0.00000,        0.00190,        0.00147,
vel_175_80%-90%:        0.00344,        0.00001,        0.00372,        0.00306,
vel_175_90%-100%:       0.00492,        0.00001,        0.00383,        0.00323,
pop_avg_sum:            0.99898

vel_185_average:        1217.10575,     134.13132,      11.58151,       9.50393,
vel_185_variance:       85424.82136,    9935545.464,    3152.07003,     2276.88637,
vel_185_std_dev:        292.23040,      29.12465,       5.39673,        3.88560,
vel_185_abs_dev:        253.41836,      8.07588,        2.84181,        2.05333,
vel_185_acorr:          -0.40505,       0.00131,        0.03618,        0.02999,
vel_185_wavg:           1210.57714,     123.59990,      11.11755,       9.18432,
vel_185_wvari:          82435.01231,    11845040.461,   3441.66246,     2553.90116,
vel_185_w_std_dev:      287.05867,      35.92771,       5.99397,        4.44640,
vel_185_w_abs_dev:      250.22913,      8.40582,        2.89928,        2.12078,
vel_185_sw_avg:         1307.72869,     143.66110,      11.98587,       10.36537,
vel_185_sw_var:         58184.05956,    8635968.604,    2938.70186,     1923.94644,
vel_185_sw_std_dev:     241.14503,      36.81360,       6.06742,        3.96751,
vel_185_sw_abs_dev:     204.84191,      203.02879,      14.24882,       11.66992,
vel_185_upperq:         1523.46000,     133.40489,      11.55010,       9.14000,
vel_185_lowerq:         975.44000,      108.91378,      10.43618,       8.24800,
vel_185_rmax:           1084.90000,     52887.38000,    229.97256,      174.36000,
vel_185_rmin:           975.44000,      108.91378,      10.43618,       8.24800,
vel_185_0%-10%:         0.00000,        0.00000,        0.00000,        0.00000,
vel_185_10%-20%:        0.00081,        0.00000,        0.00193,        0.00129,
vel_185_20%-30%:        0.08850,        0.02187,        0.14789,        0.11165,
vel_185_30%-40%:        0.41805,        0.01307,        0.11431,        0.08952,
vel_185_40%-50%:        0.19450,        0.01165,        0.10792,        0.08377,
vel_185_50%-60%:        0.26357,        0.01839,        0.13561,        0.10256,
vel_185_60%-70%:        0.02167,        0.00035,        0.01857,        0.01523,
vel_185_70%-80%:        0.00431,        0.00000,        0.00149,        0.00109,
vel_185_80%-90%:        0.00221,        0.00000,        0.00176,        0.00120,
vel_185_90%-100%:       0.00589,        0.00002,        0.00401,        0.00338,
pop_avg_sum:            0.99951

vel_195_average:        1189.35739,     111.81797,      10.57440,       9.66207,
vel_195_variance:       86210.41318,    4389010.569,    2094.99656,     1547.37885,
vel_195_std_dev:        293.59651,      12.77750,       3.57456,        2.64189,
vel_195_abs_dev:        256.04714,      4.72473,        2.17364,        1.69657,
vel_195_acorr:          -0.44839,       0.00064,        0.02531,        0.01966,
vel_195_wavg:           1182.38169,     121.78390,      11.03557,       10.07902,
vel_195_wvari:          83237.82148,    3463618.10659,  1861.07982,     1509.96796,
vel_195_w_std_dev:      288.49349,      10.36097,       3.21885,        2.61119,
vel_195_w_abs_dev:      252.45238,      4.79834,        2.19051,        1.78002,
vel_195_sw_avg:         1307.50751,     1972.25678,     44.41010,       26.96628,
vel_195_sw_var:         60377.04825,    78807394.265,   8877.35289,     5719.29326,
vel_195_sw_std_dev:     245.19012,      287.61612,      16.95925,       11.01703,
vel_195_sw_abs_dev:     211.41844,      520.04984,      22.80460,       14.40832,
```

```
vel_195_upperq:         1492.00000,     122.66667,      11.07550,       10.00000,
vel_195_lowerq:         932.20000,      103.95556,      10.19586,       9.60000,
vel_195_rmax:           987.70000,      32238.45556,    179.55070,      102.06000,
vel_195_rmin:           932.20000,      103.95556,      10.19586,       9.60000,
vel_195_0%-10%:         0.00000,        0.00000,        0.00000,        0.00000,
vel_195_10%-20%:        0.00251,        0.00002,        0.00428,        0.00345,
vel_195_20%-30%:        0.19328,        0.05397,        0.23232,        0.21064,
vel_195_30%-40%:        0.34876,        0.01448,        0.12035,        0.10191,
vel_195_40%-50%:        0.22586,        0.00858,        0.09261,        0.07358,
vel_195_50%-60%:        0.19767,        0.02754,        0.16596,        0.15325,
vel_195_60%-70%:        0.02166,        0.00056,        0.02356,        0.01894,
vel_195_70%-80%:        0.00398,        0.00001,        0.00260,        0.00181,
vel_195_80%-90%:        0.00157,        0.00000,        0.00214,        0.00143,
vel_195_90%-100%:       0.00408,        0.00002,        0.00425,        0.00394,
pop_avg_sum:            0.99937

-------------------------------------------------------------------------------------

                        ENC 2:
                        AVERAGE         VARIANCE        STND_DEV        ABS_DEV

vel_15_average:         7114.13852,     2640.59686,     51.38674,       35.53211,
vel_15_variance:        1319946.47098,  8672645692.003, 93127.04061,    71363.66317,
vel_15_std_dev:         1148.24509,     1644.08127,     40.54727,       30.93389,
vel_15_abs_dev:         791.20478,      247.08300,      15.71887,       12.47240,
vel_15_acorr:           -0.10354,       0.00295,        0.05428,        0.04686,
vel_15_wavg:            7092.18147,     4590.23582,     67.75128,       41.69548,
vel_15_wvari:           1257818.68875,  2741622320.282, 52360.50344,    41250.17956,
vel_15_w_std_dev:       1121.30480,     549.14047,      23.43375,       18.42176,
vel_15_w_abs_dev:       781.59410,      130.41258,      11.41983,       8.69295,
vel_15_sw_avg:          7719.49438,     8571.42043,     92.58197,       56.45845,
vel_15_sw_var:          796570.02562,   2209123277.973, 47001.31145,    39105.02993,
vel_15_sw_std_dev:      892.16463,      680.32196,      26.08298,       21.75417,
vel_15_sw_abs_dev:      767.41814,      1123.19867,     33.51416,       21.79683,
vel_15_upperq:          7965.72000,     4558.25067,     67.51482,       41.40800,
vel_15_lowerq:          6635.74000,     3422.34711,     58.50083,       45.44800,
vel_15_rmax:            6635.74000,     3422.34711,     58.50083,       45.44800,
vel_15_rmin:            6635.74000,     3422.34711,     58.50083,       45.44800,
vel_15_0%-10%:          0.00007,        0.00000,        0.00022,        0.00012,
vel_15_10%-20%:         0.00292,        0.00006,        0.00778,        0.00442,
vel_15_20%-30%:         0.05392,        0.00123,        0.03511,        0.01977,
vel_15_30%-40%:         0.18748,        0.04543,        0.21315,        0.12125,
vel_15_40%-50%:         0.44073,        0.02399,        0.15488,        0.08437,
vel_15_50%-60%:         0.30683,        0.01425,        0.11937,        0.08301,
vel_15_60%-70%:         0.00406,        0.00001,        0.00232,        0.00167,
vel_15_70%-80%:         0.00142,        0.00000,        0.00108,        0.00087,
vel_15_80%-90%:         0.00073,        0.00000,        0.00081,        0.00060,
vel_15_90%-100%         0.00149,        0.00000,        0.00107,        0.00075,
pop_avg_sum:            0.99965

vel_25_average:         5051.83890,     381.83177,      19.54052,       14.16571,
vel_25_variance:        692132.20990,   2101444801.856, 45841.51832,    28402.94640,
vel_25_std_dev:         831.51279,      798.54253,      28.25849,       17.25721,
vel_25_abs_dev:         560.80801,      398.31343,      19.95779,       13.21530,
vel_25_acorr:           -0.04517,       0.00029,        0.01697,        0.01460,
vel_25_wavg:            5041.28500,     393.90850,      19.84713,       13.99809,
vel_25_wvari:           674481.05437,   1783928949.601, 42236.58307,    31696.81554,
vel_25_w_std_dev:       820.89328,      683.63748,      26.14646,       19.50997,
vel_25_w_abs_dev:       562.15380,      386.23217,      19.65279,       14.04380,
vel_25_sw_avg:          5482.15719,     95.62401,       9.77875,        8.01077,
vel_25_sw_var:          353596.93379,   634137967.208,  25182.09616,    18757.20460,
vel_25_sw_std_dev:      594.28497,      469.23055,      21.66173,       15.98579,
vel_25_sw_abs_dev:      530.07966,      409.05176,      20.22503,       14.91587,
vel_25_upperq:          5604.30000,     45.18444,       6.72194,        5.22000,
vel_25_lowerq:          4753.46000,     4215.68044,     64.92827,       51.72800,
vel_25_rmax:            4753.46000,     4215.68044,     64.92827,       51.72800,
```

4.1. Motor testing

```
vel_25_rmin:            4753.46000,     4215.68044,     64.92827,       51.72800,
vel_25_0%-10%:          0.00000,        0.00000,        0.00000,        0.00000,
vel_25_10%-20%:         0.00336,        0.00000,        0.00108,        0.00091,
vel_25_20%-30%:         0.05136,        0.00006,        0.00753,        0.00526,
vel_25_30%-40%:         0.10754,        0.00003,        0.00587,        0.00515,
vel_25_40%-50%:         0.52829,        0.00087,        0.02943,        0.02358,
vel_25_50%-60%:         0.30325,        0.00132,        0.03628,        0.02860,
vel_25_60%-70%:         0.00198,        0.00000,        0.00059,        0.00048,
vel_25_70%-80%:         0.00114,        0.00000,        0.00040,        0.00031,
vel_25_80%-90%:         0.00031,        0.00000,        0.00034,        0.00025,
vel_25_90%-100%:        0.00198,        0.00000,        0.00080,        0.00061,
pop_avg_sum:            0.99920

vel_35_average:         4180.25679,     10331.55589,    101.64426,      59.81918,
vel_35_variance:        565789.96275,   1002685130.975, 31665.20379,    26348.92151,
vel_35_std_dev:         751.92806,      437.95530,      20.92738,       17.42624,
vel_35_abs_dev:         521.69029,      267.40933,      16.35266,       13.84632,
vel_35_acorr:           0.00202,        0.00062,        0.02494,        0.02032,
vel_35_wavg:            4174.62367,     9655.80581,     98.26396,       58.69716,
vel_35_wvari:           564371.43236,   626517418.297,  25030.32997,    20530.65857,
vel_35_w_std_dev:       751.08171,      275.21836,      16.58971,       13.60748,
vel_35_w_abs_dev:       525.31639,      207.09729,      14.39088,       12.31571,
vel_35_sw_avg:          4542.99599,     11308.07669,    106.33944,      60.91360,
vel_35_sw_var:          277676.51519,   160748921.059,  12678.67978,    9798.34413,
vel_35_sw_std_dev:      526.82679,      144.49917,      12.02078,       9.29378,
vel_35_sw_abs_dev:      468.26470,      129.14684,      11.36428,       8.90932,
vel_35_upperq:          4622.42000,     11231.18622,    105.97729,      59.11600,
vel_35_lowerq:          3864.50000,     26790.83778,    163.67907,      114.74000,
vel_35_rmax:            3864.50000,     26790.83778,    163.67907,      114.74000,
vel_35_rmin:            3864.50000,     26790.83778,    163.67907,      114.74000,
vel_35_0%-10%:          0.00000,        0.00000,        0.00000,        0.00000,
vel_35_10%-20%:         0.01429,        0.00002,        0.00402,        0.00276,
vel_35_20%-30%:         0.06304,        0.00003,        0.00563,        0.00462,
vel_35_30%-40%:         0.10920,        0.00007,        0.00809,        0.00599,
vel_35_40%-50%:         0.62755,        0.00516,        0.07180,        0.05305,
vel_35_50%-60%:         0.18058,        0.00614,        0.07834,        0.05797,
vel_35_60%-70%:         0.00187,        0.00000,        0.00047,        0.00037,
vel_35_70%-80%:         0.00038,        0.00000,        0.00020,        0.00012,
vel_35_80%-90%:         0.00035,        0.00000,        0.00028,        0.00014,
vel_35_90%-100%:        0.00163,        0.00000,        0.00092,        0.00073,
pop_avg_sum:            0.99889

vel_45_average:         3523.16596,     235.74672,      15.35405,       11.39253,
vel_45_variance:        465250.22326,   828720279.375,  28787.50214,    22455.59394,
vel_45_std_dev:         681.80139,      441.20817,      21.00496,       16.30454,
vel_45_abs_dev:         472.10238,      431.11532,      20.76332,       15.24969,
vel_45_acorr:           -0.00913,       0.00028,        0.01669,        0.01111,
vel_45_wavg:            3518.61906,     270.51144,      16.44723,       11.83731,
vel_45_wvari:           463757.10038,   758311912.319,  27537.46380,    20614.50446,
vel_45_w_std_dev:       680.72876,      406.05567,      20.15082,       15.05725,
vel_45_w_abs_dev:       474.36171,      460.31598,      21.45498,       15.35172,
vel_45_sw_avg:          3810.56882,     580.24180,      24.08821,       16.39053,
vel_45_sw_var:          200725.00150,   415261291.834,  20377.96093,    14112.51905,
vel_45_sw_std_dev:      447.52617,      494.80664,      22.24425,       15.60153,
vel_45_sw_abs_dev:      396.32545,      550.95090,      23.47234,       15.65035,
vel_45_upperq:          3887.10000,     21.65556,       4.65355,        3.70000,
vel_45_lowerq:          3191.92000,     4345.77956,     65.92253,       47.76000,
vel_45_rmax:            3191.92000,     4345.77956,     65.92253,       47.76000,
vel_45_rmin:            3191.92000,     4345.77956,     65.92253,       47.76000,
vel_45_0%-10%:          0.00000,        0.00000,        0.00000,        0.00000,
vel_45_10%-20%:         0.02081,        0.00001,        0.00344,        0.00233,
vel_45_20%-30%:         0.06821,        0.00004,        0.00652,        0.00491,
vel_45_30%-40%:         0.10506,        0.00003,        0.00541,        0.00443,
vel_45_40%-50%:         0.65442,        0.00692,        0.08317,        0.05538,
vel_45_50%-60%:         0.14426,        0.00747,        0.08640,        0.05561,
vel_45_60%-70%:         0.00229,        0.00000,        0.00094,        0.00071,
vel_45_70%-80%:         0.00062,        0.00000,        0.00048,        0.00037,
vel_45_80%-90%:         0.00014,        0.00000,        0.00024,        0.00019,
```

```
vel_45_90%-100%:        0.00308,        0.00000,        0.00090,        0.00073,
pop_avg_sum:            0.99889

vel_55_average:         3051.37732,     134.12087,      11.58106,       9.26290,
vel_55_variance:        389493.04341,   915271512.483,  30253.45455,    26414.68887,
vel_55_std_dev:         623.67164,      585.24802,      24.19190,       21.05523,
vel_55_abs_dev:         435.11147,      181.05254,      13.45558,       10.63984,
vel_55_acorr:           -0.01606,       0.00031,        0.01754,        0.01479,
vel_55_wavg:            3045.94297,     109.30894,      10.45509,       7.84648,
vel_55_wvari:           387949.84203,   1200571089.987, 34649.25814,    30151.76910,
vel_55_w_std_dev:       622.30063,      768.62969,      27.72417,       24.05563,
vel_55_w_abs_dev:       437.89224,      179.14043,      13.38434,       10.46134,
vel_55_sw_avg:          3314.48252,     368.49974,      19.19635,       14.81779,
vel_55_sw_var:          171676.93764,   263326108.501,  16227.32598,    13839.49875,
vel_55_sw_std_dev:      413.92393,      382.12635,      19.54805,       16.59940,
vel_55_sw_abs_dev:      367.91014,      189.52709,      13.76688,       10.95581,
vel_55_upperq:          3383.70000,     23.78889,       4.87739,        3.82000,
vel_55_lowerq:          2718.42000,     5780.53733,     76.02985,       60.94000,
vel_55_rmax:            2718.42000,     5780.53733,     76.02985,       60.94000,
vel_55_rmin:            2718.42000,     5780.53733,     76.02985,       60.94000,
vel_55_0%-10%:          0.00000,        0.00000,        0.00000,        0.00000,
vel_55_10%-20%:         0.02841,        0.00014,        0.01162,        0.00771,
vel_55_20%-30%:         0.08115,        0.00051,        0.02255,        0.01296,
vel_55_30%-40%:         0.16501,        0.04141,        0.20348,        0.11573,
vel_55_40%-50%:         0.58314,        0.03868,        0.19667,        0.11791,
vel_55_50%-60%:         0.13478,        0.00635,        0.07971,        0.06652,
vel_55_60%-70%:         0.00184,        0.00000,        0.00080,        0.00057,
vel_55_70%-80%:         0.00097,        0.00000,        0.00054,        0.00043,
vel_55_80%-90%:         0.00038,        0.00000,        0.00045,        0.00033,
vel_55_90%-100%:        0.00333,        0.00000,        0.00193,        0.00157,
pop_avg_sum:            0.99900

vel_65_average:         2695.35980,     140.90050,      11.87015,       8.82418,
vel_65_variance:        309448.12537,   335348074.717,  18312.51143,    15381.07870,
vel_65_std_dev:         556.05582,      277.83802,      16.66847,       13.94665,
vel_65_abs_dev:         382.38323,      165.80576,      12.87656,       10.63874,
vel_65_acorr:           -0.03605,       0.00041,        0.02021,        0.01447,
vel_65_wavg:            2689.48260,     122.36229,      11.06175,       8.41556,
vel_65_wvari:           308174.57390,   515443710.755,  22703.38545,    18619.47440,
vel_65_w_std_dev:       554.78587,      430.24064,      20.74224,       16.95165,
vel_65_w_abs_dev:       384.38567,      167.29524,      12.93427,       10.88626,
vel_65_sw_avg:          2930.66090,     1984.34480,     44.54599,       32.20161,
vel_65_sw_var:          140027.15234,   121428172.281,  11019.44519,    6847.59030,
vel_65_sw_std_dev:      373.92352,      231.50740,      15.21537,       9.40021,
vel_65_sw_abs_dev:      327.06502,      559.33471,      23.65026,       15.58774,
vel_65_upperq:          3004.12000,     18.18844,       4.26479,        3.27200,
vel_65_lowerq:          2442.10000,     4956.62444,     70.40330,       57.56000,
vel_65_rmax:            2442.10000,     4956.62444,     70.40330,       57.56000,
vel_65_rmin:            2442.10000,     4956.62444,     70.40330,       57.56000,
vel_65_0%-10%:          0.00038,        0.00000,        0.00109,        0.00062,
vel_65_10%-20%:         0.03486,        0.00037,        0.01918,        0.01345,
vel_65_20%-30%:         0.08661,        0.00055,        0.02337,        0.01760,
vel_65_30%-40%:         0.21270,        0.06033,        0.24563,        0.18332,
vel_65_40%-50%:         0.51974,        0.05150,        0.22694,        0.16293,
vel_65_50%-60%:         0.13847,        0.01118,        0.10575,        0.08530,
vel_65_60%-70%:         0.00222,        0.00000,        0.00074,        0.00060,
vel_65_70%-80%:         0.00094,        0.00000,        0.00059,        0.00045,
vel_65_80%-90%:         0.00048,        0.00000,        0.00052,        0.00037,
vel_65_90%-100%:        0.00270,        0.00000,        0.00203,        0.00180,
pop_avg_sum:            0.99910

vel_75_average:         2406.94480,     64.95942,       8.05974,        6.17474,
vel_75_variance:        270663.50706,   272975848.814,  16521.98078,    11568.26884,
vel_75_std_dev:         520.04376,      242.21448,      15.56324,       10.93697,
vel_75_abs_dev:         354.60089,      173.63589,      13.17710,       9.24492,
vel_75_acorr:           -0.06359,       0.00076,        0.02763,        0.02165,
vel_75_wavg:            2401.80364,     75.77911,       8.70512,        6.24789,
vel_75_wvari:           269613.49549,   340654183.015,  18456.81942,    13179.43944,
```

```
vel_75_w_std_dev:       518.97785,      306.09473,      17.49556,       12.50373,
vel_75_w_abs_dev:       355.58851,      192.15743,      13.86209,       9.63138,
vel_75_sw_avg:          2661.83715,     82.17019,       9.06478,        7.71921,
vel_75_sw_var:          138594.47310,   127944058.584,  11311.23594,    8799.48046,
vel_75_sw_std_dev:      372.00997,      225.61534,      15.02050,       11.71054,
vel_75_sw_abs_dev:      332.39627,      157.21463,      12.53853,       9.52999,
vel_75_upperq:          2719.10000,     8.54444,        2.92309,        2.30000,
vel_75_lowerq:          2189.98000,     5037.85289,     70.97783,       52.34800,
vel_75_rmax:            2189.98000,     5037.85289,     70.97783,       52.34800,
vel_75_rmin:            2189.98000,     5037.85289,     70.97783,       52.34800,
vel_75_0%-10%:          0.00000,        0.00000,        0.00000,        0.00000,
vel_75_10%-20%:         0.02983,        0.00001,        0.00256,        0.00213,
vel_75_20%-30%:         0.07909,        0.00006,        0.00744,        0.00527,
vel_75_30%-40%:         0.08740,        0.00002,        0.00402,        0.00339,
vel_75_40%-50%:         0.58324,        0.00295,        0.05429,        0.03688,
vel_75_50%-60%:         0.20990,        0.00342,        0.05852,        0.04102,
vel_75_60%-70%:         0.00288,        0.00000,        0.00049,        0.00035,
vel_75_70%-80%:         0.00156,        0.00000,        0.00041,        0.00031,
vel_75_80%-90%:         0.00059,        0.00000,        0.00073,        0.00055,
vel_75_90%-100%:        0.00399,        0.00000,        0.00120,        0.00094,
pop_avg_sum:            0.99848

vel_85_average:         2224.86641,     369.62122,      19.22554,       16.74897,
vel_85_variance:        162903.27889,   439813770.580,  20971.73742,    15528.32206,
vel_85_std_dev:         402.90480,      634.44204,      25.18813,       19.00424,
vel_85_abs_dev:         275.85412,      459.24940,      21.43010,       18.31176,
vel_85_acorr:           -0.12358,       0.00081,        0.02849,        0.02289,
vel_85_wavg:            2221.15494,     437.63177,      20.91965,       18.41998,
vel_85_wvari:           156589.09565,   582909605.050,  24143.52097,    16716.82269,
vel_85_w_std_dev:       394.75503,      841.73304,      29.01264,       20.56473,
vel_85_w_abs_dev:       272.91621,      533.59202,      23.09961,       19.99562,
vel_85_sw_avg:          2431.73578,     825.28204,      28.72772,       24.19729,
vel_85_sw_var:          86413.63017,    121130990.299,  11005.95249,    8648.41425,
vel_85_sw_std_dev:      293.42256,      352.03443,      18.76258,       14.60782,
vel_85_sw_abs_dev:      253.94822,      576.46038,      24.00959,       15.13925,
vel_85_upperq:          2507.26000,     14.56933,       3.81698,        3.06000,
vel_85_lowerq:          1981.22000,     11201.47956,    105.83704,      94.01600,
vel_85_rmax:            1981.22000,     11201.47956,    105.83704,      94.01600,
vel_85_rmin:            1981.22000,     11201.47956,    105.83704,      94.01600,
vel_85_0%-10%:          0.00014,        0.00000,        0.00044,        0.00025,
vel_85_10%-20%:         0.02572,        0.00008,        0.00909,        0.00537,
vel_85_20%-30%:         0.06413,        0.00552,        0.07430,        0.05453,
vel_85_30%-40%:         0.22609,        0.04991,        0.22341,        0.16523,
vel_85_40%-50%:         0.43943,        0.03765,        0.19403,        0.13284,
vel_85_50%-60%:         0.23688,        0.02372,        0.15403,        0.13202,
vel_85_60%-70%:         0.00297,        0.00000,        0.00118,        0.00092,
vel_85_70%-80%:         0.00114,        0.00000,        0.00069,        0.00054,
vel_85_80%-90%:         0.00042,        0.00000,        0.00048,        0.00037,
vel_85_90%-100%:        0.00176,        0.00000,        0.00172,        0.00123,
pop_avg_sum:            0.99869

vel_95_average:         1990.53059,     73.02150,       8.54526,        6.79312,
vel_95_variance:        201444.63085,   54853203.679,   7406.29487,     5827.75260,
vel_95_std_dev:         448.75638,      69.26626,       8.32264,        6.54559,
vel_95_abs_dev:         308.49574,      48.93297,       6.99521,        5.65514,
vel_95_acorr:           -0.08992,       0.00009,        0.00935,        0.00765,
vel_95_wavg:            1985.43126,     89.38125,       9.45417,        7.14757,
vel_95_wvari:           198869.38429,   64585073.724,   8036.48392,     6320.36146,
vel_95_w_std_dev:       445.86484,      82.14221,       9.06323,        7.10338,
vel_95_w_abs_dev:       307.50968,      66.82437,       8.17462,        6.11793,
vel_95_sw_avg:          2224.15535,     299.05331,      17.29316,       11.65802,
vel_95_sw_var:          112744.76050,   164400812.532,  12821.88803,    9645.51079,
vel_95_sw_std_dev:      335.26474,      380.35109,      19.50259,       14.60514,
vel_95_sw_abs_dev:      296.79816,      457.74876,      21.39506,       15.58933,
vel_95_upperq:          2312.96000,     8.01600,        2.83125,        2.04000,
vel_95_lowerq:          1801.26000,     1002.85378,     31.66787,       25.04800,
vel_95_rmax:            1801.26000,     1002.85378,     31.66787,       25.04800,
vel_95_rmin:            1801.26000,     1002.85378,     31.66787,       25.04800,
```

vel_95_0%-10%:	0.00007,	0.00000,	0.00022,	0.00012,
vel_95_10%-20%:	0.04405,	0.00075,	0.02741,	0.01985,
vel_95_20%-30%:	0.10754,	0.00683,	0.08267,	0.04640,
vel_95_30%-40%:	0.22725,	0.05203,	0.22810,	0.19134,
vel_95_40%-50%:	0.38316,	0.02916,	0.17076,	0.12612,
vel_95_50%-60%:	0.22786,	0.02438,	0.15614,	0.13457,
vel_95_60%-70%:	0.00318,	0.00000,	0.00151,	0.00117,
vel_95_70%-80%:	0.00187,	0.00000,	0.00121,	0.00083,
vel_95_80%-90%:	0.00149,	0.00000,	0.00135,	0.00100,
vel_95_90%-100%:	0.00298,	0.00001,	0.00230,	0.00194,
pop_avg_sum:	0.99945			
vel_105_average:	1848.84582,	69.57813,	8.34135,	6.33747,
vel_105_variance:	179509.62510,	111795032.085,	10573.31699,	8884.45976,
vel_105_std_dev:	423.52237,	153.80753,	12.40192,	10.46290,
vel_105_abs_dev:	294.99020,	61.32382,	7.83095,	6.43371,
vel_105_acorr:	-0.12291,	0.00069,	0.02633,	0.02031,
vel_105_wavg:	1842.94336,	86.05956,	9.27683,	7.43518,
vel_105_wvari:	177088.60523,	138742278.634,	11778.89123,	9150.37350,
vel_105_w_std_dev:	420.61284,	192.71283,	13.88210,	10.84313,
vel_105_w_abs_dev:	293.59282,	78.33111,	8.85049,	6.95014,
vel_105_sw_avg:	2077.88159,	481.12279,	21.93451,	13.72613,
vel_105_sw_var:	106596.56569,	37712156.189,	6141.02241,	4210.06312,
vel_105_sw_std_dev:	326.36498,	91.62781,	9.57224,	6.49389,
vel_105_sw_abs_dev:	291.96109,	243.77189,	15.61320,	8.97316,
vel_105_upperq:	2170.60000,	53.15556,	7.29079,	6.12000,
vel_105_lowerq:	1674.50000,	389.30000,	19.73069,	15.04000,
vel_105_rmax:	1674.50000,	389.30000,	19.73069,	15.04000,
vel_105_rmin:	1674.50000,	389.30000,	19.73069,	15.04000,
vel_105_0%-10%:	0.00000,	0.00000,	0.00000,	0.00000,
vel_105_10%-20%:	0.03093,	0.00004,	0.00646,	0.00420,
vel_105_20%-30%:	0.07566,	0.00004,	0.00620,	0.00519,
vel_105_30%-40%:	0.10879,	0.00565,	0.07520,	0.04256,
vel_105_40%-50%:	0.45447,	0.00052,	0.02274,	0.01570,
vel_105_50%-60%:	0.31671,	0.01108,	0.10524,	0.06213,
vel_105_60%-70%:	0.00412,	0.00000,	0.00090,	0.00059,
vel_105_70%-80%:	0.00229,	0.00000,	0.00066,	0.00051,
vel_105_80%-90%:	0.00180,	0.00000,	0.00148,	0.00100,
vel_105_90%-100%:	0.00433,	0.00000,	0.00157,	0.00107,
pop_avg_sum:	0.99910			
vel_115_average:	1721.80118,	70.70341,	8.40853,	5.62690,
vel_115_variance:	148827.99953,	72697020.579,	8526.25478,	6675.39921,
vel_115_std_dev:	385.64122,	120.94753,	10.99761,	8.63588,
vel_115_abs_dev:	278.07396,	37.59841,	6.13175,	4.59838,
vel_115_acorr:	-0.15045,	0.00094,	0.03060,	0.02608,
vel_115_wavg:	1715.68722,	77.98785,	8.83107,	5.34080,
vel_115_wvari:	145991.59883,	88444838.637,	9404.51161,	7548.99807,
vel_115_w_std_dev:	381.91233,	149.52324,	12.22797,	9.85650,
vel_115_w_abs_dev:	275.77453,	44.55586,	6.67502,	5.05508,
vel_115_sw_avg:	1932.23576,	311.51324,	17.64974,	10.91828,
vel_115_sw_var:	94527.23775,	33306496.93014,	5771.17812,	3398.78459,
vel_115_sw_std_dev:	307.32667,	86.17257,	9.28292,	5.47878,
vel_115_sw_abs_dev:	276.29041,	170.39089,	13.05339,	7.51506,
vel_115_upperq:	2034.70000,	17.11333,	4.13683,	3.16000,
vel_115_lowerq:	1543.74000,	111.26267,	10.54811,	7.64800,
vel_115_rmax:	1592.82000,	24031.76400,	155.02182,	88.03600,
vel_115_rmin:	1543.74000,	111.26267,	10.54811,	7.64800,
vel_115_0%-10%:	0.00000,	0.00000,	0.00000,	0.00000,
vel_115_10%-20%:	0.02117,	0.00006,	0.00752,	0.00580,
vel_115_20%-30%:	0.07770,	0.00006,	0.00749,	0.00557,
vel_115_30%-40%:	0.14811,	0.01459,	0.12080,	0.08096,
vel_115_40%-50%:	0.43207,	0.00032,	0.01802,	0.01191,
vel_115_50%-60%:	0.30611,	0.01440,	0.11998,	0.08503,
vel_115_60%-70%:	0.00423,	0.00000,	0.00103,	0.00083,
vel_115_70%-80%:	0.00253,	0.00000,	0.00097,	0.00080,
vel_115_80%-90%:	0.00170,	0.00000,	0.00079,	0.00058,
vel_115_90%-100%:	0.00482,	0.00000,	0.00219,	0.00164,

4.1. Motor testing

```
pop_avg_sum:            0.99844

vel_125_average:        1619.56570,     129.72762,      11.38980,       9.14937,
vel_125_variance:       125438.15036,   40987133.992,   6402.11949,     4750.91060,
vel_125_std_dev:        354.06948,      81.06327,       9.00351,        6.70052,
vel_125_abs_dev:        265.01881,      17.72221,       4.20978,        2.88154,
vel_125_acorr:          -0.16789,       0.00053,        0.02296,        0.01929,
vel_125_wavg:           1613.61277,     130.95311,      11.44347,       9.20382,
vel_125_wvari:          121894.66891,   52035526.51139, 7213.56545,     5142.94993,
vel_125_w_std_dev:      348.99866,      105.11609,      10.25261,       7.34886,
vel_125_w_abs_dev:      262.02354,      21.54347,       4.64149,        2.86141,
vel_125_sw_avg:         1816.77136,     326.98892,      18.08283,       13.23432,
vel_125_sw_var:         84275.72142,    18120278.97760, 4256.79210,     3406.34976,
vel_125_sw_std_dev:     290.21931,      53.85883,       7.33886,        5.88782,
vel_125_sw_abs_dev:     264.86987,      67.52460,       8.21734,        6.74829,
vel_125_upperq:         1920.92000,     34.50844,       5.87439,        4.50400,
vel_125_lowerq:         1430.86000,     118.10711,      10.86771,       9.04800,
vel_125_rmax:           1430.86000,     118.10711,      10.86771,       9.04800,
vel_125_rmin:           1430.86000,     118.10711,      10.86771,       9.04800,
vel_125_0%-10%:         0.00000,        0.00000,        0.00000,        0.00000,
vel_125_10%-20%:        0.01534,        0.00015,        0.01221,        0.01058,
vel_125_20%-30%:        0.07749,        0.00016,        0.01269,        0.00934,
vel_125_30%-40%:        0.26345,        0.03530,        0.18788,        0.17411,
vel_125_40%-50%:        0.40554,        0.00102,        0.03200,        0.02577,
vel_125_50%-60%:        0.22453,        0.03147,        0.17740,        0.16432,
vel_125_60%-70%:        0.00450,        0.00001,        0.00234,        0.00201,
vel_125_70%-80%:        0.00263,        0.00000,        0.00129,        0.00101,
vel_125_80%-90%:        0.00197,        0.00000,        0.00166,        0.00133,
vel_125_90%-100%:       0.00419,        0.00001,        0.00353,        0.00315,
pop_avg_sum:            0.99965

vel_135_average:        1518.49675,     55.37109,       7.44118,        5.20142,
vel_135_variance:       116340.02685,   34227649.32664, 5850.44010,     4413.09079,
vel_135_std_dev:        340.99007,      73.10767,       8.55030,        6.44137,
vel_135_abs_dev:        263.93044,      16.98329,       4.12108,        3.05205,
vel_135_acorr:          -0.18163,       0.00084,        0.02892,        0.02381,
vel_135_wavg:           1513.35196,     58.74968,       7.66483,        5.85071,
vel_135_wvari:          112740.54882,   35952017.84874, 5996.00015,     4022.40499,
vel_135_w_std_dev:      335.66378,      78.19813,       8.84297,        5.95032,
vel_135_w_abs_dev:      260.70362,      22.35946,       4.72858,        3.16712,
vel_135_sw_avg:         1680.14413,     685.10593,      26.17453,       18.63680,
vel_135_sw_var:         75668.44057,    33038076.99125, 5747.87587,     3860.80879,
vel_135_sw_std_dev:     274.90177,      108.28319,      10.40592,       7.01671,
vel_135_sw_abs_dev:     245.96899,      188.13015,      13.71605,       10.44191,
vel_135_upperq:         1821.40000,     44.48889,       6.67000,        4.68000,
vel_135_lowerq:         1321.62000,     43.08844,       6.56418,        4.50400,
vel_135_rmax:           1421.82000,     44530.76844,    211.02315,      160.07200,
vel_135_rmin:           1321.62000,     43.08844,       6.56418,        4.50400,
vel_135_0%-10%:         0.00000,        0.00000,        0.00000,        0.00000,
vel_135_10%-20%:        0.02048,        0.00082,        0.02858,        0.02439,
vel_135_20%-30%:        0.15945,        0.02430,        0.15588,        0.12260,
vel_135_30%-40%:        0.28869,        0.02665,        0.16325,        0.14526,
vel_135_40%-50%:        0.27709,        0.02162,        0.14702,        0.12622,
vel_135_50%-60%:        0.23893,        0.02848,        0.16876,        0.14675,
vel_135_60%-70%:        0.00525,        0.00000,        0.00123,        0.00081,
vel_135_70%-80%:        0.00304,        0.00000,        0.00142,        0.00098,
vel_135_80%-90%:        0.00145,        0.00000,        0.00219,        0.00141,
vel_135_90%-100%:       0.00404,        0.00001,        0.00313,        0.00268,
pop_avg_sum:            0.99844

vel_145_average:        1433.58744,     49.37287,       7.02658,        5.70744,
vel_145_variance:       103524.83713,   12134091.65652, 3483.40231,     1778.75890,
vel_145_std_dev:        321.71165,      29.39271,       5.42150,        2.76686,
vel_145_abs_dev:        259.51240,      8.94446,        2.99073,        1.92911,
vel_145_acorr:          -0.24912,       0.00069,        0.02633,        0.02227,
vel_145_wavg:           1428.06243,     48.58174,       6.97006,        5.37961,
vel_145_wvari:          99874.78456,    14740738.12101, 3839.36689,     2650.78163,
vel_145_w_std_dev:      315.97775,      36.49332,       6.04097,        4.18554,
```

```
vel_145_w_abs_dev:      256.12249,      9.21338,        3.03536,        1.65772,
vel_145_sw_avg:         1604.24969,     1634.46993,     40.42858,       34.53347,
vel_145_sw_var:         76026.36280,    64584229.86254, 8036.43141,     6245.22250,
vel_145_sw_std_dev:     275.39605,      203.75285,      14.27420,       11.10017,
vel_145_sw_abs_dev:     253.18780,      333.82005,      18.27074,       14.86400,
vel_145_upperq:         1737.98000,     11.73733,       3.42598,        2.97600,
vel_145_lowerq:         1227.04000,     37.48267,       6.12231,        4.76800,
vel_145_rmax:           1329.44000,     45945.75822,    214.34962,      162.62400,
vel_145_rmin:           1227.04000,     37.48267,       6.12231,        4.76800,
vel_145_0%-10%:         0.00000,        0.00000,        0.00000,        0.00000,
vel_145_10%-20%:        0.01227,        0.00057,        0.02395,        0.01417,
vel_145_20%-30%:        0.11532,        0.02070,        0.14386,        0.08174,
vel_145_30%-40%:        0.33641,        0.02615,        0.16172,        0.14605,
vel_145_40%-50%:        0.29760,        0.01553,        0.12462,        0.09099,
vel_145_50%-60%:        0.22210,        0.02517,        0.15864,        0.13901,
vel_145_60%-70%:        0.00615,        0.00002,        0.00403,        0.00276,
vel_145_70%-80%:        0.00259,        0.00000,        0.00142,        0.00110,
vel_145_80%-90%:        0.00342,        0.00001,        0.00296,        0.00258,
vel_145_90%-100%:       0.00307,        0.00001,        0.00322,        0.00287,
pop_avg_sum:            0.99893

vel_155_average:        1363.73625,     118.15292,      10.86982,       7.66105,
vel_155_variance:       94207.57672,    19258844.32030, 4388.48998,     2911.83841,
vel_155_std_dev:        306.85970,      49.66773,       7.04753,        4.68697,
vel_155_abs_dev:        253.85271,      6.74392,        2.59691,        2.04877,
vel_155_acorr:          -0.27163,       0.00359,        0.05989,        0.03711,
vel_155_wavg:           1356.53235,     138.18465,      11.75520,       8.32785,
vel_155_wvari:          90765.56425,    21940626.13319, 4684.08221,     3174.26240,
vel_155_w_std_dev:      301.18489,      59.13864,       7.69017,        5.22927,
vel_155_w_abs_dev:      250.35268,      5.73753,        2.39531,        1.89000,
vel_155_sw_avg:         1517.07527,     2272.55651,     47.67134,       40.02825,
vel_155_sw_var:         73314.13952,    96998921.86437, 9848.80307,     7767.15870,
vel_155_sw_std_dev:     270.23630,      318.31582,      17.84141,       14.12816,
vel_155_sw_abs_dev:     246.27099,      549.87372,      23.44939,       19.00027,
vel_155_upperq:         1668.80000,     25.51111,       5.05085,        3.96000,
vel_155_lowerq:         1155.20000,     21.51111,       4.63801,        3.56000,
vel_155_rmax:           1360.50000,     70750.05556,    265.98883,      247.20000,
vel_155_rmin:           1155.20000,     21.51111,       4.63801,        3.56000,
vel_155_0%-10%:         0.00000,        0.00000,        0.00000,        0.00000,
vel_155_10%-20%:        0.00791,        0.00032,        0.01791,        0.01065,
vel_155_20%-30%:        0.11208,        0.02549,        0.15965,        0.09053,
vel_155_30%-40%:        0.36501,        0.02054,        0.14331,        0.12745,
vel_155_40%-50%:        0.26008,        0.01454,        0.12059,        0.09585,
vel_155_50%-60%:        0.23739,        0.02205,        0.14850,        0.13079,
vel_155_60%-70%:        0.00738,        0.00001,        0.00371,        0.00331,
vel_155_70%-80%:        0.00280,        0.00000,        0.00192,        0.00156,
vel_155_80%-90%:        0.00318,        0.00001,        0.00231,        0.00169,
vel_155_90%-100%:       0.00269,        0.00001,        0.00258,        0.00235,
pop_avg_sum:            0.99852

vel_165_average:        1309.37525,     83.15532,       9.11895,        7.40792,
vel_165_variance:       90633.85203,    11208966.26524, 3347.97943,     2274.71738,
vel_165_std_dev:        301.00892,      30.53359,       5.52572,        3.75988,
vel_165_abs_dev:        253.25144,      10.51038,       3.24197,        2.21395,
vel_165_acorr:          -0.29589,       0.00036,        0.01908,        0.01311,
vel_165_wavg:           1301.47044,     88.08116,       9.38516,        7.67455,
vel_165_wvari:          87154.34081,    11401142.81838, 3376.55784,     2378.92847,
vel_165_w_std_dev:      295.16965,      32.46194,       5.69754,        4.04675,
vel_165_w_abs_dev:      249.36354,      10.11387,       3.18023,        2.18251,
vel_165_sw_avg:         1429.27132,     1727.65366,     41.56505,       30.72080,
vel_165_sw_var:         65463.30642,    49131438.70445, 7009.38219,     5102.61894,
vel_165_sw_std_dev:     255.54475,      177.98446,      13.34108,       9.81119,
vel_165_sw_abs_dev:     230.39141,      323.04072,      17.97333,       12.99417,
vel_165_upperq:         1610.42000,     97.24844,       9.86146,        8.82400,
vel_165_lowerq:         1091.26000,     112.67600,      10.61490,       9.19200,
vel_165_rmax:           1402.16000,     73248.25600,    270.64415,      251.40800,
vel_165_rmin:           1091.26000,     112.67600,      10.61490,       9.19200,
vel_165_0%-10%:         0.00000,        0.00000,        0.00000,        0.00000,
```

4.1. Motor testing

```
vel_165_10%-20%:        0.00014,        0.00000,        0.00043,        0.00025,
vel_165_20%-30%:        0.04967,        0.00016,        0.01268,        0.01017,
vel_165_30%-40%:        0.38457,        0.01807,        0.13444,        0.12241,
vel_165_40%-50%:        0.24794,        0.00848,        0.09207,        0.07337,
vel_165_50%-60%:        0.29121,        0.00758,        0.08707,        0.06371,
vel_165_60%-70%:        0.01377,        0.00010,        0.01013,        0.00841,
vel_165_70%-80%:        0.00347,        0.00000,        0.00136,        0.00113,
vel_165_80%-90%:        0.00422,        0.00001,        0.00278,        0.00246,
vel_165_90%-100%:       0.00439,        0.00001,        0.00344,        0.00310,
pop_avg_sum:            0.99938

vel_175_average:        1238.21749,     199.95271,      14.14046,       11.39659,
vel_175_variance:       85570.94544,    13693740.08657, 3700.50538,     2828.11875,
vel_175_std_dev:        292.46352,      40.03677,       6.32746,        4.83518,
vel_175_abs_dev:        251.87872,      8.01708,        2.83145,        2.21107,
vel_175_acorr:          -0.32199,       0.00166,        0.04072,        0.03344,
vel_175_wavg:           1231.49772,     183.89432,      13.56076,       11.41477,
vel_175_wvari:          81510.79742,    6439835.71930,  2537.68314,     2021.00479,
vel_175_w_std_dev:      285.46955,      19.92809,       4.46409,        3.55372,
vel_175_w_abs_dev:      247.92069,      5.10307,        2.25900,        1.90117,
vel_175_sw_avg:         1344.77186,     3010.55731,     54.86855,       40.31283,
vel_175_sw_var:         66830.48843,    183218493.274,  13535.82259,    9909.74886,
vel_175_sw_std_dev:     257.41405,      631.66215,      25.13289,       18.73129,
vel_175_sw_abs_dev:     227.35533,      1141.00627,     33.77878,       26.37826,
vel_175_upperq:         1547.84000,     80.55822,       8.97542,        7.56800,
vel_175_lowerq:         1018.08000,     127.03289,      11.27089,       7.53600,
vel_175_rmax:           1176.92000,     65354.43733,    255.64514,      222.04800,
vel_175_rmin:           1018.08000,     127.03289,      11.27089,       7.53600,
vel_175_0%-10%:         0.00000,        0.00000,        0.00000,        0.00000,
vel_175_10%-20%:        0.01273,        0.00135,        0.03680,        0.02087,
vel_175_20%-30%:        0.11583,        0.02757,        0.16604,        0.11517,
vel_175_30%-40%:        0.43681,        0.01333,        0.11546,        0.10752,
vel_175_40%-50%:        0.17256,        0.01833,        0.13537,        0.12621,
vel_175_50%-60%:        0.23678,        0.02075,        0.14403,        0.12358,
vel_175_60%-70%:        0.01526,        0.00033,        0.01803,        0.01450,
vel_175_70%-80%:        0.00374,        0.00000,        0.00164,        0.00106,
vel_175_80%-90%:        0.00230,        0.00000,        0.00180,        0.00137,
vel_175_90%-100%:       0.00302,        0.00000,        0.00217,        0.00175,
pop_avg_sum:            0.99904

vel_185_average:        1205.48335,     172.72944,      13.14266,       11.26381,
vel_185_variance:       85188.77216,    6608813.04906,  2570.76118,     1952.50683,
vel_185_std_dev:        291.84135,      19.33098,       4.39670,        3.34357,
vel_185_abs_dev:        253.71198,      9.57103,        3.09371,        2.36791,
vel_185_acorr:          -0.36901,       0.00020,        0.01401,        0.01132,
vel_185_wavg:           1198.17666,     179.61593,      13.40209,       11.46801,
vel_185_wvari:          81908.07303,    10708654.74564, 3272.40810,     2475.95271,
vel_185_w_std_dev:      286.14438,      32.73866,       5.72177,        4.32748,
vel_185_w_abs_dev:      250.01601,      13.46286,       3.66918,        2.72607,
vel_185_sw_avg:         1316.35323,     3694.67347,     60.78383,       43.80698,
vel_185_sw_var:         66919.23499,    181923946.157,  13487.91853,    10325.02602,
vel_185_sw_std_dev:     257.63042,      606.44661,      24.62614,       19.12236,
vel_185_sw_abs_dev:     229.46920,      1152.74342,     33.95208,       26.48047,
vel_185_upperq:         1513.50000,     113.61111,      10.65885,       8.90000,
vel_185_lowerq:         973.40000,      154.48889,      12.42936,       10.00000,
vel_185_rmax:           1081.50000,     55594.27778,    235.78439,      178.80000,
vel_185_rmin:           973.40000,      154.48889,      12.42936,       10.00000,
vel_185_0%-10%:         0.00000,        0.00000,        0.00000,        0.00000,
vel_185_10%-20%:        0.00351,        0.00012,        0.01109,        0.00631,
vel_185_20%-30%:        0.11458,        0.03018,        0.17371,        0.10940,
vel_185_30%-40%:        0.43693,        0.00787,        0.08872,        0.07829,
vel_185_40%-50%:        0.17289,        0.01343,        0.11588,        0.10106,
vel_185_50%-60%:        0.24784,        0.02110,        0.14526,        0.12367,
vel_185_60%-70%:        0.01273,        0.00010,        0.01003,        0.00812,
vel_185_70%-80%:        0.00382,        0.00000,        0.00186,        0.00114,
vel_185_80%-90%:        0.00275,        0.00000,        0.00204,        0.00138,
vel_185_90%-100%:       0.00399,        0.00001,        0.00361,        0.00312,
pop_avg_sum:            0.99904
```

vel_195_average:	1184.62396,	102.57724,	10.12804,	8.68870,
vel_195_variance:	85387.79724,	3941686.30676,	1985.36805,	1769.33504,
vel_195_std_dev:	292.19410,	11.56051,	3.40008,	3.03320,
vel_195_abs_dev:	254.92408,	4.13584,	2.03368,	1.69298,
vel_195_acorr:	-0.40429,	0.00044,	0.02098,	0.01866,
vel_195_wavg:	1176.75894,	102.53825,	10.12612,	8.47008,
vel_195_wvari:	82171.34598,	5599629.00024,	2366.35352,	2032.70254,
vel_195_w_std_dev:	286.62853,	17.14491,	4.14064,	3.54980,
vel_195_w_abs_dev:	251.40733,	5.60852,	2.36823,	1.91842,
vel_195_sw_avg:	1289.13792,	3282.84767,	57.29614,	32.49810,
vel_195_sw_var:	60542.14227,	96235130.84449,	9809.95060,	6105.53557,
vel_195_sw_std_dev:	245.42597,	342.48218,	18.50627,	11.77873,
vel_195_sw_abs_dev:	216.08980,	633.47989,	25.16903,	16.35558,
vel_195_upperq:	1481.86000,	95.21822,	9.75798,	9.14000,
vel_195_lowerq:	932.10000,	95.21111,	9.75762,	8.50000,
vel_195_rmax:	1042.26000,	53526.27600,	231.35746,	175.41600,
vel_195_rmin:	932.10000,	95.21111,	9.75762,	8.50000,
vel_195_0%-10%:	0.00000,	0.00000,	0.00000,	0.00000,
vel_195_10%-20%:	0.00231,	0.00005,	0.00732,	0.00416,
vel_195_20%-30%:	0.11346,	0.03291,	0.18140,	0.12795,
vel_195_30%-40%:	0.40684,	0.00516,	0.07182,	0.05171,
vel_195_40%-50%:	0.16907,	0.00994,	0.09971,	0.07215,
vel_195_50%-60%:	0.27230,	0.02018,	0.14206,	0.10616,
vel_195_60%-70%:	0.02330,	0.00088,	0.02973,	0.02012,
vel_195_70%-80%:	0.00456,	0.00001,	0.00244,	0.00173,
vel_195_80%-90%:	0.00338,	0.00001,	0.00305,	0.00232,
vel_195_90%-100%:	0.00422,	0.00001,	0.00373,	0.00332,
pop_avg_sum:	0.99945			

MOTOR 1, STEP: 10, SAMPLES 10 x step:

	ENC 1:			
	AVERAGE	VARIANCE	STND_DEV	ABS_DEV
vel_15_average:	7140.37207,	3149.18206,	56.11757,	37.17544,
vel_15_variance:	1399994.27884,	17393188218.18,	131883.23706,	93197.05309,
vel_15_std_dev:	1181.96456,	3282.28634,	57.29124,	40.09735,
vel_15_abs_dev:	822.38935,	620.63390,	24.91252,	14.32298,
vel_15_acorr:	0.07745,	0.00339,	0.05825,	0.04341,
vel_15_wavg:	7115.99920,	2406.67384,	49.05786,	36.69912,
vel_15_wvari:	1305912.65891,	9079633935.964,	95287.11317,	62524.70409,
vel_15_w_std_dev:	1142.04961,	1817.05422,	42.62692,	27.93133,
vel_15_w_abs_dev:	815.92755,	525.81557,	22.93067,	13.76193,
vel_15_sw_avg:	7693.72698,	6401.15643,	80.00723,	64.13676,
vel_15_sw_var:	822288.12645,	5442510194.830,	73773.37050,	58301.58931,
vel_15_sw_std_dev:	905.93725,	1739.80948,	41.71102,	32.87819,
vel_15_sw_abs_dev:	752.56092,	3246.52535,	56.97829,	46.45896,
vel_15_upperq:	7886.12000,	2467.81511,	49.67711,	32.07200,
vel_15_lowerq:	6381.86000,	9117.94711,	95.48794,	66.96800,
vel_15_rmax:	6381.86000,	9117.94711,	95.48794,	66.96800,
vel_15_rmin:	6381.86000,	9117.94711,	95.48794,	66.96800,
vel_15_0%-10%:	0.00017,	0.00000,	0.00055,	0.00031,
vel_15_10%-20%:	0.00125,	0.00000,	0.00097,	0.00080,
vel_15_20%-30%:	0.05678,	0.00035,	0.01863,	0.01254,
vel_15_30%-40%:	0.16155,	0.00113,	0.03365,	0.02151,
vel_15_40%-50%:	0.63124,	0.00596,	0.07718,	0.05515,
vel_15_50%-60%:	0.13975,	0.01372,	0.11712,	0.08387,
vel_15_60%-70%:	0.00440,	0.00000,	0.00126,	0.00094,
vel_15_70%-80%:	0.00135,	0.00000,	0.00140,	0.00107,
vel_15_80%-90%:	0.00146,	0.00000,	0.00086,	0.00068,
vel_15_90%-100%:	0.00149,	0.00000,	0.00080,	0.00066,
pop_avg_sum:	0.99945			
vel_25_average:	4976.12568,	523.97494,	22.89050,	18.34990,

4.1. Motor testing

vel_25_variance:	740340.03095,	2776244902.630,	52690.08353,	40843.85727,
vel_25_std_dev:	859.95390,	910.34770,	30.17197,	23.52120,
vel_25_abs_dev:	613.20346,	425.37609,	20.62465,	17.32742,
vel_25_acorr:	0.01700,	0.00048,	0.02185,	0.01739,
vel_25_wavg:	4970.44963,	586.39395,	24.21557,	19.48967,
vel_25_wvari:	734485.47914,	3077265969.460,	55473.11033,	40649.27125,
vel_25_w_std_dev:	856.49365,	1004.55953,	31.69479,	23.42661,
vel_25_w_abs_dev:	615.03700,	537.89610,	23.19259,	19.40276,
vel_25_sw_avg:	5344.33034,	1046.05525,	32.34278,	22.10638,
vel_25_sw_var:	312615.79158,	497241209.198,	22298.90601,	18709.58014,
vel_25_sw_std_dev:	558.80380,	393.44620,	19.83548,	16.63472,
vel_25_sw_abs_dev:	504.37551,	198.11377,	14.07529,	10.71094,
vel_25_upperq:	5439.74000,	206.72044,	14.37778,	11.04400,
vel_25_lowerq:	4469.14000,	7492.24933,	86.55778,	69.50000,
vel_25_rmax:	4469.14000,	7492.24933,	86.55778,	69.50000,
vel_25_rmin:	4469.14000,	7492.24933,	86.55778,	69.50000,
vel_25_0%-10%:	0.00000,	0.00000,	0.00000,	0.00000,
vel_25_10%-20%:	0.00652,	0.00009,	0.00933,	0.00528,
vel_25_20%-30%:	0.06982,	0.00078,	0.02795,	0.01612,
vel_25_30%-40%:	0.18859,	0.04364,	0.20890,	0.11882,
vel_25_40%-50%:	0.56549,	0.03873,	0.19679,	0.11389,
vel_25_50%-60%:	0.16323,	0.00748,	0.08649,	0.07075,
vel_25_60%-70%:	0.00198,	0.00000,	0.00063,	0.00047,
vel_25_70%-80%:	0.00097,	0.00000,	0.00073,	0.00062,
vel_25_80%-90%:	0.00042,	0.00000,	0.00027,	0.00022,
vel_25_90%-100%:	0.00243,	0.00000,	0.00124,	0.00090,
pop_avg_sum:	0.99945			

vel_35_average:	4159.32956,	21841.26440,	147.78790,	111.84823,
vel_35_variance:	609323.38034,	812552560.895,	28505.30759,	23545.40982,
vel_35_std_dev:	780.40050,	331.60435,	18.21001,	15.06480,
vel_35_abs_dev:	568.42645,	192.40696,	13.87108,	10.55195,
vel_35_acorr:	0.04088,	0.00029,	0.01706,	0.01352,
vel_35_wavg:	4153.74848,	20572.08335,	143.42972,	108.50413,
vel_35_wvari:	609549.48441,	1021259182.968,	31957.14604,	26323.77578,
vel_35_w_std_dev:	780.49455,	419.72055,	20.48708,	16.86508,
vel_35_w_abs_dev:	572.55742,	129.06524,	11.36069,	7.93769,
vel_35_sw_avg:	4473.88278,	24588.96602,	156.80869,	116.92994,
vel_35_sw_var:	256762.47987,	555092058.694,	23560.39173,	17908.05380,
vel_35_sw_std_dev:	506.22351,	555.81779,	23.57579,	17.89290,
vel_35_sw_abs_dev:	456.26161,	307.63842,	17.53962,	13.36050,
vel_35_upperq:	4583.32000,	18178.81956,	134.82885,	102.27200,
vel_35_lowerq:	3696.30000,	37479.77111,	193.59693,	144.16000,
vel_35_rmax:	3696.30000,	37479.77111,	193.59693,	144.16000,
vel_35_rmin:	3696.30000,	37479.77111,	193.59693,	144.16000,
vel_35_0%-10%:	0.00000,	0.00000,	0.00000,	0.00000,
vel_35_10%-20%:	0.01410,	0.00002,	0.00481,	0.00347,
vel_35_20%-30%:	0.07204,	0.00007,	0.00835,	0.00629,
vel_35_30%-40%:	0.11541,	0.00003,	0.00547,	0.00444,
vel_35_40%-50%:	0.62919,	0.01374,	0.11720,	0.10168,
vel_35_50%-60%:	0.16378,	0.01643,	0.12818,	0.11165,
vel_35_60%-70%:	0.00121,	0.00000,	0.00066,	0.00052,
vel_35_70%-80%:	0.00059,	0.00000,	0.00040,	0.00033,
vel_35_80%-90%:	0.00017,	0.00000,	0.00024,	0.00021,
vel_35_90%-100%:	0.00222,	0.00000,	0.00109,	0.00089,
pop_avg_sum:	0.99872			

vel_45_average:	3499.39248,	285.18780,	16.88750,	13.43859,
vel_45_variance:	486743.98828,	972014141.652,	31177.14133,	22600.33164,
vel_45_std_dev:	697.33846,	514.50444,	22.68269,	16.39126,
vel_45_abs_dev:	502.60297,	382.21825,	19.55040,	16.76612,
vel_45_acorr:	-0.00152,	0.00054,	0.02320,	0.01886,
vel_45_wavg:	3495.52655,	286.30448,	16.92053,	12.99166,
vel_45_wvari:	484148.75921,	1027906102.422,	32060.97476,	23970.75071,
vel_45_w_std_dev:	695.45528,	545.23059,	23.35017,	17.39979,
vel_45_w_abs_dev:	503.81432,	422.93734,	20.56544,	17.44821,
vel_45_sw_avg:	3777.66867,	1360.47314,	36.88459,	26.90591,
vel_45_sw_var:	208993.97186,	310489930.726,	17620.72447,	14537.36929,

```
vel_45_sw_std_dev:      456.80008,      364.06735,      19.08055,       15.76419,
vel_45_sw_abs_dev:      407.52982,      306.76674,      17.51476,       13.64325,
vel_45_upperq:          3889.50000,     44.50000,       6.67083,        5.10000,
vel_45_lowerq:          3071.04000,     5281.48267,     72.67381,       61.92800,
vel_45_rmax:            3071.04000,     5281.48267,     72.67381,       61.92800,
vel_45_rmin:            3071.04000,     5281.48267,     72.67381,       61.92800,
vel_45_0%-10%:          0.00000,        0.00000,        0.00000,        0.00000,
vel_45_10%-20%:         0.02152,        0.00002,        0.00415,        0.00323,
vel_45_20%-30%:         0.07647,        0.00004,        0.00649,        0.00495,
vel_45_30%-40%:         0.10988,        0.00001,        0.00280,        0.00194,
vel_45_40%-50%:         0.64457,        0.00907,        0.09523,        0.07872,
vel_45_50%-60%:         0.14111,        0.00929,        0.09636,        0.08002,
vel_45_60%-70%:         0.00218,        0.00000,        0.00057,        0.00047,
vel_45_70%-80%:         0.00042,        0.00000,        0.00036,        0.00025,
vel_45_80%-90%:         0.00042,        0.00000,        0.00039,        0.00035,
vel_45_90%-100%:        0.00267,        0.00000,        0.00116,        0.00082,
pop_avg_sum:            0.99924

vel_55_average:         3054.61338,     388.75047,      19.71676,       12.74332,
vel_55_variance:        414452.23370,   1204315732.248, 34703.25247,    27829.82467,
vel_55_std_dev:         643.28193,      711.76701,      26.67896,       21.30897,
vel_55_abs_dev:         453.93197,      557.92060,      23.62034,       15.78917,
vel_55_acorr:           -0.01473,       0.00034,        0.01837,        0.01411,
vel_55_wavg:            3051.88099,     437.09557,      20.90683,       13.83860,
vel_55_wvari:           414662.18441,   1510045811.690, 38859.30792,    29508.36090,
vel_55_w_std_dev:       643.32281,      886.61102,      29.77601,       22.65663,
vel_55_w_abs_dev:       455.06548,      632.25638,      25.14471,       16.41058,
vel_55_sw_avg:          3329.85469,     1611.50628,     40.14357,       26.06982,
vel_55_sw_var:          181793.74978,   435080228.765,  20858.57686,    13386.53546,
vel_55_sw_std_dev:      425.78966,      552.12585,      23.49736,       15.21991,
vel_55_sw_abs_dev:      379.42269,      686.96049,      26.20993,       18.03598,
vel_55_upperq:          3436.48000,     18.03733,       4.24704,        3.18400,
vel_55_lowerq:          2671.70000,     8175.70889,     90.41963,       65.92000,
vel_55_rmax:            2671.70000,     8175.70889,     90.41963,       65.92000,
vel_55_rmin:            2671.70000,     8175.70889,     90.41963,       65.92000,
vel_55_0%-10%:          0.00000,        0.00000,        0.00000,        0.00000,
vel_55_10%-20%:         0.02924,        0.00005,        0.00700,        0.00521,
vel_55_20%-30%:         0.08016,        0.00024,        0.01562,        0.01069,
vel_55_30%-40%:         0.10627,        0.00007,        0.00823,        0.00590,
vel_55_40%-50%:         0.58377,        0.00770,        0.08774,        0.07367,
vel_55_50%-60%:         0.19212,        0.01014,        0.10071,        0.08253,
vel_55_60%-70%:         0.00191,        0.00000,        0.00068,        0.00052,
vel_55_70%-80%:         0.00059,        0.00000,        0.00063,        0.00045,
vel_55_80%-90%:         0.00111,        0.00000,        0.00175,        0.00107,
vel_55_90%-100%:        0.00309,        0.00000,        0.00203,        0.00122,
pop_avg_sum:            0.99827

vel_65_average:         2713.10703,     66.10450,       8.13047,        6.13947,
vel_65_variance:        339587.41760,   295704962.717,  17196.07405,    13421.52012,
vel_65_std_dev:         582.57201,      219.18366,      14.80485,       11.57615,
vel_65_abs_dev:         406.12770,      110.97869,      10.53464,       8.34772,
vel_65_acorr:           -0.05335,       0.00037,        0.01934,        0.01383,
vel_65_wavg:            2708.44990,     67.05220,       8.18854,        6.56724,
vel_65_wvari:           339309.60158,   370519237.004,  19248.87625,    15181.85630,
vel_65_w_std_dev:       582.29131,      273.81056,      16.54722,       13.09211,
vel_65_w_abs_dev:       408.71016,      147.26884,      12.13544,       9.08606,
vel_65_sw_avg:          2985.93055,     459.20942,      21.42917,       13.43951,
vel_65_sw_var:          163994.61149,   50959074.59702, 7138.56250,     5917.44288,
vel_65_sw_std_dev:      404.87649,      77.37594,       8.79636,        7.30387,
vel_65_sw_abs_dev:      361.48874,      89.83521,       9.47814,        7.22210,
vel_65_upperq:          3091.60000,     21.82222,       4.67143,        3.60000,
vel_65_lowerq:          2374.14000,     2300.61378,     47.96471,       35.80800,
vel_65_rmax:            2374.14000,     2300.61378,     47.96471,       35.80800,
vel_65_rmin:            2374.14000,     2300.61378,     47.96471,       35.80800,
vel_65_0%-10%:          0.00007,        0.00000,        0.00022,        0.00013,
vel_65_10%-20%:         0.03172,        0.00011,        0.01047,        0.00594,
vel_65_20%-30%:         0.08556,        0.00019,        0.01378,        0.00841,
vel_65_30%-40%:         0.15673,        0.02864,        0.16922,        0.09631,
```

vel_65_40%-50%:	0.50297,	0.02025,	0.14231,	0.08123,
vel_65_50%-60%:	0.21475,	0.01302,	0.11411,	0.09507,
vel_65_60%-70%:	0.00243,	0.00000,	0.00111,	0.00083,
vel_65_70%-80%:	0.00104,	0.00000,	0.00118,	0.00070,
vel_65_80%-90%:	0.00056,	0.00000,	0.00059,	0.00049,
vel_65_90%-100%:	0.00333,	0.00000,	0.00142,	0.00105,
pop_avg_sum:	0.99917			
vel_75_average:	2428.90571,	66.75818,	8.17057,	6.43642,
vel_75_variance:	305301.02878,	139141033.480,	11795.80576,	10338.62197,
vel_75_std_dev:	552.44863,	112.82775,	10.62204,	9.31830,
vel_75_abs_dev:	378.77622,	54.08639,	7.35435,	5.81432,
vel_75_acorr:	-0.07143,	0.00057,	0.02383,	0.01462,
vel_75_wavg:	2422.04062,	74.25348,	8.61705,	6.92479,
vel_75_wvari:	304861.00758,	222380464.316,	14912.42651,	12766.90637,
vel_75_w_std_dev:	551.99507,	180.49770,	13.43494,	11.50574,
vel_75_w_abs_dev:	381.88184,	82.75161,	9.09679,	7.24417,
vel_75_sw_avg:	2708.98264,	81.47401,	9.02630,	6.91509,
vel_75_sw_var:	162416.66265,	92185044.90754,	9601.30433,	7782.15099,
vel_75_sw_std_dev:	402.85234,	140.72947,	11.86295,	9.61016,
vel_75_sw_abs_dev:	360.92645,	113.22283,	10.64062,	8.31082,
vel_75_upperq:	2811.54000,	13.00489,	3.60623,	2.83200,
vel_75_lowerq:	2128.56000,	1202.49600,	34.67702,	27.68000,
vel_75_rmax:	2128.56000,	1202.49600,	34.67702,	27.68000,
vel_75_rmin:	2128.56000,	1202.49600,	34.67702,	27.68000,
vel_75_0%-10%:	0.00000,	0.00000,	0.00000,	0.00000,
vel_75_10%-20%:	0.03895,	0.00002,	0.00484,	0.00405,
vel_75_20%-30%:	0.08079,	0.00005,	0.00680,	0.00565,
vel_75_30%-40%:	0.09226,	0.00004,	0.00626,	0.00544,
vel_75_40%-50%:	0.49969,	0.00310,	0.05568,	0.04254,
vel_75_50%-60%:	0.27743,	0.00360,	0.05999,	0.04594,
vel_75_60%-70%:	0.00345,	0.00000,	0.00069,	0.00054,
vel_75_70%-80%:	0.00164,	0.00000,	0.00044,	0.00036,
vel_75_80%-90%:	0.00056,	0.00000,	0.00064,	0.00053,
vel_75_90%-100%:	0.00432,	0.00000,	0.00103,	0.00080,
pop_avg_sum:	0.99909			
vel_85_average:	2182.42518,	2955.68237,	54.36619,	47.80383,
vel_85_variance:	257799.60089,	994052989.560,	31528.60589,	20549.06836,
vel_85_std_dev:	506.93658,	905.44737,	30.09065,	20.02444,
vel_85_abs_dev:	360.36183,	828.57031,	28.78490,	22.71528,
vel_85_acorr:	-0.07586,	0.00143,	0.03783,	0.03330,
vel_85_wavg:	2175.52220,	3264.14488,	57.13270,	50.68512,
vel_85_wvari:	254119.59752,	935551750.913,	30586.79046,	21629.21012,
vel_85_w_std_dev:	503.31997,	876.22637,	29.60112,	21.29972,
vel_85_w_abs_dev:	361.62122,	982.86316,	31.35065,	25.37681,
vel_85_sw_avg:	2439.94914,	1297.42793,	36.01983,	22.27309,
vel_85_sw_var:	153142.82298,	2333732703.698,	48308.72285,	43412.25675,
vel_85_sw_std_dev:	386.51430,	4166.13310,	64.54559,	58.05426,
vel_85_sw_abs_dev:	345.70758,	6361.60437,	79.75967,	70.59756,
vel_85_upperq:	2561.04000,	18.06044,	4.24976,	3.16800,
vel_85_lowerq:	1700.72000,	15735.46844,	125.44110,	100.49600,
vel_85_rmax:	1700.72000,	15735.46844,	125.44110,	100.49600,
vel_85_rmin:	1700.72000,	15735.46844,	125.44110,	100.49600,
vel_85_0%-10%:	0.00000,	0.00000,	0.00000,	0.00000,
vel_85_10%-20%:	0.01406,	0.00011,	0.01059,	0.00796,
vel_85_20%-30%:	0.11209,	0.00227,	0.04761,	0.04123,
vel_85_30%-40%:	0.10915,	0.00226,	0.04758,	0.04313,
vel_85_40%-50%:	0.42953,	0.02406,	0.15510,	0.08199,
vel_85_50%-60%:	0.28752,	0.01222,	0.11054,	0.07288,
vel_85_60%-70%:	0.03617,	0.01095,	0.10466,	0.05957,
vel_85_70%-80%:	0.00260,	0.00001,	0.00253,	0.00186,
vel_85_80%-90%:	0.00350,	0.00004,	0.00669,	0.00412,
vel_85_90%-100%:	0.00392,	0.00002,	0.00479,	0.00388,
pop_avg_sum:	0.99853			
vel_95_average:	2012.06544,	94.64814,	9.72873,	8.49520,
vel_95_variance:	232276.22946,	54436498.17763,	7378.10939,	5610.85604,

4.1. Motor testing

```
vel_95_std_dev:         481.89567,      58.66224,       7.65913,        5.82142,
vel_95_abs_dev:         329.43377,      50.14465,       7.08129,        5.80418,
vel_95_acorr:           -0.12335,       0.00028,        0.01661,        0.01353,
vel_95_wavg:            2006.40528,     82.92505,       9.10632,        7.56768,
vel_95_wvari:           229762.40622,   75428803.12173, 8684.97571,     6823.63690,
vel_95_w_std_dev:       479.25851,      81.87062,       9.04824,        7.11388,
vel_95_w_abs_dev:       328.80641,      58.48577,       7.64760,        5.64345,
vel_95_sw_avg:          2276.66742,     86.38522,       9.29436,        8.31486,
vel_95_sw_var:          141794.58621,   26623466.17529, 5159.79323,     4115.71868,
vel_95_sw_std_dev:      376.49963,      47.35121,       6.88122,        5.47634,
vel_95_sw_abs_dev:      334.68883,      51.97719,       7.20952,        6.04796,
vel_95_upperq:          2392.50000,     12.72222,       3.56682,        3.00000,
vel_95_lowerq:          1752.94000,     1688.33822,     41.08939,       35.23200,
vel_95_rmax:            1752.94000,     1688.33822,     41.08939,       35.23200,
vel_95_rmin:            1752.94000,     1688.33822,     41.08939,       35.23200,
vel_95_0%-10%:          0.00000,        0.00000,        0.00000,        0.00000,
vel_95_10%-20%:         0.04425,        0.00020,        0.01416,        0.01000,
vel_95_20%-30%:         0.08998,        0.00025,        0.01570,        0.01246,
vel_95_30%-40%:         0.16651,        0.02516,        0.15860,        0.12025,
vel_95_40%-50%:         0.42517,        0.00230,        0.04796,        0.03629,
vel_95_50%-60%:         0.26263,        0.01863,        0.13649,        0.10343,
vel_95_60%-70%:         0.00360,        0.00000,        0.00112,        0.00092,
vel_95_70%-80%:         0.00240,        0.00000,        0.00100,        0.00059,
vel_95_80%-90%:         0.00102,        0.00000,        0.00056,        0.00040,
vel_95_90%-100%:        0.00406,        0.00001,        0.00228,        0.00163,
pop_avg_sum:            0.99965

vel_105_average:        1861.78323,     99.97805,       9.99890,        7.21788,
vel_105_variance:       207436.40246,   118575933.087,  10889.25769,    8022.07312,
vel_105_std_dev:        455.30759,      146.00077,      12.08308,       8.84987,
vel_105_abs_dev:        316.19314,      63.90590,       7.99412,        5.90878,
vel_105_acorr:          -0.15425,       0.00039,        0.01964,        0.01561,
vel_105_wavg:           1856.41119,     103.98252,      10.19718,       7.43059,
vel_105_wvari:          203711.94818,   166356021.546,  12897.90764,    9253.22584,
vel_105_w_std_dev:      451.13250,      212.67942,      14.58353,       10.39312,
vel_105_w_abs_dev:      313.76619,      90.60279,       9.51855,        6.78120,
vel_105_sw_avg:         2121.35884,     175.91298,      13.26322,       10.36804,
vel_105_sw_var:         133281.58693,   48374349.52821, 6955.16711,     5276.14880,
vel_105_sw_std_dev:     364.96640,      90.12858,       9.49361,        7.22038,
vel_105_sw_abs_dev:     327.76194,      68.10952,       8.25285,        6.42857,
vel_105_upperq:         2235.30000,     83.66444,       9.14683,        7.02000,
vel_105_lowerq:         1660.74000,     2095.94711,     45.78151,       34.99200,
vel_105_rmax:           1660.74000,     2095.94711,     45.78151,       34.99200,
vel_105_rmin:           1660.74000,     2095.94711,     45.78151,       34.99200,
vel_105_0%-10%:         0.00000,        0.00000,        0.00000,        0.00000,
vel_105_10%-20%:        0.05005,        0.00028,        0.01683,        0.01340,
vel_105_20%-30%:        0.08552,        0.00020,        0.01415,        0.01168,
vel_105_30%-40%:        0.20729,        0.03661,        0.19135,        0.16626,
vel_105_40%-50%:        0.40881,        0.00319,        0.05652,        0.04890,
vel_105_50%-60%:        0.23651,        0.02581,        0.16064,        0.13953,
vel_105_60%-70%:        0.00303,        0.00000,        0.00130,        0.00107,
vel_105_70%-80%:        0.00321,        0.00000,        0.00148,        0.00106,
vel_105_80%-90%:        0.00073,        0.00000,        0.00060,        0.00052,
vel_105_90%-100%:       0.00432,        0.00001,        0.00316,        0.00259,
pop_avg_sum:            0.99948

vel_115_average         1741.67818,     31.51052,       5.61342,        3.83089,
vel_115_variance:       161816.52922,   23369479.28900, 4834.19893,
    4187.18800,
vel_115_std_dev:        402.22374,      36.21784,       6.01813,        5.21723,
vel_115_abs_dev:        291.46450,      11.25880,       3.35541,        2.84604,
vel_115_acorr:          -0.16180,       0.00041,        0.02014,        0.01493,
vel_115_wavg:           1735.35024,     37.61520,       6.13312,        4.44957,
vel_115_wvari:          158906.57939,   33534679.44651, 5790.91352,     5024.57841,
vel_115_w_std_dev:      398.57132,      52.75706,       7.26341,        6.30349,
vel_115_w_abs_dev:      288.62188,      13.43207,       3.66498,        3.02610,
vel_115_sw_avg:         1980.75901,     266.98750,      16.33975,       13.64803,
vel_115_sw_var:         111682.98707,   62210146.44015, 7887.34090,     6375.43849,
```

vel_115_sw_std_dev:	333.99766,	142.83169,	11.95122,	9.63397,
vel_115_sw_abs_dev:	304.93742,	244.72399,	15.64366,	12.84541,
vel_115_upperq:	2091.00000,	32.00000,	5.65685,	4.40000,
vel_115_lowerq:	1587.94000,	44.92489,	6.70260,	4.62000,
vel_115_rmax:	1587.94000,	44.92489,	6.70260,	4.62000,
vel_115_rmin:	1587.94000,	44.92489,	6.70260,	4.62000,
vel_115_0%-10%:	0.00000,	0.00000,	0.00000,	0.00000,
vel_115_10%-20%:	0.02409,	0.00006,	0.00755,	0.00578,
vel_115_20%-30%:	0.08334,	0.00011,	0.01069,	0.00871,
vel_115_30%-40%:	0.19339,	0.03268,	0.18079,	0.15706,
vel_115_40%-50%:	0.44399,	0.00164,	0.04051,	0.03468,
vel_115_50%-60%:	0.24138,	0.02385,	0.15443,	0.13412,
vel_115_60%-70%:	0.00400,	0.00000,	0.00100,	0.00076,
vel_115_70%-80%:	0.00240,	0.00000,	0.00116,	0.00093,
vel_115_80%-90%:	0.00257,	0.00001,	0.00234,	0.00186,
vel_115_90%-100%:	0.00435,	0.00001,	0.00325,	0.00264,
pop_avg_sum:	0.99951			
vel_125_average:	1633.51313,	86.74454,	9.31367,	6.83926,
vel_125_variance:	142997.26069,	34048697.75200,	5835.12620,	4123.34306,
vel_125_std_dev:	378.07886,	59.59444,	7.71974,	5.45461,
vel_125_abs_dev:	284.53501,	11.11109,	3.33333,	2.31107,
vel_125_acorr:	-0.17668,	0.00047,	0.02166,	0.01653,
vel_125_wavg:	1626.41608,	109.86242,	10.48153,	7.41902,
vel_125_wvari:	139695.63098,	52811547.30770,	7267.15538,	5346.50781,
vel_125_w_std_dev:	373.64558,	94.01388,	9.69608,	7.14923,
vel_125_w_abs_dev:	281.39170,	19.69058,	4.43741,	3.44700,
vel_125_sw_avg:	1855.17190,	1050.27500,	32.40795,	24.40840,
vel_125_sw_var:	102672.71297,	132505536.326,	11511.10491,	8461.89233,
vel_125_sw_std_dev:	319.93622,	348.36372,	18.66450,	13.77135,
vel_125_sw_abs_dev:	292.18601,	700.00158,	26.45754,	19.84018,
vel_125_upperq:	1974.42000,	57.58622,	7.58856,	6.09600,
vel_125_lowerq:	1463.90000,	60.95333,	7.80726,	5.92000,
vel_125_rmax:	1514.50000,	25563.35333,	159.88544,	90.90000,
vel_125_rmin:	1463.90000,	60.95333,	7.80726,	5.92000,
vel_125_0%-10%:	0.00000,	0.00000,	0.00000,	0.00000,
vel_125_10%-20%:	0.03449,	0.00123,	0.03505,	0.02621,
vel_125_20%-30%:	0.14856,	0.02458,	0.15678,	0.11492,
vel_125_30%-40%:	0.22686,	0.03775,	0.19430,	0.17423,
vel_125_40%-50%:	0.34826,	0.03490,	0.18682,	0.14330,
vel_125_50%-60%:	0.22752,	0.02613,	0.16165,	0.14288,
vel_125_60%-70%:	0.00510,	0.00000,	0.00209,	0.00168,
vel_125_70%-80%:	0.00274,	0.00000,	0.00132,	0.00102,
vel_125_80%-90%:	0.00184,	0.00001,	0.00233,	0.00146,
vel_125_90%-100%:	0.00375,	0.00001,	0.00340,	0.00300,
pop_avg_sum:	0.99913			
vel_135_average:	1532.72012,	67.07508,	8.18994,	6.27668,
vel_135_variance:	123029.09965,	31794503.78700,	5638.66152,	4818.46716,
vel_135_std_dev:	350.67326,	63.73485,	7.98341,	6.83301,
vel_135_abs_dev:	276.35595,	20.30766,	4.50640,	3.55556,
vel_135_acorr:	-0.21262,	0.00028,	0.01666,	0.01222,
vel_135_wavg:	1527.77810,	62.46333,	7.90337,	6.32795,
vel_135_wvari:	119575.62958,	20308941.72793,	4506.54432,	3866.84186,
vel_135_w_std_dev:	345.74242,	42.00914,	6.48145,	5.56734,
vel_135_w_abs_dev:	273.33532,	13.52193,	3.67722,	2.91622,
vel_135_sw_avg:	1706.47582,	3283.40543,	57.30101,	43.46726,
vel_135_sw_var:	86156.28354,	169756546.462,	13029.06545,	9350.32670,
vel_135_sw_std_dev:	292.68240,	548.10570,	23.41166,	16.89292,
vel_135_sw_abs_dev:	264.16191,	1130.83606,	33.62791,	24.46874,
vel_135_upperq:	1868.50000,	42.19778,	6.49598,	4.76000,
vel_135_lowerq:	1343.74000,	86.23156,	9.28609,	6.80800,
vel_135_rmax:	1448.22000,	47431.017,	217.78663,	165.15200,
vel_135_rmin:	1343.74000,	86.23156,	9.28609,	6.80800,
vel_135_0%-10%:	0.00000,	0.00000,	0.00000,	0.00000,
vel_135_10%-20%:	0.00879,	0.00006,	0.00751,	0.00603,
vel_135_20%-30%:	0.07406,	0.00012,	0.01113,	0.00805,
vel_135_30%-40%:	0.24894,	0.02635,	0.16233,	0.14406,

```
vel_135_40%-50%:        0.34897,        0.01393,        0.11803,        0.09763,
vel_135_50%-60%:        0.29322,        0.01118,        0.10574,        0.07942,
vel_135_60%-70%:        0.00659,        0.00000,        0.00136,        0.00115,
vel_135_70%-80%:        0.00275,        0.00000,        0.00136,        0.00102,
vel_135_80%-90%:        0.00251,        0.00000,        0.00212,        0.00158,
vel_135_90%-100%:       0.00474,        0.00001,        0.00275,        0.00212,
pop_avg_sum:            0.99059

vel_145_average:        1446.66720,     63.21699,       7.95091,        6.31824,
vel_145_variance:       110318.81793,   19159982.96425, 4377.21178,     3100.37111,
vel_145_std_dev:        332.08272,      44.31912,       6.65726,        4.68773,
vel_145_abs_dev:        269.52518,      8.04861,        2.83701,        2.17041,
vel_145_acorr:          -0.26681,       0.00188,        0.04333,        0.03494,
vel_145_wavg:           1441.62444,     73.72240,       8.58617,        6.82541,
vel_145_wvari:          106100.86842,   23670593.66258, 4865.24343,     3558.84513,
vel_145_w_std_dev:      325.65349,      56.30579,       7.50372,        5.47319,
vel_145_w_abs_dev:      265.86950,      9.66837,        3.10940,        2.68040,
vel_145_sw_avg:         1582.93544,     1322.85443,     36.37107,       27.11798,
vel_145_sw_var:         76906.19123,    85393284.14065, 9240.84867,     7011.39337,
vel_145_sw_std_dev:     276.88725,      266.27101,      16.31781,       12.45623,
vel_145_sw_abs_dev:     247.14899,      463.82172,      21.53652,       16.69153,
vel_145_upperq:         1779.90000,     12.32222,       3.51030,        2.90000,
vel_145_lowerq:         1245.92000,     21.31733,       4.61707,        3.81600,
vel_145_rmax:           1619.86000,     66618.596,      258.10578,      224.39600,
vel_145_rmin:           1245.92000,     21.31733,       4.61707,        3.81600,
vel_145_0%-10%:         0.00000,        0.00000,        0.00000,        0.00000,
vel_145_10%-20%:        0.00764,        0.00027,        0.01633,        0.01194,
vel_145_20%-30%:        0.09847,        0.00795,        0.08914,        0.05196,
vel_145_30%-40%:        0.27955,        0.00971,        0.09856,        0.06652,
vel_145_40%-50%:        0.32278,        0.00245,        0.04945,        0.03714,
vel_145_50%-60%:        0.27353,        0.02026,        0.14233,        0.10764,
vel_145_60%-70%:        0.00652,        0.00001,        0.00280,        0.00217,
vel_145_70%-80%:        0.00403,        0.00000,        0.00144,        0.00115,
vel_145_80%-90%:        0.00128,        0.00000,        0.00075,        0.00057,
vel_145_90%-100%:       0.00500,        0.00001,        0.00306,        0.00237,
pop_avg_sum:            0.99879

vel_155_average:        1380.41765,     205.94839,      14.35090,       9.08944,
vel_155_variance:       107935.18295,   28981621.71397, 5383.45816,     4400.25251,
vel_155_std_dev:        328.44183,      67.94186,       8.24269,        6.73571,
vel_155_abs_dev:        269.33016,      14.26117,       3.77640,        3.09090,
vel_155_acorr:          -0.25197,       0.00245,        0.04953,        0.03775,
vel_155_wavg:           1374.22760,     234.03148,      15.29809,       9.53708,
vel_155_wvari:          104297.27028,   25916403.62093, 5090.81561,     4276.52349,
vel_155_w_std_dev:      322.86369,      62.56768,       7.90997,        6.65203,
vel_155_w_abs_dev:      265.98115,      14.21652,       3.77048,        3.14788,
vel_155_sw_avg:         1497.52570,     2576.83593,     50.76254,       32.99638,
vel_155_sw_var:         72092.88645,    190754615.541,  13811.39441,    10391.78092,
vel_155_sw_std_dev:     267.49381,      599.94506,      24.49378,       18.55244,
vel_155_sw_abs_dev:     237.36589,      724.27873,      26.91243,       19.70263,
vel_155_upperq:         1707.46000,     17.09378,       4.13446,        3.15200,
vel_155_lowerq:         1168.20000,     31.06667,       5.57375,        3.92000,
vel_155_rmax:           1545.76000,     68106.39822,    260.97203,      226.85600,
vel_155_rmin:           1168.20000,     31.06667,       5.57375,        3.92000,
vel_155_0%-10%:         0.00000,        0.00000,        0.00000,        0.00000,
vel_155_10%-20%:        0.01322,        0.00047,        0.02172,        0.01851,
vel_155_20%-30%:        0.15004,        0.03034,        0.17417,        0.13087,
vel_155_30%-40%:        0.33874,        0.00333,        0.05769,        0.04364,
vel_155_40%-50%:        0.23977,        0.00721,        0.08490,        0.05368,
vel_155_50%-60%:        0.23844,        0.02566,        0.16017,        0.13921,
vel_155_60%-70%:        0.00744,        0.00001,        0.00265,        0.00217,
vel_155_70%-80%:        0.00440,        0.00000,        0.00221,        0.00161,
vel_155_80%-90%:        0.00129,        0.00000,        0.00116,        0.00090,
vel_155_90%-100%:       0.00584,        0.00002,        0.00455,        0.00350,
pop_avg_sum:            0.99918

vel_165_average:        1320.57195,     83.70411,       9.14899,        8.10812,
vel_165_variance:       96620.56038,    14758205.86362, 3841.64104,     3160.74149,
```

4.1. Motor testing

vel_165_std_dev:	310.78280,	38.45863,	6.20150,	5.10426,
vel_165_abs_dev:	260.81325,	6.60402,	2.56983,	2.12541,
vel_165_acorr:	-0.31138,	0.00087,	0.02952,	0.02046,
vel_165_wavg:	1314.77926,	85.07830,	9.22379,	7.96810,
vel_165_wvari:	92312.06669,	11851017.68911,	3442.53071,	2865.64549,
vel_165_w_std_dev:	303.78128,	32.22438,	5.67665,	4.72053,
vel_165_w_abs_dev:	257.15208,	6.99321,	2.64447,	2.22532,
vel_165_sw_avg:	1427.15441,	2855.19565,	53.43403,	32.42856,
vel_165_sw_var:	65998.36167,	120688574.784,	10985.83519,	6524.23474,
vel_165_sw_std_dev:	256.20993,	394.25788,	19.85593,	12.02162,
vel_165_sw_abs_dev:	224.46016,	613.99080,	24.77884,	13.80120,
vel_165_upperq:	1643.70000,	106.45556,	10.31773,	9.24000,
vel_165_lowerq:	1102.16000,	133.14489,	11.53884,	10.27200,
vel_165_rmax:	1372.66000,	81208.35600,	284.97080,	270.14000,
vel_165_rmin:	1102.16000,	133.14489,	11.53884,	10.27200,
vel_165_0%-10%:	0.00000,	0.00000,	0.00000,	0.00000,
vel_165_10%-20%:	0.01025,	0.00029,	0.01703,	0.01435,
vel_165_20%-30%:	0.16783,	0.03703,	0.19244,	0.16081,
vel_165_30%-40%:	0.38665,	0.01057,	0.10282,	0.08529,
vel_165_40%-50%:	0.18598,	0.01087,	0.10425,	0.09187,
vel_165_50%-60%:	0.23137,	0.02475,	0.15733,	0.13563,
vel_165_60%-70%:	0.00807,	0.00001,	0.00361,	0.00257,
vel_165_70%-80%:	0.00340,	0.00000,	0.00207,	0.00159,
vel_165_80%-90%:	0.00232,	0.00001,	0.00257,	0.00198,
vel_165_90%-100%:	0.00351,	0.00001,	0.00346,	0.00307,
pop_avg_sum:	0.99938			

vel_175_average:	1258.92679,	111.43183,	10.55613,	8.57073,
vel_175_variance:	91653.16728,	9617404.22874,	3101.19400,	2351.25177,
vel_175_std_dev:	302.70389,	26.13390,	5.11213,	3.87328,
vel_175_abs_dev:	257.46562,	12.27703,	3.50386,	2.97546,
vel_175_acorr:	-0.31877,	0.00083,	0.02879,	0.01904,
vel_175_wavg:	1252.71826,	131.27907,	11.45771,	9.15741,
vel_175_wvari:	87832.16336,	12084673.28916,	3476.30167,	2764.07538,
vel_175_w_std_dev:	296.31263,	34.43071,	5.86777,	4.67609,
vel_175_w_abs_dev:	253.98523,	15.12263,	3.88878,	3.23099,
vel_175_sw_avg:	1389.36188,	2926.57347,	54.09781,	46.34538,
vel_175_sw_var:	70801.66798,	150995764.847,	12288.03340,	10061.33012,
vel_175_sw_std_dev:	265.18240,	533.29454,	23.09317,	18.99480,
vel_175_sw_abs_dev:	237.70661,	1126.60919,	33.56500,	27.51150,
vel_175_upperq:	1577.62000,	116.12844,	10.77629,	8.14000,
vel_175_lowerq:	1030.80000,	114.62222,	10.70618,	8.00000,
vel_175_rmax:	1140.10000,	53027.21111,	230.27638,	174.56000,
vel_175_rmin:	1030.80000,	114.62222,	10.70618,	8.00000,
vel_175_0%-10%:	0.00000,	0.00000,	0.00000,	0.00000,
vel_175_10%-20%:	0.00224,	0.00002,	0.00404,	0.00300,
vel_175_20%-30%:	0.10019,	0.00649,	0.08054,	0.05770,
vel_175_30%-40%:	0.48439,	0.00745,	0.08633,	0.07250,
vel_175_40%-50%:	0.23011,	0.01656,	0.12868,	0.11530,
vel_175_50%-60%:	0.16148,	0.02291,	0.15136,	0.14112,
vel_175_60%-70%:	0.00928,	0.00012,	0.01077,	0.00596,
vel_175_70%-80%:	0.00559,	0.00001,	0.00248,	0.00208,
vel_175_80%-90%:	0.00338,	0.00001,	0.00275,	0.00229,
vel_175_90%-100%:	0.00250,	0.00001,	0.00332,	0.00292,
pop_avg_sum:	0.99916			

vel_185_average:	1212.81185,	281.84707,	16.78830,	10.28495,
vel_185_variance:	88353.56405,	18258982.59393,	4273.05308,	3437.19521,
vel_185_std_dev:	297.16475,	51.86173,	7.20151,	5.78538,
vel_185_abs_dev:	253.91965,	7.62559,	2.76145,	2.34593,
vel_185_acorr:	-0.33115,	0.00189,	0.04352,	0.03650,
vel_185_wavg:	1206.94164,	324.78007,	18.02166,	10.84388,
vel_185_wvari:	84427.33697,	14027953.71024,	3745.39100,	2838.89579,
vel_185_w_std_dev:	290.49963,	41.44699,	6.43793,	4.88285,
vel_185_w_abs_dev:	250.11751,	7.56170,	2.74985,	2.50220,
vel_185_sw_avg:	1331.20954,	3893.43470,	62.39739,	51.54375,
vel_185_sw_var:	65762.09146,	231238584.169,	15206.53097,	11826.14228,
vel_185_sw_std_dev:	255.04585,	793.00575,	28.16036,	22.05251,

vel_185_sw_abs_dev:	222.13044,	1550.68097,	39.37869,	30.78861,
vel_185_upperq:	1534.00000,	302.00000,	17.37815,	11.60000,
vel_185_lowerq:	986.40000,	325.82222,	18.05055,	11.80000,
vel_185_rmax:	1096.10000,	57582.76667,	239.96409,	181.76000,
vel_185_rmin:	986.40000,	325.82222,	18.05055,	11.80000,
vel_185_0%-10%:	0.00000,	0.00000,	0.00000,	0.00000,
vel_185_10%-20%:	0.00690,	0.00024,	0.01557,	0.01104,
vel_185_20%-30%:	0.15227,	0.05007,	0.22377,	0.16848,
vel_185_30%-40%:	0.44683,	0.02227,	0.14925,	0.12292,
vel_185_40%-50%:	0.15194,	0.01330,	0.11533,	0.10036,
vel_185_50%-60%:	0.21124,	0.02010,	0.14179,	0.11816,
vel_185_60%-70%:	0.01926,	0.00118,	0.03435,	0.01944,
vel_185_70%-80%:	0.00425,	0.00000,	0.00183,	0.00127,
vel_185_80%-90%:	0.00191,	0.00000,	0.00185,	0.00121,
vel_185_90%-100%:	0.00457,	0.00002,	0.00415,	0.00359,
pop_avg_sum:	0.99917			

vel_195_average:	1178.58701,	302.13175,	17.38194,	13.52952,
vel_195_variance:	84698.62536,	7979041.65585,	2824.71975,	2100.78418,
vel_195_std_dev:	290.99343,	23.83288,	4.88189,	3.61756,
vel_195_abs_dev:	252.57385,	7.20774,	2.68472,	2.19848,
vel_195_acorr:	-0.36355,	0.00106,	0.03254,	0.02667,
vel_195_wavg:	1172.74031,	321.67364,	17.93526,	14.16908,
vel_195_wvari:	81170.26380,	11234519.73567,	3351.79351,	2583.97338,
vel_195_w_std_dev:	284.84836,	35.19778,	5.93277,	4.56241,
vel_195_w_abs_dev:	249.02110,	8.81787,	2.96949,	2.45608,
vel_195_sw_avg:	1299.42157,	6870.71273,	82.88976,	57.06380,
vel_195_sw_var:	66229.25582,	341117984.565,	18469.37965,	13391.92359,
vel_195_sw_std_dev:	255.34382,	1143.09907,	33.80975,	24.14821,
vel_195_sw_abs_dev:	221.49665,	1919.24397,	43.80918,	32.59289,
vel_195_upperq:	1498.64000,	233.47378,	15.27985,	11.48800,
vel_195_lowerq:	947.70000,	276.45556,	16.62695,	13.24000,
vel_195_rmax:	1057.10000,	59144.32222,	243.19606,	184.36000,
vel_195_rmin:	947.70000,	276.45556,	16.62695,	13.24000,
vel_195_0%-10%:	0.00000,	0.00000,	0.00000,	0.00000,
vel_195_10%-20%:	0.00472,	0.00010,	0.01003,	0.00755,
vel_195_20%-30%:	0.14358,	0.05771,	0.24022,	0.18223,
vel_195_30%-40%:	0.47767,	0.01268,	0.11260,	0.10435,
vel_195_40%-50%:	0.07706,	0.00384,	0.06197,	0.03595,
vel_195_50%-60%:	0.25177,	0.01706,	0.13062,	0.09852,
vel_195_60%-70%:	0.01506,	0.00016,	0.01267,	0.00931,
vel_195_70%-80%:	0.00323,	0.00000,	0.00215,	0.00180,
vel_195_80%-90%:	0.00309,	0.00001,	0.00282,	0.00214,
vel_195_90%-100%:	0.00419,	0.00001,	0.00353,	0.00312,
pop_avg_sum:	0.98038			

	ENC 2: AVERAGE	VARIANCE	STND_DEV	ABS_DEV
vel_15_average:	7206.62676,	4496.65708,	67.05712,	41.84828,
vel_15_variance:	1824269.59182,	14591025282,	120793.31638,	100022.43843,
vel_15_std_dev:	1349.98395,	2014.35912,	44.88161,	37.23284,
vel_15_abs_dev:	1030.57870,	499.86553,	22.35767,	18.91822,
vel_15_acorr:	-0.22421,	0.00325,	0.05704,	0.04135,
vel_15_wavg:	7180.32702,	3031.95704,	55.06321,	34.57643,
vel_15_wvari:	1707451.09165,	6447067822.042,	80293.63500,	66582.88464,
vel_15_w_std_dev:	1306.37056,	941.15722,	30.67829,	25.46719,
vel_15_w_abs_dev:	1015.46953,	361.00837,	19.00022,	15.91612,
vel_15_sw_avg:	7908.56275,	5789.13713,	76.08638,	56.21413,
vel_15_sw_var:	1351548.19640,	2518794016.527,	50187.58827,	42135.27694,
vel_15_sw_std_dev:	1162.37918,	469.81556,	21.67523,	18.15287,
vel_15_sw_abs_dev:	990.09430,	2553.00303,	50.52725,	42.58292,
vel_15_upperq:	8367.96000,	5466.07822,	73.93293,	45.77600,
vel_15_lowerq:	6363.16000,	4089.90933,	63.95240,	46.37600,
vel_15_rmax:	6363.16000,	4089.90933,	63.95240,	46.37600,
vel_15_rmin:	6363.16000,	4089.90933,	63.95240,	46.37600,
vel_15_0%-10%:	0.00000,	0.00000,	0.00000,	0.00000,

vel_15_10%-20%:	0.00702,	0.00013,	0.01136,	0.00879,
vel_15_20%-30%:	0.10602,	0.00835,	0.09137,	0.06878,
vel_15_30%-40%:	0.32695,	0.03403,	0.18447,	0.15125,
vel_15_40%-50%:	0.38132,	0.02874,	0.16954,	0.12710,
vel_15_50%-60%:	0.17066,	0.01518,	0.12320,	0.10065,
vel_15_60%-70%:	0.00351,	0.00000,	0.00114,	0.00094,
vel_15_70%-80%:	0.00174,	0.00000,	0.00080,	0.00063,
vel_15_80%-90%:	0.00083,	0.00000,	0.00074,	0.00063,
vel_15_90%-100%:	0.00136,	0.00000,	0.00108,	0.00087,
pop_avg_sum:	0.99941			

vel_25_average:	4999.02823,	505.96929,	22.49376,	18.32174,
vel_25_variance:	878812.89812,	2477145059.205,	49770.92584,	42885.54143,
vel_25_std_dev:	937.10988,	708.85539,	26.62434,	22.98754,
vel_25_abs_dev:	671.92801,	126.75795,	11.25868,	9.00865,
vel_25_acorr:	-0.18490,	0.00087,	0.02945,	0.02143,
vel_25_wavg:	4991.82538,	558.47817,	23.63214,	19.59447,
vel_25_wvari:	876394.13713,	2847247196.358,	53359.60266,	43371.76184,
vel_25_w_std_dev:	935.76600,	817.92697,	28.59942,	23.29944,
vel_25_w_abs_dev:	673.00942,	178.93781,	13.37676,	10.88943,
vel_25_sw_avg:	5519.52376,	524.47543,	22.90143,	17.69127,
vel_25_sw_var:	560114.96761,	507323470.261,	22523.84226,	15297.89398,
vel_25_sw_std_dev:	748.27142,	227.61131,	15.08679,	10.20416,
vel_25_sw_abs_dev:	662.11296,	192.82392,	13.88611,	11.36551,
vel_25_upperq:	5795.68000,	242.65956,	15.57753,	11.70400,
vel_25_lowerq:	4411.36000,	966.73600,	31.09238,	25.08800,
vel_25_rmax:	4411.36000,	966.73600,	31.09238,	25.08800,
vel_25_rmin:	4411.36000,	966.73600,	31.09238,	25.08800,
vel_25_0%-10%:	0.00163,	0.00000,	0.00112,	0.00080,
vel_25_10%-20%:	0.00351,	0.00000,	0.00142,	0.00109,
vel_25_20%-30%:	0.05647,	0.00003,	0.00507,	0.00357,
vel_25_30%-40%:	0.12741,	0.00008,	0.00898,	0.00772,
vel_25_40%-50%:	0.42581,	0.00008,	0.00909,	0.00727,
vel_25_50%-60%:	0.37890,	0.00002,	0.00494,	0.00401,
vel_25_60%-70%:	0.00216,	0.00000,	0.00039,	0.00029,
vel_25_70%-80%:	0.00094,	0.00000,	0.00047,	0.00038,
vel_25_80%-90%:	0.00063,	0.00000,	0.00046,	0.00029,
vel_25_90%-100%:	0.00216,	0.00000,	0.00100,	0.00078,
pop_avg_sum:	0.99962			

vel_35_average:	4190.48662,	19630.01311,	140.10715,	105.28216,
vel_35_variance:	663882.71753,	973956187.348,	31208.27114,	21146.39966,
vel_35_std_dev:	814.58973,	362.54649,	19.04065,	12.90982,
vel_35_abs_dev:	574.69910,	209.94315,	14.48942,	9.71090,
vel_35_acorr:	-0.11225,	0.00072,	0.02677,	0.02062,
vel_35_wavg:	4183.08557,	19014.29657,	137.89234,	103.62152,
vel_35_wvari:	664974.48124,	1194944579.809,	34567.97043,	26883.10737,
vel_35_w_std_dev:	815.21231,	448.18258,	21.17032,	16.41873,
vel_35_w_abs_dev:	578.63680,	273.02407,	16.52344,	12.32494,
vel_35_sw_avg:	4614.35621,	28641.25926,	169.23729,	122.19091,
vel_35_sw_var:	387984.81527,	1476110614.770,	38420.18499,	24443.76646,
vel_35_sw_std_dev:	622.16281,	998.05452,	31.59200,	20.07710,
vel_35_sw_abs_dev:	545.03025,	2553.71661,	50.53431,	29.15172,
vel_35_upperq:	4830.64000,	26098.03378,	161.54886,	122.50400,
vel_35_lowerq:	3709.34000,	17760.80044,	133.26965,	99.39600,
vel_35_rmax:	3709.34000,	17760.80044,	133.26965,	99.39600,
vel_35_rmin:	3709.34000,	17760.80044,	133.26965,	99.39600,
vel_35_0%-10%:	0.00073,	0.00000,	0.00125,	0.00075,
vel_35_10%-20%:	0.01472,	0.00025,	0.01566,	0.00875,
vel_35_20%-30%:	0.07603,	0.00108,	0.03282,	0.01849,
vel_35_30%-40%:	0.18060,	0.04118,	0.20292,	0.11543,
vel_35_40%-50%:	0.38678,	0.01815,	0.13471,	0.07659,
vel_35_50%-60%:	0.33557,	0.01375,	0.11725,	0.06663,
vel_35_60%-70%:	0.00195,	0.00000,	0.00107,	0.00074,
vel_35_70%-80%:	0.00087,	0.00000,	0.00050,	0.00038,
vel_35_80%-90%:	0.00028,	0.00000,	0.00022,	0.00017,
vel_35_90%-100%:	0.00195,	0.00000,	0.00090,	0.00065,
pop_avg_sum:	0.99948			

4.1. Motor testing

```
vel_45_average:         3519.17518,     77.01236,       8.77567,        6.72142,
vel_45_variance:        498950.02084,   653429443.184,  25562.26600,    21217.27985,
vel_45_std_dev:         706.15438,      328.90422,      18.13572,       14.98785,
vel_45_abs_dev:         501.95413,      108.12845,      10.39848,       7.88187,
vel_45_acorr:           -0.06478,       0.00062,        0.02491,        0.02026,
vel_45_wavg:            3513.55834,     71.70724,       8.46801,        6.41218,
vel_45_wvari:           498136.80014,   449467928.781,  21200.65869,    17061.75601,
vel_45_w_std_dev:       705.64418,      225.66162,      15.02204,       12.08974,
vel_45_w_abs_dev:       505.76130,      63.72456,       7.98277,        5.92927,
vel_45_sw_avg:          3888.79824,     71.50558,       8.45610,        6.45496,
vel_45_sw_var:          272457.51825,   58264629.469,   7633.12711,     5747.70775,
vel_45_sw_std_dev:      521.92819,      53.87073,       7.33967,        5.52457,
vel_45_sw_abs_dev:      473.56131,      53.15902,       7.29102,        5.65622,
vel_45_upperq:          4022.10000,     39.82444,       6.31066,        5.22000,
vel_45_lowerq:          3121.78000,     1421.76400,     37.70629,       30.50400,
vel_45_rmax:            3121.78000,     1421.76400,     37.70629,       30.50400,
vel_45_rmin:            3121.78000,     1421.76400,     37.70629,       30.50400,
vel_45_0%-10%:          0.00146,        0.00000,        0.00101,        0.00084,
vel_45_10%-20%:         0.01346,        0.00001,        0.00254,        0.00185,
vel_45_20%-30%:         0.09053,        0.00005,        0.00740,        0.00571,
vel_45_30%-40%:         0.09536,        0.00003,        0.00540,        0.00404,
vel_45_40%-50%:         0.44758,        0.00046,        0.02145,        0.01681,
vel_45_50%-60%:         0.34514,        0.00054,        0.02333,        0.01786,
vel_45_60%-70%:         0.00212,        0.00000,        0.00042,        0.00032,
vel_45_70%-80%:         0.00077,        0.00000,        0.00054,        0.00049,
vel_45_80%-90%:         0.00007,        0.00000,        0.00015,        0.00011,
vel_45_90%-100%:        0.00303,        0.00000,        0.00114,        0.00087,
pop_avg_sum:            0.99951

vel_55_average:         3089.81646,     299.57463,      17.30822,       15.88236,
vel_55_variance:        367230.32503,   247289350.576,  15725.43642,    12614.64631,
vel_55_std_dev:         605.86842,      170.86643,      13.07159,       10.44341,
vel_55_abs_dev:         423.69196,      227.59755,      15.08634,       13.60471,
vel_55_acorr:           -0.03113,       0.00046,        0.02146,        0.01502,
vel_55_wavg:            3086.11207,     309.59520,      17.59532,       16.21779,
vel_55_wvari:           367360.92734,   439551950.054,  20965.49427,    16345.84024,
vel_55_w_std_dev:       605.87695,      304.50403,      17.45004,       13.58581,
vel_55_w_abs_dev:       425.29650,      254.77308,      15.96161,       14.03431,
vel_55_sw_avg:          3376.00349,     1240.14245,     35.21566,       20.67123,
vel_55_sw_var:          175789.78544,   311346769.703,  17645.02110,    14318.89384,
vel_55_sw_std_dev:      418.78412,      455.16645,      21.33463,       17.16114,
vel_55_sw_abs_dev:      377.13118,      1189.43222,     34.48815,       21.98577,
vel_55_upperq:          3467.00000,     21.11111,       4.59468,        3.20000,
vel_55_lowerq:          2773.80000,     3916.19556,     62.57951,       54.08000,
vel_55_rmax:            2773.80000,     3916.19556,     62.57951,       54.08000,
vel_55_rmin:            2773.80000,     3916.19556,     62.57951,       54.08000,
vel_55_0%-10%:          0.00139,        0.00000,        0.00202,        0.00111,
vel_55_10%-20%:         0.01560,        0.00029,        0.01705,        0.00966,
vel_55_20%-30%:         0.10347,        0.00043,        0.02078,        0.01405,
vel_55_30%-40%:         0.15331,        0.04584,        0.21411,        0.12182,
vel_55_40%-50%:         0.46489,        0.02747,        0.16575,        0.09551,
vel_55_50%-60%:         0.25331,        0.00917,        0.09577,        0.05715,
vel_55_60%-70%:         0.00198,        0.00000,        0.00112,        0.00083,
vel_55_70%-80%:         0.00139,        0.00000,        0.00064,        0.00042,
vel_55_80%-90%:         0.00024,        0.00000,        0.00023,        0.00019,
vel_55_90%-100%:        0.00323,        0.00000,        0.00151,        0.00108,
pop_avg_sum:            0.99882

vel_65_average:         2760.58723,     97.04502,       9.85114,        7.97685,
vel_65_variance:        260697.56546,   132490324.220,  11510.44414,    9627.49598,
vel_65_std_dev:         510.47330,      127.31097,      11.28322,       9.43254,
vel_65_abs_dev:         350.10133,      82.85421,       9.10243,        7.20228,
vel_65_acorr:           0.01381,        0.00018,        0.01339,        0.00924,
vel_65_wavg:            2756.66361,     138.62348,      11.77385,       9.03522,
vel_65_wvari:           260383.89020,   246616599.224,  15704.03130,    13508.16742,
vel_65_w_std_dev:       510.06854,      237.74994,      15.41914,       13.25209,
vel_65_w_abs_dev:       351.26660,      127.49819,      11.29151,       8.86017,
```

4.1. Motor testing

```
vel_65_sw_avg:          2968.99157,     1584.66897,     39.80790,       29.69225,
vel_65_sw_var:          117703.79819,   214695488.375,  14652.49086,    11755.27068,
vel_65_sw_std_dev:      342.45976,      472.34624,      21.73353,       17.27120,
vel_65_sw_abs_dev:      301.54101,      627.15316,      25.04303,       17.45614,
vel_65_upperq:          3035.84000,     26.39822,       5.13792,        4.35200,
vel_65_lowerq:          2563.04000,     652.30933,      25.54035,       19.08000,
vel_65_rmax:            2563.04000,     652.30933,      25.54035,       19.08000,
vel_65_rmin:            2563.04000,     652.30933,      25.54035,       19.08000,
vel_65_0%-10%:          0.00000,        0.00000,        0.00000,        0.00000,
vel_65_10%-20%:         0.01039,        0.00038,        0.01948,        0.01106,
vel_65_20%-30%:         0.10177,        0.00052,        0.02287,        0.01587,
vel_65_30%-40%:         0.14987,        0.04726,        0.21740,        0.12373,
vel_65_40%-50%:         0.57753,        0.04874,        0.22077,        0.13547,
vel_65_50%-60%:         0.15145,        0.01201,        0.10959,        0.09306,
vel_65_60%-70%:         0.00240,        0.00000,        0.00122,        0.00081,
vel_65_70%-80%:         0.00108,        0.00000,        0.00064,        0.00052,
vel_65_80%-90%:         0.00042,        0.00000,        0.00043,        0.00036,
vel_65_90%-100%:        0.00399,        0.00000,        0.00186,        0.00129,
pop_avg_sum:            0.99889

vel_75_average:         2480.65201,     70.94506,       8.42289,        6.88166,
vel_75_variance:        210038.60063,   118265534.725,  10874.99585,    7943.72040,
vel_75_std_dev:         458.16486,      137.29226,      11.71718,       8.59459,
vel_75_abs_dev:         308.74809,      31.06861,       5.57392,        4.19990,
vel_75_acorr:           0.03815,        0.00053,        0.02307,        0.01740,
vel_75_wavg:            2474.77840,     67.31099,       8.20433,        7.20675,
vel_75_wvari:           205429.93981,   153592021.909,  12393.22484,    10381.91415,
vel_75_w_std_dev:       453.06027,      184.81185,      13.59455,       11.43172,
vel_75_w_abs_dev:       308.71642,      27.07958,       5.20380,        4.22639,
vel_75_sw_avg:          2642.30398,     879.41778,      29.65498,       25.59434,
vel_75_sw_var:          83354.97200,    201571808.375,  14197.59868,    10649.80846,
vel_75_sw_std_dev:      287.73029,      629.17230,      25.08331,       18.99286,
vel_75_sw_abs_dev:      248.31747,      697.03313,      26.40138,       21.38109,
vel_75_upperq:          2700.22000,     5.41733,        2.32752,        1.66800,
vel_75_lowerq:          2348.38000,     148.33289,      12.17920,       10.17600,
vel_75_rmax:            2348.38000,     148.33289,      12.17920,       10.17600,
vel_75_rmin:            2348.38000,     148.33289,      12.17920,       10.17600,
vel_75_0%-10%:          0.00003,        0.00000,        0.00011,        0.00006,
vel_75_10%-20%:         0.00608,        0.00001,        0.00347,        0.00274,
vel_75_20%-30%:         0.10283,        0.00071,        0.02664,        0.02273,
vel_75_30%-40%:         0.16718,        0.04629,        0.21514,        0.12094,
vel_75_40%-50%:         0.64502,        0.04476,        0.21157,        0.11752,
vel_75_50%-60%:         0.06885,        0.00465,        0.06817,        0.06076,
vel_75_60%-70%:         0.00242,        0.00000,        0.00073,        0.00062,
vel_75_70%-80%:         0.00194,        0.00000,        0.00148,        0.00087,
vel_75_80%-90%:         0.00169,        0.00001,        0.00261,        0.00195,
vel_75_90%-100%:        0.00287,        0.00000,        0.00214,        0.00182,
pop_avg_sum:            0.99893

vel_85_average:         2229.81837,     501.40373,      22.39205,       18.70680,
vel_85_variance:        128580.44300,   726345578.106,  26950.79921,    21591.18976,
vel_85_std_dev:         356.85240,      1374.22988,     37.07061,       30.03591,
vel_85_abs_dev:         256.39801,      1092.26906,     33.04949,       29.14321,
vel_85_acorr:           0.02792,        0.00560,        0.07481,        0.06379,
vel_85_wavg:            2223.54607,     577.55970,      24.03247,       20.56830,
vel_85_wvari:           121455.58874,   768339657.454,  27718.94041,    22094.98944,
vel_85_w_std_dev:       346.49005,      1555.81901,     39.44387,       31.73929,
vel_85_w_abs_dev:       254.91016,      1289.26253,     35.90630,       31.75890,
vel_85_sw_avg:          2373.03257,     775.72029,      27.85176,       22.68316,
vel_85_sw_var:          62025.71806,    203391709.645,  14261.54654,    12906.79441,
vel_85_sw_std_dev:      247.56878,      817.12807,      28.58545,       25.99473,
vel_85_sw_abs_dev:      214.52907,      637.93515,      25.25738,       21.24458,
vel_85_upperq:          2427.70000,     8.23333,        2.86938,        2.10000,
vel_85_lowerq:          2027.22000,     24934.21733,    157.90572,      131.25200,
vel_85_rmax:            2027.22000,     24934.21733,    157.90572,      131.25200,
vel_85_rmin:            2027.22000,     24934.21733,    157.90572,      131.25200,
vel_85_0%-10%:          0.00000,        0.00000,        0.00000,        0.00000,
vel_85_10%-20%:         0.00234,        0.00000,        0.00197,        0.00150,
```

4.1. Motor testing

vel_85_20%-30%:	0.06746,	0.00077,	0.02775,	0.02053,
vel_85_30%-40%:	0.12620,	0.00116,	0.03410,	0.02776,
vel_85_40%-50%:	0.70744,	0.00373,	0.06106,	0.04970,
vel_85_50%-60%:	0.09011,	0.00570,	0.07552,	0.06168,
vel_85_60%-70%:	0.00265,	0.00000,	0.00080,	0.00061,
vel_85_70%-80%:	0.00110,	0.00000,	0.00071,	0.00056,
vel_85_80%-90%:	0.00048,	0.00000,	0.00049,	0.00033,
vel_85_90%-100%:	0.00110,	0.00000,	0.00139,	0.00101,
pop_avg_sum:	0.99890			
vel_95_average:	2032.34870,	60.27229,	7.76352,	5.99025,
vel_95_variance:	158770.65731,	50295452.77812,	7091.92871,	5120.11973,
vel_95_std_dev:	398.37308,	77.27771,	8.79077,	6.37495,
vel_95_abs_dev:	272.08655,	47.14485,	6.86621,	5.28341,
vel_95_acorr:	0.04334,	0.00047,	0.02158,	0.01629,
vel_95_wavg:	2026.63434,	61.59125,	7.84801,	6.07712,
vel_95_wvari:	155364.94490,	49537947.81287,	7038.31996,	4758.01684,
vel_95_w_std_dev:	394.07323,	79.14707,	8.89646,	6.02328,
vel_95_w_abs_dev:	271.11948,	50.81070,	7.12816,	5.19714,
vel_95_sw_avg:	2184.60026,	883.19700,	29.71863,	24.87889,
vel_95_sw_var:	69751.59306,	46019570.15986,	6783.77256,	5446.89789,
vel_95_sw_std_dev:	263.81291,	171.49247,	13.09551,	10.47304,
vel_95_sw_abs_dev:	226.54662,	476.01830,	21.81784,	18.26772,
vel_95_upperq:	2255.90000,	23.21111,	4.81779,	3.90000,
vel_95_lowerq:	1903.72000,	250.96178,	15.84177,	11.73600,
vel_95_rmax:	1903.72000,	250.96178,	15.84177,	11.73600,
vel_95_rmin:	1903.72000,	250.96178,	15.84177,	11.73600,
vel_95_0%-10%:	0.00000,	0.00000,	0.00000,	0.00000,
vel_95_10%-20%:	0.02706,	0.00092,	0.03041,	0.02635,
vel_95_20%-30%:	0.11647,	0.00397,	0.06300,	0.05434,
vel_95_30%-40%:	0.28023,	0.08765,	0.29606,	0.25734,
vel_95_40%-50%:	0.46821,	0.10505,	0.32411,	0.27765,
vel_95_50%-60%:	0.09749,	0.00790,	0.08888,	0.07505,
vel_95_60%-70%:	0.00345,	0.00000,	0.00092,	0.00079,
vel_95_70%-80%:	0.00191,	0.00000,	0.00099,	0.00081,
vel_95_80%-90%:	0.00058,	0.00000,	0.00051,	0.00044,
vel_95_90%-100%:	0.00324,	0.00001,	0.00233,	0.00191,
pop_avg_sum:	0.99864			
vel_105_average:	1890.40707,	242.34239,	15.56735,	12.35132,
vel_105_variance:	148513.17360,	39447551.07415,	6280.72855,	4286.39472,
vel_105_std_dev:	385.29687,	66.10896,	8.13074,	5.54203,
vel_105_abs_dev:	249.08379,	41.44008,	6.43740,	4.60441,
vel_105_acorr:	0.03736,	0.00078,	0.02791,	0.01883,
vel_105_wavg:	1883.18636,	256.40784,	16.01274,	12.60924,
vel_105_wvari:	144345.10332,	37573769.19265,	6129.74463,	4830.33345,
vel_105_w_std_dev:	379.84944,	66.11776,	8.13128,	6.39506,
vel_105_w_abs_dev:	249.10368,	52.97987,	7.27873,	5.96999,
vel_105_sw_avg:	2019.60838,	770.58208,	27.75936,	24.67870,
vel_105_sw_var:	64834.22902,	34632661.97561,	5884.95216,	3707.98107,
vel_105_sw_std_dev:	254.37285,	142.98239,	11.95752,	7.44559,
vel_105_sw_abs_dev:	208.15438,	465.23923,	21.56940,	19.46334,
vel_105_upperq:	2107.90000,	79.87778,	8.93744,	7.28000,
vel_105_lowerq:	1796.10000,	1530.02889,	39.11558,	34.82000,
vel_105_rmax:	1796.10000,	1530.02889,	39.11558,	34.82000,
vel_105_rmin:	1796.10000,	1530.02889,	39.11558,	34.82000,
vel_105_0%-10%:	0.00000,	0.00000,	0.00000,	0.00000,
vel_105_10%-20%:	0.02019,	0.00074,	0.02713,	0.02042,
vel_105_20%-30%:	0.09523,	0.00041,	0.02022,	0.01737,
vel_105_30%-40%:	0.23707,	0.08453,	0.29073,	0.22003,
vel_105_40%-50%:	0.54824,	0.08815,	0.29690,	0.22308,
vel_105_50%-60%:	0.08556,	0.00993,	0.09964,	0.08241,
vel_105_60%-70%:	0.00349,	0.00000,	0.00204,	0.00135,
vel_105_70%-80%:	0.00239,	0.00000,	0.00158,	0.00114,
vel_105_80%-90%:	0.00100,	0.00000,	0.00093,	0.00080,
vel_105_90%-100%:	0.00490,	0.00001,	0.00270,	0.00196,
pop_avg_sum:	0.99807			

```
vel_115_average:        1747.01638,     92.52857,       9.61918,        7.76787,
vel_115_variance:       124892.66739,   26687719.00472, 5166.01578,     3818.95046,
vel_115_std_dev:        353.33146,      55.04905,       7.41950,        5.44896,
vel_115_abs_dev:        227.87368,      24.72489,       4.97241,        4.49168,
vel_115_acorr:          -0.00435,       0.00031,        0.01762,        0.01323,
vel_115_wavg:           1740.39663,     91.58598,       9.57006,        7.77169,
vel_115_wvari:          122010.09299,   20574705.09596, 4535.93486,     3211.66235,
vel_115_w_std_dev:      349.24356,      43.36589,       6.58528,        4.66180,
vel_115_w_abs_dev:      227.09182,      24.37072,       4.93667,        4.29858,
vel_115_sw_avg:         1909.50897,     599.38406,      24.48232,       17.65672,
vel_115_sw_var:         59968.53231,    34290597.42048, 5855.81740,     3649.33191,
vel_115_sw_std_dev:     244.59289,      158.72276,      12.59852,       7.81287,
vel_115_sw_abs_dev:     212.02931,      393.21044,      19.82953,       11.86001,
vel_115_upperq:         1973.20000,     25.28889,       5.02881,        4.08000,
vel_115_lowerq:         1670.94000,     456.37378,      21.36291,       15.99200,
vel_115_rmax:           1670.94000,     456.37378,      21.36291,       15.99200,
vel_115_rmin:           1670.94000,     456.37378,      21.36291,       15.99200,
vel_115_0%-10%:         0.00000,        0.00000,        0.00000,        0.00000,
vel_115_10%-20%:        0.01067,        0.00023,        0.01501,        0.00848,
vel_115_20%-30%:        0.08898,        0.00028,        0.01665,        0.01212,
vel_115_30%-40%:        0.15154,        0.02741,        0.16556,        0.09981,
vel_115_40%-50%:        0.53154,        0.02035,        0.14265,        0.08999,
vel_115_50%-60%:        0.20412,        0.01737,        0.13178,        0.11405,
vel_115_60%-70%:        0.00378,        0.00000,        0.00152,        0.00128,
vel_115_70%-80%:        0.00284,        0.00000,        0.00119,        0.00079,
vel_115_80%-90%:        0.00139,        0.00000,        0.00147,        0.00090,
vel_115_90%-100%:       0.00451,        0.00001,        0.00270,        0.00201,
pop_avg_sum:            0.99938

vel_125_average:        1626.91482,     94.81666,       9.73738,        7.84427,
vel_125_variance:       112980.26895,   29043619.30769, 5389.21324,     4370.06526,
vel_125_std_dev:        336.03824,      65.08160,       8.06732,        6.52030,
vel_125_abs_dev:        218.71768,      23.05370,       4.80143,        3.98839,
vel_125_acorr:          -0.02062,       0.00040,        0.01998,        0.01437,
vel_125_wavg:           1620.55089,     102.22714,      10.11074,       8.26544,
vel_125_wvari:          109625.18127,   36043232.70728, 6003.60164,     4988.07596,
vel_125_w_std_dev:      330.98316,      83.69774,       9.14865,        7.55979,
vel_125_w_abs_dev:      216.11152,      28.57612,       5.34566,        4.61000,
vel_125_sw_avg:         1788.33123,     1121.67664,     33.49144,       24.27920,
vel_125_sw_var:         60307.39811,    8210043.43889,  2865.31734,     1942.76875,
vel_125_sw_std_dev:     245.51326,      34.04284,       5.83462,        3.94271,
vel_125_sw_abs_dev:     211.37806,      300.87070,      17.34562,       11.54077,
vel_125_upperq:         1862.18000,     42.10178,       6.48859,        5.38000,
vel_125_lowerq:         1537.52000,     353.56622,      18.80336,       13.88000,
vel_125_rmax:           1537.52000,     353.56622,      18.80336,       13.88000,
vel_125_rmin:           1537.52000,     353.56622,      18.80336,       13.88000,
vel_125_0%-10%:         0.00000,        0.00000,        0.00000,        0.00000,
vel_125_10%-20%:        0.02162,        0.00092,        0.03038,        0.02597,
vel_125_20%-30%:        0.09190,        0.00049,        0.02211,        0.01657,
vel_125_30%-40%:        0.27372,        0.09006,        0.30009,        0.25589,
vel_125_40%-50%:        0.38649,        0.04255,        0.20627,        0.16883,
vel_125_50%-60%:        0.21366,        0.02102,        0.14500,        0.12598,
vel_125_60%-70%:        0.00393,        0.00000,        0.00114,        0.00087,
vel_125_70%-80%:        0.00309,        0.00000,        0.00154,        0.00114,
vel_125_80%-90%:        0.00076,        0.00000,        0.00065,        0.00056,
vel_125_90%-100%:       0.00424,        0.00001,        0.00311,        0.00256,
pop_avg_sum:            0.99941

vel_135_average:        1519.47910,     142.74073,      11.94742,       8.57553,
vel_135_variance:       107731.42106,   49562802.44589, 7040.08540,     6088.38797,
vel_135_std_dev:        328.06598,      115.70492,      10.75662,       9.28734,
vel_135_abs_dev:        216.50629,      26.85767,       5.18244,        4.13854,
vel_135_acorr:          -0.03707,       0.00071,        0.02663,        0.01959,
vel_135_wavg:           1513.27607,     137.65439,      11.73262,       8.56247,
vel_135_wvari:          104116.37094,   46493169.29111, 6818.58998,     5445.71208,
vel_135_w_std_dev:      322.51387,      112.41541,      10.60261,       8.45208,
vel_135_w_abs_dev:      212.65029,      25.78370,       5.07777,        4.01406,
vel_135_sw_avg:         1676.40030,     862.54977,      29.36920,       22.51881,
```

4.1. Motor testing

```
vel_135_sw_var:         56682.85379,    39494085.63267, 6284.43201,     4484.21841,
vel_135_sw_std_dev:     237.73346,      184.06418,      13.56703,       9.70149,
vel_135_sw_abs_dev:     205.78525,      502.24098,      22.41073,       17.33581,
vel_135_upperq:         1759.38000,     62.28844,       7.89230,        5.09600,
vel_135_lowerq:         1420.88000,     243.61956,      15.60832,       10.32800,
vel_135_rmax:           1420.88000,     243.61956,      15.60832,       10.32800,
vel_135_rmin:           1420.88000,     243.61956,      15.60832,       10.32800,
vel_135_0%-10%:         0.00000,        0.00000,        0.00000,        0.00000,
vel_135_10%-20%:        0.02900,        0.00159,        0.03986,        0.03349,
vel_135_20%-30%:        0.17983,        0.03467,        0.18621,        0.14870,
vel_135_30%-40%:        0.22011,        0.03773,        0.19425,        0.15007,
vel_135_40%-50%:        0.38376,        0.07000,        0.26457,        0.22617,
vel_135_50%-60%:        0.17226,        0.02027,        0.14236,        0.12783,
vel_135_60%-70%:        0.00463,        0.00000,        0.00221,        0.00157,
vel_135_70%-80%:        0.00239,        0.00000,        0.00176,        0.00150,
vel_135_80%-90%:        0.00149,        0.00000,        0.00114,        0.00093,
vel_135_90%-100%:       0.00491,        0.00002,        0.00399,        0.00339,
pop_avg_sum:            0.99838

vel_145_average:        1423.17261,     61.08698,       7.81582,        5.95266,
vel_145_variance:       90107.02658,    6345655.03665,  2519.05836,     2213.77442,
vel_145_std_dev:        300.15201,      17.55113,       4.18941,        3.67937,
vel_145_abs_dev:        207.97709,      5.59408,        2.36518,        1.97992,
vel_145_acorr:          -0.07614,       0.00057,        0.02394,        0.01685,
vel_145_wavg:           1416.80416,     45.72279,       6.76186,        4.99064,
vel_145_wvari:          85475.39297,    8311512.04543,  2882.96931,     2380.81234,
vel_145_w_std_dev:      292.32432,      24.31868,       4.93140,        4.07223,
vel_145_w_abs_dev:      202.70012,      7.17083,        2.67784,        2.04521,
vel_145_sw_avg:         1581.93306,     802.39957,      28.32666,       24.29600,
vel_145_sw_var:         56977.62131,    44145756.29684, 6644.22729,     5548.84482,
vel_145_sw_std_dev:     238.31778,      202.50816,      14.23054,       11.91916,
vel_145_sw_abs_dev:     211.53201,      426.33967,      20.64799,       17.63684,
vel_145_upperq:         1670.02000,     31.43511,       5.60670,        4.50400,
vel_145_lowerq:         1292.60000,     288.48889,      16.98496,       12.40000,
vel_145_rmax:           1407.26000,     32421.03156,    180.05841,      155.96400,
vel_145_rmin:           1292.60000,     288.48889,      16.98496,       12.40000,
vel_145_0%-10%:         0.00000,        0.00000,        0.00000,        0.00000,
vel_145_10%-20%:        0.00829,        0.00029,        0.01698,        0.01278,
vel_145_20%-30%:        0.07079,        0.00014,        0.01196,        0.00833,
vel_145_30%-40%:        0.22570,        0.03375,        0.18372,        0.13132,
vel_145_40%-50%:        0.45705,        0.01015,        0.10076,        0.08193,
vel_145_50%-60%:        0.22278,        0.01344,        0.11595,        0.08700,
vel_145_60%-70%:        0.00423,        0.00000,        0.00120,        0.00086,
vel_145_70%-80%:        0.00327,        0.00000,        0.00165,        0.00096,
vel_145_80%-90%:        0.00153,        0.00000,        0.00128,        0.00082,
vel_145_90%-100%:       0.00583,        0.00001,        0.00330,        0.00234,
pop_avg_sum:            0.99945

vel_155_average:        1346.96383,     109.41524,      10.46017,       6.24068,
vel_155_variance:       72501.05435,    15277774.01694, 3908.67932,     3218.01879,
vel_155_std_dev:        269.17154,      53.04113,       7.28293,        6.00310,
vel_155_abs_dev:        197.59593,      8.03137,        2.83397,        2.24662,
vel_155_acorr:          -0.05628,       0.00125,        0.03529,        0.02484,
vel_155_wavg:           1341.27446,     125.56764,      11.20570,       6.38986,
vel_155_wvari:          67495.89749,    10479681.59274, 3237.23363,     2767.16371,
vel_155_w_std_dev:      259.73246,      38.83070,       6.23143,        5.30981,
vel_155_w_abs_dev:      191.67194,      7.35632,        2.71225,        2.21176,
vel_155_sw_avg:         1476.78697,     1448.40761,     38.05795,       32.64075,
vel_155_sw_var:         49552.93193,    81412653.38016, 9022.89606,     7800.04425,
vel_155_sw_std_dev:     221.80889,      393.05448,      19.82560,       17.21301,
vel_155_sw_abs_dev:     189.15142,      737.79835,      27.16244,       23.58685,
vel_155_upperq:         1594.40000,     24.93333,       4.99333,        3.60000,
vel_155_lowerq:         1179.20000,     208.40000,      14.43607,       8.76000,
vel_155_rmax:           1429.20000,     44922.40000,    211.94905,      196.96000,
vel_155_rmin:           1179.20000,     208.40000,      14.43607,       8.76000,
vel_155_0%-10%:         0.00000,        0.00000,        0.00000,        0.00000,
vel_155_10%-20%:        0.04612,        0.01603,        0.12662,        0.07190,
vel_155_20%-30%:        0.10997,        0.02778,        0.16667,        0.09383,
```

4.1. Motor testing

```
vel_155_30%-40%:        0.33922,        0.03681,        0.19186,        0.15104,
vel_155_40%-50%:        0.31894,        0.01464,        0.12098,        0.08402,
vel_155_50%-60%:        0.17196,        0.02105,        0.14508,        0.13453,
vel_155_60%-70%:        0.00435,        0.00001,        0.00231,        0.00193,
vel_155_70%-80%:        0.00455,        0.00001,        0.00237,        0.00158,
vel_155_80%-90%:        0.00105,        0.00000,        0.00082,        0.00070,
vel_155_90%-100%:       0.00337,        0.00001,        0.00298,        0.00263,
pop_avg_sum:            0.99953

vel_165_average:        1276.92694,     70.64530,       8.40508,        7.05959,
vel_165_variance:       67026.19544,    18646238.85381, 4318.12909,     2915.70914,
vel_165_std_dev:        258.77172,      70.43421,       8.39251,        5.67266,
vel_165_abs_dev:        196.83124,      15.34378,       3.91711,        2.52646,
vel_165_acorr:          -0.06331,       0.00053,        0.02298,        0.01800,
vel_165_wavg:           1270.76998,     72.42873,       8.51051,        7.36495,
vel_165_wvari:          62241.39678,    17372049.62058, 4167.97908,     2675.17386,
vel_165_w_std_dev:      249.35554,      70.23428,       8.38059,        5.40028,
vel_165_w_abs_dev:      190.77995,      15.19556,       3.89815,        2.55852,
vel_165_sw_avg:         1395.67846,     1445.26327,     38.01662,       28.61727,
vel_165_sw_var:         46986.17823,    68195818.70084, 8258.07597,     6335.48415,
vel_165_sw_std_dev:     216.06661,      334.88591,      18.29989,       14.23277,
vel_165_sw_abs_dev:     182.07326,      679.35422,      26.06442,       19.55258,
vel_165_upperq:         1532.34000,     75.64489,       8.69741,        7.39200,
vel_165_lowerq:         1094.90000,     78.76667,       8.87506,        6.70000,
vel_165_rmax:           1312.30000,     53303.56667,    230.87565,      218.90000,
vel_165_rmin:           1094.90000,     78.76667,       8.87506,        6.70000,
vel_165_0%-10%:         0.00000,        0.00000,        0.00000,        0.00000,
vel_165_10%-20%:        0.00459,        0.00006,        0.00784,        0.00581,
vel_165_20%-30%:        0.10074,        0.01774,        0.13319,        0.10002,
vel_165_30%-40%:        0.40057,        0.01440,        0.12002,        0.10483,
vel_165_40%-50%:        0.31329,        0.00857,        0.09256,        0.06761,
vel_165_50%-60%:        0.16501,        0.01902,        0.13793,        0.12781,
vel_165_60%-70%:        0.00536,        0.00000,        0.00161,        0.00124,
vel_165_70%-80%:        0.00424,        0.00000,        0.00116,        0.00084,
vel_165_80%-90%:        0.00177,        0.00000,        0.00137,        0.00092,
vel_165_90%-100%:       0.00380,        0.00001,        0.00358,        0.00314,
pop_avg_sum:            0.99937

vel_175_average:        1211.24695,     253.34701,      15.91688,       12.52629,
vel_175_variance:       62341.61479,    3973656.07574,  1993.40314,     1451.07694,
vel_175_std_dev:        249.65378,      16.22964,       4.02860,        2.92395,
vel_175_abs_dev:        195.47277,      6.20380,        2.49074,        1.73768,
vel_175_acorr:          -0.10757,       0.00051,        0.02265,        0.01563,
vel_175_wavg:           1205.38885,     272.92774,      16.52052,       13.10912,
vel_175_wvari:          57705.53798,    2992924.81445,  1730.00717,     1310.04902,
vel_175_w_std_dev:      240.19536,      13.02810,       3.60945,        2.74081,
vel_175_w_abs_dev:      189.19322,      5.08833,        2.25573,        1.71325,
vel_175_sw_avg:         1311.75987,     613.36591,      24.76623,       15.69370,
vel_175_sw_var:         40264.77814,    10404782.41204, 3225.64450,     2344.73955,
vel_175_sw_std_dev:     200.51978,      62.88418,       7.92995,        5.80889,
vel_175_sw_abs_dev:     162.48767,      123.87271,      11.12981,       8.18032,
vel_175_upperq:         1469.26000,     245.38711,      15.66484,       12.39200,
vel_175_lowerq:         1029.70000,     623.12222,      24.96242,       21.10000,
vel_175_rmax:           1119.30000,     31375.12222,    177.13024,      133.48000,
vel_175_rmin:           1029.70000,     623.12222,      24.96242,       21.10000,
vel_175_0%-10%:         0.00000,        0.00000,        0.00000,        0.00000,
vel_175_10%-20%:        0.00087,        0.00001,        0.00275,        0.00156,
vel_175_20%-30%:        0.05566,        0.00822,        0.09067,        0.05153,
vel_175_30%-40%:        0.45119,        0.00497,        0.07053,        0.06199,
vel_175_40%-50%:        0.26229,        0.00253,        0.05028,        0.04388,
vel_175_50%-60%:        0.21309,        0.00604,        0.07769,        0.04919,
vel_175_60%-70%:        0.00549,        0.00000,        0.00167,        0.00135,
vel_175_70%-80%:        0.00356,        0.00000,        0.00117,        0.00081,
vel_175_80%-90%:        0.00263,        0.00000,        0.00132,        0.00085,
vel_175_90%-100%:       0.00446,        0.00000,        0.00206,        0.00164,
pop_avg_sum:            0.99924

vel_185_average:        1170.13581,     259.15133,      16.09818,       9.86336,
```

```
vel_185_variance:       59104.46068,    6234918.39977,  2496.98186,     1893.71910,
vel_185_std_dev:        243.06510,      26.46271,       5.14419,        3.88688,
vel_185_abs_dev:        193.82743,      4.04232,        2.01055,        1.39254,
vel_185_acorr:          -0.11453,       0.00185,        0.04305,        0.03368,
vel_185_wavg:           1163.90719,     281.30419,      16.77213,       10.50859,
vel_185_wvari:          54316.74857,    4968333.44853,  2228.97587,     1695.17051,
vel_185_w_std_dev:      233.01599,      22.54965,       4.74865,        3.62776,
vel_185_w_abs_dev:      187.66160,      3.89498,        1.97357,        1.45341,
vel_185_sw_avg:         1291.89553,     3362.64899,     57.98835,       50.31468,
vel_185_sw_var:         47587.82799,    242806260.789,  15582.24184,    13481.38644,
vel_185_sw_std_dev:     215.76308,      1149.02333,     33.89725,       29.33263,
vel_185_sw_abs_dev:     184.33094,      1920.12602,     43.81924,       37.90764,
vel_185_upperq:         1432.68000,     155.06844,      12.45265,       8.85600,
vel_185_lowerq:         994.30000,      244.67778,      15.64218,       11.42000,
vel_185_rmax:           1080.10000,     33714.98889,    183.61642,      138.96000,
vel_185_rmin:           994.30000,      244.67778,      15.64218,       11.42000,
vel_185_0%-10%:         0.00000,        0.00000,        0.00000,        0.00000,
vel_185_10%-20%:        0.00000,        0.00000,        0.00000,        0.00000,
vel_185_20%-30%:        0.03596,        0.00034,        0.01842,        0.01542,
vel_185_30%-40%:        0.50646,        0.00864,        0.09297,        0.08122,
vel_185_40%-50%:        0.26036,        0.00138,        0.03710,        0.02509,
vel_185_50%-60%:        0.18073,        0.01442,        0.12010,        0.10295,
vel_185_60%-70%:        0.00566,        0.00000,        0.00209,        0.00165,
vel_185_70%-80%:        0.00362,        0.00000,        0.00109,        0.00075,
vel_185_80%-90%:        0.00251,        0.00000,        0.00065,        0.00050,
vel_185_90%-100%:       0.00355,        0.00001,        0.00255,        0.00213,
pop_avg_sum:            0.99886

vel_195_average:        1151.05788,     220.00594,      14.83260,       10.58960,
vel_195_variance:       58996.12592,    13021278.54128, 3608.50087,     3140.17081,
vel_195_std_dev:        242.79031,      54.43394,       7.37794,        6.42511,
vel_195_abs_dev:        195.88139,      8.37745,        2.89438,        2.32072,
vel_195_acorr:          -0.17131,       0.00141,        0.03762,        0.03062,
vel_195_wavg:           1145.79458,     208.02633,      14.42312,       10.36105,
vel_195_wvari:          55572.49724,    15698805.74526, 3962.17185,     3311.16067,
vel_195_w_std_dev:      235.60446,      70.03811,       8.36888,        6.98459,
vel_195_w_abs_dev:      191.42429,      13.81120,       3.71634,        2.85430,
vel_195_sw_avg:         1251.69533,     1823.78774,     42.70583,       32.57186,
vel_195_sw_var:         39957.12989,    57210129.70833, 7563.73781,     5594.22258,
vel_195_sw_std_dev:     199.16913,      320.87345,      17.91294,       13.57826,
vel_195_sw_abs_dev:     167.78535,      508.51495,      22.55028,       16.40403,
vel_195_upperq:         1409.48000,     168.09956,      12.96532,       9.74400,
vel_195_lowerq:         988.90000,      350.32222,      18.71690,       14.50000,
vel_195_rmax:           1030.60000,     18119.15556,    134.60741,      75.88000,
vel_195_rmin:           988.90000,      350.32222,      18.71690,       14.50000,
vel_195_0%-10%:         0.00000,        0.00000,        0.00000,        0.00000,
vel_195_10%-20%:        0.00098,        0.00001,        0.00311,        0.00177,
vel_195_20%-30%:        0.07505,        0.02808,        0.16756,        0.09530,
vel_195_30%-40%:        0.46670,        0.01232,        0.11099,        0.08512,
vel_195_40%-50%:        0.21499,        0.00357,        0.05976,        0.04756,
vel_195_50%-60%:        0.22256,        0.01039,        0.10191,        0.07449,
vel_195_60%-70%:        0.00674,        0.00001,        0.00271,        0.00211,
vel_195_70%-80%:        0.00337,        0.00000,        0.00125,        0.00106,
vel_195_80%-90%:        0.00320,        0.00000,        0.00156,        0.00103,
vel_195_90%-100%:       0.00505,        0.00001,        0.00330,        0.00241,
pop_avg_sum:            0.99864
```

Massive output, ain't it?. As you can see the test has generated a full set of statistics for all the velocities ranging from velocity 15 to velocity 195 for both encoder lines of the two motors. I decided to copy the full output here because it is a pretty good survey of how our Raspberry Pi will behave with two motors plugged. As mentioned above the test took 10 samples for each velocity, so the fields in the rows must be interpreted like the fields of the latter text explained, test number 3, the columns as labeled, stands for the average, variance, standard deviation and absolute deviation of the ten samples taken for each velocity so there is a lot of useless fields, however it are computed to keep the code simple, with this values we can say that we have a

pretty reliable values of time between ticks (average row, average column) and physical error (abs_dev row, average column) for each velocity, this values will be computed by the function `mt_calibrate` on demand. You can see the average of the population distribution relative to the maximum values taken, so, actually the data set is not really stable hence the closed loop control wont be very reliable neither. As explained above too, our pulses are produced at a very low frequency and the allocation time of the structures needed to manage it make the closed loop control a little more than an illusion. However it is implemented and it will be introduced later on.

I'll jump now to the test number 7, it will calibrate one, or both motors and once it is done it will interpolate the data set computed by the calibration to get a time between ticks and physical error assumption, we gonna compare this values with the ones obtained in the latter section to see if our the calibrations are effective and to get a minimum set of velocities we need configure for our interpolations to be reliable. The test number 7 will calibrate the motor/s for the number of velocities we request trough the `--samples / -S` option, later on it will interpolate the velocities from `MIN_VEL + (turns)`, to `MAX_VEL` (200) in steps of the size requested trough the `--step / -s` parameter, if the turns parameter is not set the initial gap added to `MIN_VEL` will be 0. Lets then make a fist execution with the more precise set of velocities we can get, it is, with 20 velocities from `MIN_VEL` to `MAX_VEL`, we will request an step of 7 to interpolate the values in the middle of the values calculated for the `mt_calibrate` function, that, since we are setting the samples option to 20, will match the values of the last test done, later on we will set less samples and try to interpolate the values get for the test number 4. For this test the turns parameter `--turns / -r` is used in a little bit tricky way, to avoid interpolate for `MIN_VEL`, which is always computed by the `mt_calibrate` function, the turns parameter will be added to `MIN_VEL` as an initial gap for the set of interpolations we will do, after this we will continue interpolating values with the requested step till the last velocity smaller than `MAX_VEL`, then:

```
$> sudo ./mt_test -t7 -S20 -s7 -p2 -r3
```

will generate an output like:

```
Calibrating MOTORS

MOTOR 0: params for  18 vel >> tbticks: 7326, err: 1235
MOTOR 1: params for  18 vel >> tbticks: 6488, err: 1321
MOTOR 0: params for  25 vel >> tbticks: 5013, err: 911
MOTOR 1: params for  25 vel >> tbticks: 4832, err: 975
MOTOR 0: params for  32 vel >> tbticks: 4316, err: 769
MOTOR 1: params for  32 vel >> tbticks: 4171, err: 857
MOTOR 0: params for  39 vel >> tbticks: 3862, err: 706
MOTOR 1: params for  39 vel >> tbticks: 3734, err: 755
MOTOR 0: params for  46 vel >> tbticks: 3486, err: 656
MOTOR 1: params for  46 vel >> tbticks: 3377, err: 672
MOTOR 0: params for  53 vel >> tbticks: 3169, err: 606
MOTOR 1: params for  53 vel >> tbticks: 3073, err: 655
MOTOR 0: params for  60 vel >> tbticks: 2875, err: 554
MOTOR 1: params for  60 vel >> tbticks: 2824, err: 595
MOTOR 0: params for  67 vel >> tbticks: 2648, err: 504
MOTOR 1: params for  67 vel >> tbticks: 2612, err: 540
MOTOR 0: params for  74 vel >> tbticks: 2476, err: 472
MOTOR 1: params for  74 vel >> tbticks: 2424, err: 513
MOTOR 0: params for  81 vel >> tbticks: 2282, err: 444
MOTOR 1: params for  81 vel >> tbticks: 2254, err: 496
MOTOR 0: params for  88 vel >> tbticks: 2125, err: 422
MOTOR 1: params for  88 vel >> tbticks: 2104, err: 478
MOTOR 0: params for  95 vel >> tbticks: 2017, err: 407
MOTOR 1: params for  95 vel >> tbticks: 1978, err: 458
MOTOR 0: params for 102 vel >> tbticks: 1910, err: 369
```

```
MOTOR 1: params for 102 vel >> tbticks: 1877, err: 408
MOTOR 0: params for 109 vel >> tbticks: 1800, err: 353
MOTOR 1: params for 109 vel >> tbticks: 1793, err: 369
MOTOR 0: params for 116 vel >> tbticks: 1692, err: 348
MOTOR 1: params for 116 vel >> tbticks: 1714, err: 341
MOTOR 0: params for 123 vel >> tbticks: 1613, err: 331
MOTOR 1: params for 123 vel >> tbticks: 1618, err: 316
MOTOR 0: params for 130 vel >> tbticks: 1546, err: 322
MOTOR 1: params for 130 vel >> tbticks: 1548, err: 310
MOTOR 0: params for 137 vel >> tbticks: 1479, err: 319
MOTOR 1: params for 137 vel >> tbticks: 1482, err: 306
MOTOR 0: params for 144 vel >> tbticks: 1411, err: 321
MOTOR 1: params for 144 vel >> tbticks: 1404, err: 299
MOTOR 0: params for 151 vel >> tbticks: 1351, err: 325
MOTOR 1: params for 151 vel >> tbticks: 1333, err: 297
MOTOR 0: params for 158 vel >> tbticks: 1301, err: 324
MOTOR 1: params for 158 vel >> tbticks: 1281, err: 294
MOTOR 0: params for 165 vel >> tbticks: 1261, err: 317
MOTOR 1: params for 165 vel >> tbticks: 1241, err: 289
MOTOR 0: params for 172 vel >> tbticks: 1240, err: 312
MOTOR 1: params for 172 vel >> tbticks: 1191, err: 283
MOTOR 0: params for 179 vel >> tbticks: 1210, err: 307
MOTOR 1: params for 179 vel >> tbticks: 1151, err: 282
MOTOR 0: params for 186 vel >> tbticks: 1178, err: 294
MOTOR 1: params for 186 vel >> tbticks: 1122, err: 284
MOTOR 0: params for 193 vel >> tbticks: 1181, err: 261
MOTOR 1: params for 193 vel >> tbticks: 1098, err: 273
MOTOR 0: params for 199 vel >> tbticks: 1190, err: 236
MOTOR 1: params for 199 vel >> tbticks: 1089, err: 270
```

If you take a look at the huge output of the test number 4 you will find that for the first motor, the motor plugged to port 0, the values of time between ticks for 15 velocity are 5384 for the first encoder line and 5051 for the second encoder line, the `mt_calibrate` will compute the time between ticks value with the average (once again ...) of both lines, so $^{(5384+5051)}/_{2} = 5217$, the interpolation gives for this motor a value of 5013µs, it is, we have 200µs of difference, which is pretty fair, specially when the physical error (the absolute deviation of the samples) is about 900µs. Same thing happens for the motor two, if we take a look at the big output of test number 4 it says that the time between ticks for 25 velocity is 4976 for the encoder line 1 and 4999 for the encoder line 2, so $^{(4976+4999)}/_{2} = 4987$ and the interpolation give us a time of 4832µs, pretty fair again. Lets take a look at a higher velocity, 165 is the one that exactly matches the values computed in test 4. Test 4 says that for a 165 velocity the value of time between ticks is for the motor port 0 $^{(1318+1309)}/_{2} = 1313$ our interpolation gives a value of 1261, which is not bad to me, now for the motor port 1 test 4 says $^{(1320+1276)}/_{2} = 1298$ and the interpolation gives a value of 1241µs, so everything is working pretty well with the hardware we have. The rest of the values interpolated aren't computed on the test 4, however you can check the closest velocity you can find in test four for each of the values returned in the last test and you will realize that they are pretty fair with reality too, or, at least with our reality. Nevertheless you can catch me out here, since the `mt_calibrate` function will compute our parameters (time between ticks and physical error) for all velocities from `MIN_VEL` (15) to `MAX_VEL` (200), with a step of `MAX_VEL/samples`, then, with a few calculations you will realize that the velocities 15, 25 and 165 have been computed for the `mt_calibrate` function, hence the interpolation will return the values computed for the function, no interpolation will be performed, it is, the `mt_calibrate` function is working fine, however we didn't interpolate any value that was computed for the test number 4 here. Lets then make another execution of the test 7, we will set the samples to 10 this time, to see if our interpolations are fair with less points of reference, and the velocities for what the `mt_calibrate` function will compute the parameters will be {(`MIN_VEL`)15, 35, 55, 75, 95, 115, 135, 155, 175 and 200 (`MAX_VEL`)}, then we will set the step option to 10 and we won't set the turns

option this time, then the velocities we will interpolate for will be {(MIN_VEL)15, 25, 35, 45, 55, 65, 75, 85, 95, 105, 115, 125, 135, 145, 155, 165, 175 and 195}. As you can see the velocities colored in blue won't be computed by mt_calibrate, and will be in the middle of two computed velocities, then we can figure out if our interpolations are fair with reality or not, and with the half available amount of velocities computed. As usual we will run the test with both motors acting at the same time:

```
$> sudo ./mt_test -t7 -S10 -s10 -p2
```

has produced the following output:

```
Calibrating MOTORS

MOTOR 0: params for  15 vel >> tbticks: 6720, err: 1187
MOTOR 1: params for  15 vel >> tbticks: 6738, err: 1236
MOTOR 0: params for  25 vel >> tbticks: 5695, err: 971
MOTOR 1: params for  25 vel >> tbticks: 5491, err: 1037
MOTOR 0: params for  35 vel >> tbticks: 4351, err: 741
MOTOR 1: params for  35 vel >> tbticks: 3977, err: 788
MOTOR 0: params for  45 vel >> tbticks: 3644, err: 651
MOTOR 1: params for  45 vel >> tbticks: 3406, err: 687
MOTOR 0: params for  55 vel >> tbticks: 3130, err: 578
MOTOR 1: params for  55 vel >> tbticks: 2980, err: 613
MOTOR 0: params for  65 vel >> tbticks: 2798, err: 515
MOTOR 1: params for  65 vel >> tbticks: 2657, err: 557
MOTOR 0: params for  75 vel >> tbticks: 2520, err: 466
MOTOR 1: params for  75 vel >> tbticks: 2377, err: 513
MOTOR 0: params for  85 vel >> tbticks: 2286, err: 442
MOTOR 1: params for  85 vel >> tbticks: 2149, err: 484
MOTOR 0: params for  95 vel >> tbticks: 2071, err: 421
MOTOR 1: params for  95 vel >> tbticks: 1955, err: 455
MOTOR 0: params for 105 vel >> tbticks: 1881, err: 384
MOTOR 1: params for 105 vel >> tbticks: 1794, err: 420
MOTOR 0: params for 115 vel >> tbticks: 1721, err: 353
MOTOR 1: params for 115 vel >> tbticks: 1662, err: 387
MOTOR 0: params for 125 vel >> tbticks: 1597, err: 349
MOTOR 1: params for 125 vel >> tbticks: 1556, err: 362
MOTOR 0: params for 135 vel >> tbticks: 1494, err: 350
MOTOR 1: params for 135 vel >> tbticks: 1463, err: 343
MOTOR 0: params for 145 vel >> tbticks: 1403, err: 337
MOTOR 1: params for 145 vel >> tbticks: 1379, err: 327
MOTOR 0: params for 155 vel >> tbticks: 1323, err: 325
MOTOR 1: params for 155 vel >> tbticks: 1301, err: 315
MOTOR 0: params for 165 vel >> tbticks: 1263, err: 324
MOTOR 1: params for 165 vel >> tbticks: 1235, err: 307
MOTOR 0: params for 175 vel >> tbticks: 1212, err: 327
MOTOR 1: params for 175 vel >> tbticks: 1174, err: 300
MOTOR 0: params for 185 vel >> tbticks: 1198, err: 314
MOTOR 1: params for 185 vel >> tbticks: 1138, err: 292
MOTOR 0: params for 195 vel >> tbticks: 1195, err: 274
MOTOR 1: params for 195 vel >> tbticks: 1113, err: 284
```

Lets, for example, take a look at the velocity 45, the output table of the test number 4 give us a time between ticks value for motor 0 of $^{(3504+3523)}/_{2} = 3513$ while the interpolated value is 3644µs, for the motor 1 test 4 give us a value of $^{(3499+3519)}/_{2} = 3509$ and the interpolated value is 3046µs, pretty good. Now lets take a look at the velocity 145, for the motor 0 test number 4 let a value of $^{(1425+1433)}/_{2} = 1429$ and the interpolated value, for this calibration, is 1403µs, and for the motor port 1 test 4 let a value of $^{(1466+1423)}/_{2} = 1444$ and the interpolated value is 1379. So, to me the results are pretty fair, and we need to take into account that we can set up to 20 samples to motor calibrate, so the interpolations can go even more precise, actually will be more problematic the readings than the interpolation itself. The library makes use of the gsl (GNU scientific library) library to perform the interpolations, there is several options to

choose to interpolate a data set with this library, however, after some testing I found out that akima spline with conditions on the boundaries will be the one that will behave the best. The data must be lineal, for velocity it should be this way, seems obvious that the more velocity the smaller will be the time between ticks, the physical error seems to be pretty linear too, however it can't be ensured, specially at low velocities so I chose this kind of interpolation because of that, otherwise linear interpolation behave pretty well for velocities requiring less computation. One thing to keep in mind is that akima splines are the best, specially in the boundaries of the data set and to the outliers (what will come in handy for the possible oscillations of the physical error), which also are the critical parts of our data set however this splines requires at last 5 points to interpolate our data set, so, the minimum samples `mt_calibrate` function will accept is 5, and the maximum, fixed for the LEGO-Pi library, will be 20, however ten points use to be more than enough. To see how interpolations behave when the minimum points are used we will run the last execution of this test for 5 samples, so the velocities computed by the `mt_calibrate` function will be {(MIN_VEL) 15, 55, 95, 135, 200 (MAX_VEL)}, then we will set the step parameter to 15 so the velocities interpolated will be {(MIN_VEL) 15, 30, 45, 60, 75, 90, 120, 135, 150, 165, 180, 195}, then:

```
$> sudo ./mt_test -t7 -S5 -s15 -p2
```

will produce an output like:

```
Calibrating MOTORS

MOTOR 0: params for  15 vel >> tbticks: 8076, err: 1316
MOTOR 1: params for  15 vel >> tbticks: 6672, err: 1362
MOTOR 0: params for  30 vel >> tbticks: 6552, err: 1084
MOTOR 1: params for  30 vel >> tbticks: 5543, err: 1144
MOTOR 0: params for  45 vel >> tbticks: 4075, err: 704
MOTOR 1: params for  45 vel >> tbticks: 3720, err: 794
MOTOR 0: params for  60 vel >> tbticks: 2937, err: 536
MOTOR 1: params for  60 vel >> tbticks: 2807, err: 618
MOTOR 0: params for  75 vel >> tbticks: 2460, err: 472
MOTOR 1: params for  75 vel >> tbticks: 2361, err: 530
MOTOR 0: params for  90 vel >> tbticks: 2109, err: 426
MOTOR 1: params for  90 vel >> tbticks: 2031, err: 464
MOTOR 0: params for 105 vel >> tbticks: 1882, err: 397
MOTOR 1: params for 105 vel >> tbticks: 1820, err: 421
MOTOR 0: params for 120 vel >> tbticks: 1678, err: 370
MOTOR 1: params for 120 vel >> tbticks: 1631, err: 383
MOTOR 0: params for 135 vel >> tbticks: 1484, err: 344
MOTOR 1: params for 135 vel >> tbticks: 1453, err: 348
MOTOR 0: params for 150 vel >> tbticks: 1370, err: 324
MOTOR 1: params for 150 vel >> tbticks: 1335, err: 326
MOTOR 0: params for 165 vel >> tbticks: 1350, err: 312
MOTOR 1: params for 165 vel >> tbticks: 1286, err: 321
MOTOR 0: params for 180 vel >> tbticks: 1337, err: 300
MOTOR 1: params for 180 vel >> tbticks: 1241, err: 317
MOTOR 0: params for 195 vel >> tbticks: 1245, err: 282
MOTOR 1: params for 195 vel >> tbticks: 1135, err: 298
```

Taking a quick look at the results we can see how the interpolations are a little bit worst when the minimum samples are taken, however they are pretty reliable, to me, the conclusion here is, if you are willing to calibrate the motors, give to the function `mt_calibrate` at least 10 samples to ensure the values interpolated are fair with reality.

Now that we know that our calibration function is fair enough with reality we can try to run a motor rotation with our "homemade" P.I.D control approach. To do this we will use test number 8, this is a very simple test that will perform a rotation of one motor of the turns requested via the `-r / --turns` at a velocity requested trough the `-v/ --vel` option, if we do not request P.I.D

control, however, it won't be activated, so it will be a useless test. The gains for pid control can be set with the options `-P / --kp`, `-I / --ki` and `-D / --kd` and the bizarre `ttc` field of the `PID` structure can be set with the `-T / --ttc` option. Additionally we can set a position control, a floating point value ranging from 0 to 1, trough the option `-b / --psctrl`, if you want a more precise explanation on how position control works you can refer to the appendix B, concretely to the function `mt_move_t`. As mentioned above P.I.D control is experimental, do do not expect a lot on this one, due to the low frequency of our pulses the less we reset it the better will the motor behave, and P.I.D control resets the pulse constantly, with such a slow pulse generation it can lead to more error when resetting the pulse, entering in an infinite correct - error, generate - error behavior. We will calibrate the motor before effectively run the test to have a proper data set of time between ticks and physical error value for one motor, this can be done with the `-c / --calib` option, to activate P.I.D control and set the library in debug mode we will set the flags `-C / --pid` and `-l / --dbg`. Lets then, run the test:

```
$> sudo ./mt_test -Cl -c15 -P6 -I2 -D1 -v90 -t8 -r20 -T11 > outs/t8
```

This test produces a very big output, I'll paste here snippets of it to explain how it behave.

```
PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 5
MOTOR_0: Set point: 2427, Mesured value: 2427, Physical error: 234, ticks_dt: 2, act_pw
    : 90, Last : 2, posCtrl: 0.000000, ratio rec: 0.000278, poscont: 0
MOTOR_0: Output value: 0
MOTOR_0: NO ACT: PIDout: 0, max_pw: 107 >= act_pw: 90 >= min_pw: 73
```

This is how an standard loop of the P.I.D algorithm will look like, or in other words, the information that it prints. As you can see it gives a lot of information, it prints the position control state, `posCtrl = X.XXX`, if there are two motors running at a time, hence we need to synchronize it `sinc = "boolean"` and the ticks spent till the moment `now_tk` in the first line. In the second line we can see the "traditional" P.I.D parameters `Set point` stands for the target time between ticks the motor must rotate ideally, the `Mesured value` is the actual time between ticks reading, this is the first loop for what the library equals it to the `Set Point` field, reality use to be far more cruel..., and the `Physical error` which stands for the physical error computed by the library as explained in the latter test. The other fields in the second line are used internally for the library, and have been useful for debug purposes. In the third line we can see the `Output value` which stands for the output of the P.I.D algorithm, it is computed as in a traditional P.I.D algorithm it is, $Kp * Prop + Ki * Integral + Kd * Derivative$, actually the only thing that varies here is way we get the measured value, since we do a partial mean for the time between ticks every `ticks_dt`. The last line is the result of this loop, it is, depending on the P.I.D output value it will reset the pulse to one of the two pulses we can see in the fields either `max_pw` if we are turning slowly than expected, it is, the error is a negative value greater than the (physical error * 3) in absolute value or `min_pw` if we are turning faster than expected, it is, the error is a positive value greater than the (physical error * 3). The `act_pw` stands for the actual pulse width we are applying to the motor, in this case the base pulse width that we apply to the motor, it is the pulse that we will apply when no error is detected, so the pulse widths aren't calculate dynamically at each loop, instead, we have a pre-computed maximal and minimal pulses we will set for each velocity and, if an error occurs, the algorithm will set the appropriate pulse depending on the error and remain in this state till the output of the P.I.D algorithm is smaller than the (physical error * 3) in absolute value. Due to this strange behavior of the algorithm is recommended to set the gains to a floating point value ranging from 0 to 1 to weight the Proportional, Integrative and Derivative errors as desired, otherwise what the user will do is downplay the physical error compensation, however, as we are testing and I'm interested on putting the board in a bad situation I set the gains far greater than 1.

Usually, when accelerating the motor a negative error will occur, due to the physical friction of the motor, it is, if we require velocity 90, as in this case, the motor won't turn at a desired velocity till a little time has elapsed, as any motor it have an acceleration time, curiously, due to our low frequency pulse we are facing the opposite scenario, we are facing a positive error at the beginning of the rotation, funny ain't it?. Here I leave you a snippet of the loops that accumulates the positive error:

```
PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 21
MOTOR_0: Set point: 2430, Mesured value: 1670, Physical error: 234, ticks_dt: 8, act_pw
    : 90, Last : 10, posCtrl: 0.000000, ratio rec: 0.001389, poscont: 0
MOTOR_0: Output value: 11637
clear_channel_gpio: channel=8, gpio=4
add_channel_pulse: channel=8, gpio=4, start=0, width=73
MOTOR_0: MIN: PIDout: 11637, max_pw: 107 >= act_pw: 73 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 37
MOTOR_0: Set point: 2430, Mesured value: 1670, Physical error: 234, ticks_dt: 8, act_pw
    : 73, Last : 18, posCtrl: 0.000000, ratio rec: 0.002500, poscont: 0
MOTOR_0: Output value: 19988
MOTOR_0: NO ACT: PIDout: 19988, max_pw: 107 >= act_pw: 73 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 52
MOTOR_0: Set point: 2430, Mesured value: 1671, Physical error: 234, ticks_dt: 8, act_pw
    : 73, Last : 26, posCtrl: 0.000000, ratio rec: 0.003611, poscont: 0
MOTOR_0: Output value: 28382
MOTOR_0: NO ACT: PIDout: 28382, max_pw: 107 >= act_pw: 73 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 66
MOTOR_0: Set point: 2430, Mesured value: 1911, Physical error: 234, ticks_dt: 7, act_pw
    : 73, Last : 33, posCtrl: 0.000000, ratio rec: 0.004583, poscont: 0
MOTOR_0: Output value: 30898
MOTOR_0: NO ACT: PIDout: 30898, max_pw: 107 >= act_pw: 73 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 79
MOTOR_0: Set point: 2430, Mesured value: 2673, Physical error: 234, ticks_dt: 5, act_pw
    : 73, Last : 39, posCtrl: 0.000000, ratio rec: 0.005417, poscont: 0
MOTOR_0: Output value: 27319
MOTOR_0: NO ACT: PIDout: 27319, max_pw: 107 >= act_pw: 73 >= min_pw: 73

        *
        *
        *
```

Really funny situation! if we think a little we can figure out what is going on here, our motor isn't magic, it have, as mentioned above, an acceleration time, however, this acceleration is always performed applying the maximum voltage, or at least a higher voltage than the expected, due to the low frequency of our pulses, so we turn faster than expected when the velocity demanded is not near the maximum velocity, is some kind of "automatic" acceleration error control, however our P.I.D algorithm doesn't expect that, and it is correct any way, we just face the opposite error we expect. Then, as you can see in the first loop, the algorithm sets the minimum pulse allowed, after this four more loops are spent till the appropriate time between ticks is detected, from this moment error will start recovering. Once a few loops has spent the error will be totally compensated and the base pulse width will be set:

```
        *
        *
        *

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 281
MOTOR_0: Set point: 2430, Mesured value: 2672, Physical error: 234, ticks_dt: 5, act_pw
    : 73, Last : 140, posCtrl: 0.000000, ratio rec: 0.019444, poscont: 0
MOTOR_0: Output value: 5764
```

```
MOTOR_0: NO ACT: PIDout: 5764, max_pw: 107 >= act_pw: 73 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 291
MOTOR_0: Set point: 2430, Mesured value: 2687, Physical error: 234, ticks_dt: 5, act_pw
    : 73, Last : 145, posCtrl: 0.000000, ratio rec: 0.020139, poscont: 0
MOTOR_0: Output value: 4391
MOTOR_0: NO ACT: PIDout: 4391, max_pw: 107 >= act_pw: 73 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 300
MOTOR_0: Set point: 2430, Mesured value: 2673, Physical error: 234, ticks_dt: 5, act_pw
    : 73, Last : 150, posCtrl: 0.000000, ratio rec: 0.020833, poscont: 0
MOTOR_0: Output value: 3340
MOTOR_0: NO ACT: PIDout: 3340, max_pw: 107 >= act_pw: 73 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 310
MOTOR_0: Set point: 2430, Mesured value: 2672, Physical error: 234, ticks_dt: 5, act_pw
    : 73, Last : 155, posCtrl: 0.000000, ratio rec: 0.021528, poscont: 0
MOTOR_0: Output value: 2214
MOTOR_0: NO ACT: PIDout: 2214, max_pw: 107 >= act_pw: 73 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 320
MOTOR_0: Set point: 2430, Mesured value: 2700, Physical error: 234, ticks_dt: 5, act_pw
    : 73, Last : 160, posCtrl: 0.000000, ratio rec: 0.022222, poscont: 0
MOTOR_0: Output value: 631
MOTOR_0: ENTRO BASE: PIDout: 631, PHbackup * 3 : 702
add_channel_pulse: channel=8, gpio=4, start=90, width=17
MOTOR_0: BASE: PIDout: 631, max_pw: 107 >= act_pw: 90 >= min_pw: 73, incr : 17

        *
        *
        *
```

As you can see in the `now_tk` field 320 ticks has spent till the velocity can be considered stabilized, the P.I.D implementation is encoder lines independent, so it is a roughly a turn, which is a lot to me, and the bad news doesn't end here, after the base pulse is set, due to the software generation of the pulse there is a lack time when resetting the pulse while the motor won't receive voltage, it is a very small time, however it will make the velocity fluctuate a little bit:

```
        *
        *
        *
PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 329
MOTOR_0: Set point: 2430, Mesured value: 3341, Physical error: 234, ticks_dt: 4, act_pw
    : 90, Last : 164, posCtrl: 0.000000, ratio rec: 0.022778, poscont: 0
MOTOR_0: Output value: -8130
add_channel_pulse: channel=8, gpio=4, start=90, width=17
MOTOR_0: MAX: PIDout: -8130, max_pw: 107 >= act_pw: 107 >= min_pw: 73, incr: 17

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 339
MOTOR_0: Set point: 2430, Mesured value: 2673, Physical error: 234, ticks_dt: 5, act_pw
    : 107, Last : 169, posCtrl: 0.000000, ratio rec: 0.023472, poscont: 0
MOTOR_0: Output value: -3945
MOTOR_0: NO ACT: PIDout: -3945, max_pw: 107 >= act_pw: 107 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 350
MOTOR_0: Set point: 2430, Mesured value: 2236, Physical error: 234, ticks_dt: 6, act_pw
    : 107, Last : 175, posCtrl: 0.000000, ratio rec: 0.024306, poscont: 0
MOTOR_0: Output value: -2824
MOTOR_0: NO ACT: PIDout: -2824, max_pw: 107 >= act_pw: 107 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 360
MOTOR_0: Set point: 2430, Mesured value: 2672, Physical error: 234, ticks_dt: 5, act_pw
    : 107, Last : 180, posCtrl: 0.000000, ratio rec: 0.025000, poscont: 0
MOTOR_0: Output value: -3374
MOTOR_0: NO ACT: PIDout: -3374, max_pw: 107 >= act_pw: 107 >= min_pw: 73
```

```
PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 370
MOTOR_0: Set point: 2430, Mesured value: 2672, Physical error: 234, ticks_dt: 5, act_pw
    : 107, Last : 185, posCtrl: 0.000000, ratio rec: 0.025694, poscont: 0
MOTOR_0: Output value: -3440
MOTOR_0: NO ACT: PIDout: -3440, max_pw: 107 >= act_pw: 107 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 381
MOTOR_0: Set point: 2430, Mesured value: 2676, Physical error: 234, ticks_dt: 5, act_pw
    : 107, Last : 190, posCtrl: 0.000000, ratio rec: 0.026389, poscont: 0
MOTOR_0: Output value: -3584
MOTOR_0: NO ACT: PIDout: -3584, max_pw: 107 >= act_pw: 107 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 391
MOTOR_0: Set point: 2430, Mesured value: 2673, Physical error: 234, ticks_dt: 5, act_pw
    : 107, Last : 195, posCtrl: 0.000000, ratio rec: 0.027083, poscont: 0
MOTOR_0: Output value: -3656
MOTOR_0: NO ACT: PIDout: -3656, max_pw: 107 >= act_pw: 107 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 402
MOTOR_0: Set point: 2430, Mesured value: 2227, Physical error: 234, ticks_dt: 6, act_pw
    : 107, Last : 201, posCtrl: 0.000000, ratio rec: 0.027917, poscont: 0
MOTOR_0: Output value: -2238
MOTOR_0: NO ACT: PIDout: -2238, max_pw: 107 >= act_pw: 107 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 413
MOTOR_0: Set point: 2430, Mesured value: 2672, Physical error: 234, ticks_dt: 5, act_pw
    : 107, Last : 206, posCtrl: 0.000000, ratio rec: 0.028611, poscont: 0
MOTOR_0: Output value: -2846
MOTOR_0: NO ACT: PIDout: -2846, max_pw: 107 >= act_pw: 107 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 424
MOTOR_0: Set point: 2430, Mesured value: 2227, Physical error: 234, ticks_dt: 6, act_pw
    : 107, Last : 212, posCtrl: 0.000000, ratio rec: 0.029444, poscont: 0
MOTOR_0: Output value: -1419
MOTOR_0: NO ACT: PIDout: -1419, max_pw: 107 >= act_pw: 107 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 435
MOTOR_0: Set point: 2430, Mesured value: 2672, Physical error: 234, ticks_dt: 5, act_pw
    : 107, Last : 217, posCtrl: 0.000000, ratio rec: 0.030139, poscont: 0
MOTOR_0: Output value: -2026
MOTOR_0: NO ACT: PIDout: -2026, max_pw: 107 >= act_pw: 107 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 446
MOTOR_0: Set point: 2430, Mesured value: 2227, Physical error: 234, ticks_dt: 6, act_pw
    : 107, Last : 223, posCtrl: 0.000000, ratio rec: 0.030972, poscont: 0
MOTOR_0: Output value: -599
MOTOR_0: ENTRO BASE: PIDout: -599, PHbackup * 3 : 702
clear_channel_gpio: channel=8, gpio=4
add_channel_pulse: channel=8, gpio=4, start=0, width=90
MOTOR_0: BASE: PIDout: -599, max_pw: 107 >= act_pw: 90 >= min_pw: 73, incr : 0

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 458
MOTOR_0: Set point: 2430, Mesured value: 2227, Physical error: 234, ticks_dt: 6, act_pw
    : 90, Last : 229, posCtrl: 0.000000, ratio rec: 0.031806, poscont: 0
MOTOR_0: Output value: -1074
add_channel_pulse: channel=8, gpio=4, start=90, width=17
MOTOR_0: MAX: PIDout: -1074, max_pw: 107 >= act_pw: 107 >= min_pw: 73, incr: 17

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 471
MOTOR_0: Set point: 2430, Mesured value: 2227, Physical error: 234, ticks_dt: 6, act_pw
    : 107, Last : 235, posCtrl: 0.000000, ratio rec: 0.032639, poscont: 0
MOTOR_0: Output value: 300
MOTOR_0: ENTRO BASE: PIDout: 300, PHbackup * 3 : 702
clear_channel_gpio: channel=8, gpio=4
add_channel_pulse: channel=8, gpio=4, start=0, width=90
MOTOR_0: BASE: PIDout: 300, max_pw: 107 >= act_pw: 90 >= min_pw: 73, incr : 0

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 484
```

```
MOTOR_0: Set point: 2430, Mesured value: 1909, Physical error: 234, ticks_dt: 7, act_pw
    : 90, Last : 242, posCtrl: 0.000000, ratio rec: 0.033611, poscont: 0
MOTOR_0: Output value: 5608
clear_channel_gpio: channel=8, gpio=4
add_channel_pulse: channel=8, gpio=4, start=0, width=73
MOTOR_0: MIN: PIDout: 5608, max_pw: 107 >= act_pw: 73 >= min_pw: 73

        *
        *
        *
```

Here we can see how, now, we accumulate a negative error, and at the end back to a positive error, the problem here is that every pulse switch is susceptible of this behavior, specially if we set the gains carelessly as I did this time, this execution keeps on fluctuating till the end of the travel. Lets now execute the test once again, however this time I will set the gains between 0 and 1, so we can wight the errors reading, and keep the physical error compensation, to see if we can make an execution keeping the velocity stable

```
$> sudo ./mt_test -Cl -c15 -P1 -I0.6 -D0.2 -v90 -t8 -r20 -T11 > outs/t8pon
```

This execution let this values:

```
PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 22
MOTOR_0: Set point: 2334, Mesured value: 1604, Physical error: 246, ticks_dt: 8, act_pw
    : 90, Last : 11, posCtrl: 0.000000, ratio rec: 0.001528, poscont: 0
MOTOR_0: Output value: 2819
clear_channel_gpio: channel=8, gpio=4
add_channel_pulse: channel=8, gpio=4, start=0, width=73
MOTOR_0: MIN: PIDout: 2819, max_pw: 107 >= act_pw: 73 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 37
MOTOR_0: Set point: 2334, Mesured value: 1833, Physical error: 246, ticks_dt: 7, act_pw
    : 73, Last : 18, posCtrl: 0.000000, ratio rec: 0.002500, poscont: 0
MOTOR_0: Output value: 3642
MOTOR_0: NO ACT: PIDout: 3642, max_pw: 107 >= act_pw: 73 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 52
MOTOR_0: Set point: 2334, Mesured value: 1604, Physical error: 246, ticks_dt: 8, act_pw
    : 73, Last : 26, posCtrl: 0.000000, ratio rec: 0.003611, poscont: 0
MOTOR_0: Output value: 6207
MOTOR_0: NO ACT: PIDout: 6207, max_pw: 107 >= act_pw: 73 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 66
MOTOR_0: Set point: 2334, Mesured value: 1834, Physical error: 246, ticks_dt: 7, act_pw
    : 73, Last : 33, posCtrl: 0.000000, ratio rec: 0.004583, poscont: 0
MOTOR_0: Output value: 7031
MOTOR_0: NO ACT: PIDout: 7031, max_pw: 107 >= act_pw: 73 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 79
MOTOR_0: Set point: 2334, Mesured value: 2567, Physical error: 246, ticks_dt: 5, act_pw
    : 73, Last : 39, posCtrl: 0.000000, ratio rec: 0.005417, poscont: 0
MOTOR_0: Output value: 6382
MOTOR_0: NO ACT: PIDout: 6382, max_pw: 107 >= act_pw: 73 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 92
MOTOR_0: Set point: 2334, Mesured value: 2567, Physical error: 246, ticks_dt: 5, act_pw
    : 73, Last : 46, posCtrl: 0.000000, ratio rec: 0.006389, poscont: 0
MOTOR_0: Output value: 6105
MOTOR_0: NO ACT: PIDout: 6105, max_pw: 107 >= act_pw: 73 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 104
MOTOR_0: Set point: 2334, Mesured value: 2567, Physical error: 246, ticks_dt: 5, act_pw
    : 73, Last : 52, posCtrl: 0.000000, ratio rec: 0.007222, poscont: 0
MOTOR_0: Output value: 5814
MOTOR_0: NO ACT: PIDout: 5814, max_pw: 107 >= act_pw: 73 >= min_pw: 73
```

```
PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 116
MOTOR_0: Set point: 2334, Mesured value: 2567, Physical error: 246, ticks_dt: 5, act_pw
    : 73, Last : 58, posCtrl: 0.000000, ratio rec: 0.008056, poscont: 0
MOTOR_0: Output value: 5523
MOTOR_0: NO ACT: PIDout: 5523, max_pw: 107 >= act_pw: 73 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 127
MOTOR_0: Set point: 2334, Mesured value: 2567, Physical error: 246, ticks_dt: 5, act_pw
    : 73, Last : 63, posCtrl: 0.000000, ratio rec: 0.008750, poscont: 0
MOTOR_0: Output value: 5232
MOTOR_0: NO ACT: PIDout: 5232, max_pw: 107 >= act_pw: 73 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 138
MOTOR_0: Set point: 2334, Mesured value: 2567, Physical error: 246, ticks_dt: 5, act_pw
    : 73, Last : 69, posCtrl: 0.000000, ratio rec: 0.009583, poscont: 0
MOTOR_0: Output value: 4941
MOTOR_0: NO ACT: PIDout: 4941, max_pw: 107 >= act_pw: 73 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 149
MOTOR_0: Set point: 2334, Mesured value: 2589, Physical error: 246, ticks_dt: 5, act_pw
    : 73, Last : 74, posCtrl: 0.000000, ratio rec: 0.010278, poscont: 0
MOTOR_0: Output value: 4561
MOTOR_0: NO ACT: PIDout: 4561, max_pw: 107 >= act_pw: 73 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 160
MOTOR_0: Set point: 2334, Mesured value: 2599, Physical error: 246, ticks_dt: 5, act_pw
    : 73, Last : 80, posCtrl: 0.000000, ratio rec: 0.011111, poscont: 0
MOTOR_0: Output value: 4164
MOTOR_0: NO ACT: PIDout: 4164, max_pw: 107 >= act_pw: 73 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 170
MOTOR_0: Set point: 2334, Mesured value: 2567, Physical error: 246, ticks_dt: 5, act_pw
    : 73, Last : 85, posCtrl: 0.000000, ratio rec: 0.011806, poscont: 0
MOTOR_0: Output value: 3907
MOTOR_0: NO ACT: PIDout: 3907, max_pw: 107 >= act_pw: 73 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 180
MOTOR_0: Set point: 2334, Mesured value: 2567, Physical error: 246, ticks_dt: 5, act_pw
    : 73, Last : 90, posCtrl: 0.000000, ratio rec: 0.012500, poscont: 0
MOTOR_0: Output value: 3615
MOTOR_0: NO ACT: PIDout: 3615, max_pw: 107 >= act_pw: 73 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 190
MOTOR_0: Set point: 2334, Mesured value: 2567, Physical error: 246, ticks_dt: 5, act_pw
    : 73, Last : 95, posCtrl: 0.000000, ratio rec: 0.013194, poscont: 0
MOTOR_0: Output value: 3324
MOTOR_0: NO ACT: PIDout: 3324, max_pw: 107 >= act_pw: 73 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 200
MOTOR_0: Set point: 2334, Mesured value: 2567, Physical error: 246, ticks_dt: 5, act_pw
    : 73, Last : 100, posCtrl: 0.000000, ratio rec: 0.013889, poscont: 0
MOTOR_0: Output value: 3033
MOTOR_0: NO ACT: PIDout: 3033, max_pw: 107 >= act_pw: 73 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 210
MOTOR_0: Set point: 2334, Mesured value: 2567, Physical error: 246, ticks_dt: 5, act_pw
    : 73, Last : 105, posCtrl: 0.000000, ratio rec: 0.014583, poscont: 0
MOTOR_0: Output value: 2742
MOTOR_0: NO ACT: PIDout: 2742, max_pw: 107 >= act_pw: 73 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 220
MOTOR_0: Set point: 2334, Mesured value: 2567, Physical error: 246, ticks_dt: 5, act_pw
    : 73, Last : 110, posCtrl: 0.000000, ratio rec: 0.015278, poscont: 0
MOTOR_0: Output value: 2451
MOTOR_0: NO ACT: PIDout: 2451, max_pw: 107 >= act_pw: 73 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 230
MOTOR_0: Set point: 2334, Mesured value: 2567, Physical error: 246, ticks_dt: 5, act_pw
```

```
    : 73, Last : 115, posCtrl: 0.000000, ratio rec: 0.015972, poscont: 0
MOTOR_0: Output value: 2160
MOTOR_0: NO ACT: PIDout: 2160, max_pw: 107 >= act_pw: 73 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 239
MOTOR_0: Set point: 2334, Mesured value: 3209, Physical error: 246, ticks_dt: 4, act_pw
    : 73, Last : 119, posCtrl: 0.000000, ratio rec: 0.016528, poscont: 0
MOTOR_0: Output value: -287
MOTOR_0: ENTRO BASE: PIDout: -287, PHbackup * 3 : 738
add_channel_pulse: channel=8, gpio=4, start=90, width=17
MOTOR_0: BASE: PIDout: -287, max_pw: 107 >= act_pw: 90 >= min_pw: 73, incr : 17

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 249
MOTOR_0: Set point: 2334, Mesured value: 2566, Physical error: 246, ticks_dt: 5, act_pw
    : 90, Last : 124, posCtrl: 0.000000, ratio rec: 0.017222, poscont: 0
MOTOR_0: Output value: 513
MOTOR_0: NO ACT: PIDout: 513, max_pw: 107 >= act_pw: 90 >= min_pw: 73

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 260
MOTOR_0: Set point: 2334, Mesured value: 2139, Physical error: 246, ticks_dt: 6, act_pw
    : 90, Last : 130, posCtrl: 0.000000, ratio rec: 0.018056, poscont: 0
MOTOR_0: Output value: 483
MOTOR_0: NO ACT: PIDout: 483, max_pw: 107 >= act_pw: 90 >= min_pw: 73

        *
        *
        *
```

Here we can see more clear, at the beginning we face the inverted acceleration issue, as in the first case, however this time once the minimum pulse is set it remains solving the error till the loop where we have 239 ticks, then the error is compensated and the base pulse set, After this we can see a little bit of fluctuation, which is compensated for the physical error value, not very clean I know, however it remains stable till the end of the travel.
Now lets see if this strange counter-acceleration issue is solved when high speed is demanded

```
$> sudo ./mt_test -Cl -c15 -P1 -I0.6 -D0.2 -v180 -t8 -r20 -T11 > outs/t8180
```

will generate an output like:

```
STARTING MOVE_T

add_channel_pulse: channel=8, gpio=4, start=0, width=180
PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 10
MOTOR_0: Set point: 1266, Mesured value: 1266, Physical error: 302, ticks_dt: 5, act_pw
    : 180, Last : 5, posCtrl: 0.000000, ratio rec: 0.000694, poscont: 0
MOTOR_0: Output value: 0
MOTOR_0: NO ACT: PIDout: 0, max_pw: 200 >= act_pw: 180 >= min_pw: 118

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 20
MOTOR_0: Set point: 1266, Mesured value: 1393, Physical error: 302, ticks_dt: 5, act_pw
    : 180, Last : 10, posCtrl: 0.000000, ratio rec: 0.001389, poscont: 0
MOTOR_0: Output value: 0
MOTOR_0: NO ACT: PIDout: 0, max_pw: 200 >= act_pw: 180 >= min_pw: 118

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 29
MOTOR_0: Set point: 1266, Mesured value: 1740, Physical error: 302, ticks_dt: 4, act_pw
    : 180, Last : 14, posCtrl: 0.000000, ratio rec: 0.001944, poscont: 0
MOTOR_0: Output value: -593
MOTOR_0: NO ACT: PIDout: -593, max_pw: 200 >= act_pw: 180 >= min_pw: 118

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 38
MOTOR_0: Set point: 1266, Mesured value: 1392, Physical error: 302, ticks_dt: 5, act_pw
    : 180, Last : 19, posCtrl: 0.000000, ratio rec: 0.002639, poscont: 0
MOTOR_0: Output value: -406
MOTOR_0: NO ACT: PIDout: -406, max_pw: 200 >= act_pw: 180 >= min_pw: 118
```

```
PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 49
MOTOR_0: Set point: 1266, Mesured value: 1393, Physical error: 302, ticks_dt: 5, act_pw
    : 180, Last : 24, posCtrl: 0.000000, ratio rec: 0.003333, poscont: 0
MOTOR_0: Output value: -412
MOTOR_0: NO ACT: PIDout: -412, max_pw: 200 >= act_pw: 180 >= min_pw: 118

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 59
MOTOR_0: Set point: 1266, Mesured value: 1392, Physical error: 302, ticks_dt: 5, act_pw
    : 180, Last : 29, posCtrl: 0.000000, ratio rec: 0.004028, poscont: 0
MOTOR_0: Output value: -412
MOTOR_0: NO ACT: PIDout: -412, max_pw: 200 >= act_pw: 180 >= min_pw: 118

PID_MOTOR_0: posCtrl = 0.000000, sinc = "false", now_tk = 68
MOTOR_0: Set point: 1266, Mesured value: 1392, Physical error: 302, ticks_dt: 5, act_pw
    : 180, Last : 34, posCtrl: 0.000000, ratio rec: 0.004722, poscont: 0
MOTOR_0: Output value: -412
MOTOR_0: NO ACT: PIDout: -412, max_pw: 200 >= act_pw: 180 >= min_pw: 118

        *
        *
        *
```

This time the acceleration issue is not happening, actually it happens, but as the velocity is high no error is detected, hence, no pulse reconfiguration is needed, hopefully, till the end of the travel.
As a final test for P.I.D control we will set position control to 0.3, it is, we will start breaking the motor in the last 30% of the travel, as we are demanding 20 turns, we will break for the last 6 turns till the end of the travel. When position control starts acting the algorithm will disable the derivative error to avoid fluctuations, for this execution I will show only the part when position control starts acting, you can see it in the `ratio_rec` field of the output, it must be `1-posCtrl`, then every ten loops the velocity will decrease till we arrive to the set point, I will leave here only the 30 first loops of the position control.

```
$> sudo ./mt_test -Cl -c15 -P1 -I0.6 -D0.2 -v90 -t8 -r20 -T11 -b0.3 > outs/t8psctr
```

has produced, this time, the following output:

```
        *
        *
        *

PID_MOTOR_0: posCtrl = 0.300000, sinc = "false", now_tk = 10091
MOTOR_0: Set point: 2295, Mesured value: 1912, Physical error: 252, ticks_dt: 9, act_pw
    : 90, Last : 5045, posCtrl: 0.300000, ratio rec: 0.700694, poscont: 1
MOTOR_0: Output value: 87
MOTOR_0: NO ACT: PIDout: 87, max_pw: 108 >= act_pw: 90 >= min_pw: 72

PID_MOTOR_0: posCtrl = 0.300000, sinc = "false", now_tk = 10111
MOTOR_0: Set point: 2295, Mesured value: 1721, Physical error: 252, ticks_dt: 10, act_pw
     : 90, Last : 5055, posCtrl: 0.300000, ratio rec: 0.702083, poscont: 2
MOTOR_0: Output value: 2211
clear_channel_gpio: channel=8, gpio=4
add_channel_pulse: channel=8, gpio=4, start=0, width=72
MOTOR_0: MIN: PIDout: 2211, max_pw: 108 >= act_pw: 72 >= min_pw: 72

PID_MOTOR_0: posCtrl = 0.300000, sinc = "false", now_tk = 10131
MOTOR_0: Set point: 2295, Mesured value: 2151, Physical error: 252, ticks_dt: 8, act_pw
    : 72, Last : 5065, posCtrl: 0.300000, ratio rec: 0.703472, poscont: 3
MOTOR_0: Output value: 1877
MOTOR_0: NO ACT: PIDout: 1877, max_pw: 108 >= act_pw: 72 >= min_pw: 72

PID_MOTOR_0: posCtrl = 0.300000, sinc = "false", now_tk = 10151
MOTOR_0: Set point: 2295, Mesured value: 2151, Physical error: 252, ticks_dt: 8, act_pw
    : 72, Last : 5075, posCtrl: 0.300000, ratio rec: 0.704861, poscont: 4
```

```
MOTOR_0: Output value: 1885
MOTOR_0: NO ACT: PIDout: 1885, max_pw: 108 >= act_pw: 72 >= min_pw: 72

PID_MOTOR_0: posCtrl = 0.300000, sinc = "false", now_tk = 10169
MOTOR_0: Set point: 2295, Mesured value: 2151, Physical error: 252, ticks_dt: 8, act_pw
    : 72, Last : 5084, posCtrl: 0.300000, ratio rec: 0.706111, poscont: 5
MOTOR_0: Output value: 1885
MOTOR_0: NO ACT: PIDout: 1885, max_pw: 108 >= act_pw: 72 >= min_pw: 72

PID_MOTOR_0: posCtrl = 0.300000, sinc = "false", now_tk = 10186
MOTOR_0: Set point: 2295, Mesured value: 2151, Physical error: 252, ticks_dt: 8, act_pw
    : 72, Last : 5093, posCtrl: 0.300000, ratio rec: 0.707361, poscont: 6
MOTOR_0: Output value: 1885
MOTOR_0: NO ACT: PIDout: 1885, max_pw: 108 >= act_pw: 72 >= min_pw: 72

PID_MOTOR_0: posCtrl = 0.300000, sinc = "false", now_tk = 10202
MOTOR_0: Set point: 2295, Mesured value: 2151, Physical error: 252, ticks_dt: 8, act_pw
    : 72, Last : 5101, posCtrl: 0.300000, ratio rec: 0.708472, poscont: 7
MOTOR_0: Output value: 1885
MOTOR_0: NO ACT: PIDout: 1885, max_pw: 108 >= act_pw: 72 >= min_pw: 72

PID_MOTOR_0: posCtrl = 0.300000, sinc = "false", now_tk = 10218
MOTOR_0: Set point: 2295, Mesured value: 2151, Physical error: 252, ticks_dt: 8, act_pw
    : 72, Last : 5109, posCtrl: 0.300000, ratio rec: 0.709583, poscont: 8
MOTOR_0: Output value: 1885
MOTOR_0: NO ACT: PIDout: 1885, max_pw: 108 >= act_pw: 72 >= min_pw: 72

PID_MOTOR_0: posCtrl = 0.300000, sinc = "false", now_tk = 10233
MOTOR_0: Set point: 2295, Mesured value: 2467, Physical error: 252, ticks_dt: 7, act_pw
    : 72, Last : 5116, posCtrl: 0.300000, ratio rec: 0.710556, poscont: 9
MOTOR_0: Output value: 1712
MOTOR_0: NO ACT: PIDout: 1712, max_pw: 108 >= act_pw: 72 >= min_pw: 72

PID_MOTOR_0: posCtrl = 0.300000, sinc = "false", now_tk = 10248
POSCTRL_0: newmin: 72, newbase: 80, newmax : 91, gain multiplier: 3.500000
MOTOR_0: Set point: 2433, Mesured value: 2151, Physical error: 252, ticks_dt: 8, act_pw
    : 72, Last : 5124, posCtrl: 0.300000, ratio rec: 0.711667, poscont: 0
MOTOR_0: Output value: 1920
MOTOR_0: NO ACT: PIDout: 1920, max_pw: 91 >= act_pw: 72 >= min_pw: 72

PID_MOTOR_0: posCtrl = 0.300000, sinc = "false", now_tk = 10263
MOTOR_0: Set point: 2433, Mesured value: 2459, Physical error: 252, ticks_dt: 7, act_pw
    : 72, Last : 5131, posCtrl: 0.300000, ratio rec: 0.712639, poscont: 1
MOTOR_0: Output value: 1890
MOTOR_0: NO ACT: PIDout: 1890, max_pw: 91 >= act_pw: 72 >= min_pw: 72

PID_MOTOR_0: posCtrl = 0.300000, sinc = "false", now_tk = 10277
MOTOR_0: Set point: 2433, Mesured value: 2459, Physical error: 252, ticks_dt: 7, act_pw
    : 72, Last : 5138, posCtrl: 0.300000, ratio rec: 0.713611, poscont: 2
MOTOR_0: Output value: 1890
MOTOR_0: NO ACT: PIDout: 1890, max_pw: 91 >= act_pw: 72 >= min_pw: 72

PID_MOTOR_0: posCtrl = 0.300000, sinc = "false", now_tk = 10291
MOTOR_0: Set point: 2433, Mesured value: 2458, Physical error: 252, ticks_dt: 7, act_pw
    : 72, Last : 5145, posCtrl: 0.300000, ratio rec: 0.714583, poscont: 3
MOTOR_0: Output value: 1890
MOTOR_0: NO ACT: PIDout: 1890, max_pw: 91 >= act_pw: 72 >= min_pw: 72

PID_MOTOR_0: posCtrl = 0.300000, sinc = "false", now_tk = 10305
MOTOR_0: Set point: 2433, Mesured value: 2459, Physical error: 252, ticks_dt: 7, act_pw
    : 72, Last : 5152, posCtrl: 0.300000, ratio rec: 0.715556, poscont: 4
MOTOR_0: Output value: 1890
MOTOR_0: NO ACT: PIDout: 1890, max_pw: 91 >= act_pw: 72 >= min_pw: 72

PID_MOTOR_0: posCtrl = 0.300000, sinc = "false", now_tk = 10319
MOTOR_0: Set point: 2433, Mesured value: 2459, Physical error: 252, ticks_dt: 7, act_pw
    : 72, Last : 5159, posCtrl: 0.300000, ratio rec: 0.716528, poscont: 5
MOTOR_0: Output value: 1890
```

4.1. Motor testing

```
MOTOR_0: NO ACT: PIDout: 1890, max_pw: 91 >= act_pw: 72 >= min_pw: 72

PID_MOTOR_0: posCtrl = 0.300000, sinc = "false", now_tk = 10332
MOTOR_0: Set point: 2433, Mesured value: 2459, Physical error: 252, ticks_dt: 7, act_pw
    : 72, Last : 5166, posCtrl: 0.300000, ratio rec: 0.717500, poscont: 6
MOTOR_0: Output value: 1890
MOTOR_0: NO ACT: PIDout: 1890, max_pw: 91 >= act_pw: 72 >= min_pw: 72

PID_MOTOR_0: posCtrl = 0.300000, sinc = "false", now_tk = 10345
MOTOR_0: Set point: 2433, Mesured value: 2868, Physical error: 252, ticks_dt: 6, act_pw
    : 72, Last : 5172, posCtrl: 0.300000, ratio rec: 0.718333, poscont: 7
MOTOR_0: Output value: 529
MOTOR_0: ENTRO BASE: PIDout: 529, PHbackup * 3 : 756
add_channel_pulse: channel=8, gpio=4, start=80, width=8
MOTOR_0: BASE: PIDout: 529, max_pw: 91 >= act_pw: 80 >= min_pw: 72, incr : 8

PID_MOTOR_0: posCtrl = 0.300000, sinc = "false", now_tk = 10359
MOTOR_0: Set point: 2433, Mesured value: 2465, Physical error: 252, ticks_dt: 7, act_pw
    : 80, Last : 5179, posCtrl: 0.300000, ratio rec: 0.719306, poscont: 8
MOTOR_0: Output value: 825
clear_channel_gpio: channel=8, gpio=4
add_channel_pulse: channel=8, gpio=4, start=0, width=72
MOTOR_0: MIN: PIDout: 825, max_pw: 91 >= act_pw: 72 >= min_pw: 72

PID_MOTOR_0: posCtrl = 0.300000, sinc = "false", now_tk = 10371
MOTOR_0: Set point: 2433, Mesured value: 2868, Physical error: 252, ticks_dt: 6, act_pw
    : 72, Last : 5185, posCtrl: 0.300000, ratio rec: 0.720139, poscont: 9
MOTOR_0: Output value: -536
MOTOR_0: ENTRO BASE: PIDout: -536, PHbackup * 3 : 756
add_channel_pulse: channel=8, gpio=4, start=80, width=8
MOTOR_0: BASE: PIDout: -536, max_pw: 91 >= act_pw: 80 >= min_pw: 72, incr : 8

PID_MOTOR_0: posCtrl = 0.300000, sinc = "false", now_tk = 10384
POSCTRL_0: newmin: 72, newbase: 77, newmax : 86, gain multiplier: 3.500000
MOTOR_0: Set point: 2552, Mesured value: 2458, Physical error: 252, ticks_dt: 7, act_pw
    : 80, Last : 5192, posCtrl: 0.300000, ratio rec: 0.721111, poscont: 0
MOTOR_0: Output value: -240
MOTOR_0: ENTRO BASE: PIDout: -240, PHbackup * 3 : 756
clear_channel_gpio: channel=8, gpio=4
add_channel_pulse: channel=8, gpio=4, start=0, width=77
MOTOR_0: BASE: PIDout: -240, max_pw: 86 >= act_pw: 77 >= min_pw: 72, incr : 0

PID_MOTOR_0: posCtrl = 0.300000, sinc = "false", now_tk = 10397
MOTOR_0: Set point: 2552, Mesured value: 2868, Physical error: 252, ticks_dt: 6, act_pw
    : 77, Last : 5198, posCtrl: 0.300000, ratio rec: 0.721944, poscont: 1
MOTOR_0: Output value: -535
MOTOR_0: NO ACT: PIDout: -535, max_pw: 86 >= act_pw: 77 >= min_pw: 72

PID_MOTOR_0: posCtrl = 0.300000, sinc = "false", now_tk = 10410
MOTOR_0: Set point: 2552, Mesured value: 2459, Physical error: 252, ticks_dt: 7, act_pw
    : 77, Last : 5205, posCtrl: 0.300000, ratio rec: 0.722917, poscont: 2
MOTOR_0: Output value: -471
MOTOR_0: NO ACT: PIDout: -471, max_pw: 86 >= act_pw: 77 >= min_pw: 72

PID_MOTOR_0: posCtrl = 0.300000, sinc = "false", now_tk = 10423
MOTOR_0: Set point: 2552, Mesured value: 2869, Physical error: 252, ticks_dt: 6, act_pw
    : 77, Last : 5211, posCtrl: 0.300000, ratio rec: 0.723750, poscont: 3
MOTOR_0: Output value: -770
add_channel_pulse: channel=8, gpio=4, start=77, width=9
MOTOR_0: MAX: PIDout: -770, max_pw: 86 >= act_pw: 86 >= min_pw: 72, incr: 9

PID_MOTOR_0: posCtrl = 0.300000, sinc = "false", now_tk = 10437
MOTOR_0: Set point: 2552, Mesured value: 2458, Physical error: 252, ticks_dt: 7, act_pw
    : 86, Last : 5218, posCtrl: 0.300000, ratio rec: 0.724722, poscont: 4
MOTOR_0: Output value: -705
MOTOR_0: ENTRO BASE: PIDout: -705, PHbackup * 3 : 756
clear_channel_gpio: channel=8, gpio=4
add_channel_pulse: channel=8, gpio=4, start=0, width=77
```

```
MOTOR_0: BASE: PIDout: -705, max_pw: 86 >= act_pw: 77 >= min_pw: 72, incr : 0

PID_MOTOR_0: posCtrl = 0.300000, sinc = "false", now_tk = 10453
MOTOR_0: Set point: 2552, Mesured value: 2155, Physical error: 252, ticks_dt: 8, act_pw
    : 77, Last : 5226, posCtrl: 0.300000, ratio rec: 0.725833, poscont: 5
MOTOR_0: Output value: 136
MOTOR_0: NO ACT: PIDout: 136, max_pw: 86 >= act_pw: 77 >= min_pw: 72

PID_MOTOR_0: posCtrl = 0.300000, sinc = "false", now_tk = 10471
MOTOR_0: Set point: 2552, Mesured value: 2151, Physical error: 252, ticks_dt: 8, act_pw
    : 77, Last : 5235, posCtrl: 0.300000, ratio rec: 0.727083, poscont: 6
MOTOR_0: Output value: 855
clear_channel_gpio: channel=8, gpio=4
add_channel_pulse: channel=8, gpio=4, start=0, width=72
MOTOR_0: MIN: PIDout: 855, max_pw: 86 >= act_pw: 72 >= min_pw: 72

PID_MOTOR_0: posCtrl = 0.300000, sinc = "false", now_tk = 10490
MOTOR_0: Set point: 2552, Mesured value: 2151, Physical error: 252, ticks_dt: 8, act_pw
    : 72, Last : 5245, posCtrl: 0.300000, ratio rec: 0.728472, poscont: 7
MOTOR_0: Output value: 1570
MOTOR_0: NO ACT: PIDout: 1570, max_pw: 86 >= act_pw: 72 >= min_pw: 72

PID_MOTOR_0: posCtrl = 0.300000, sinc = "false", now_tk = 10508
MOTOR_0: Set point: 2552, Mesured value: 2151, Physical error: 252, ticks_dt: 8, act_pw
    : 72, Last : 5254, posCtrl: 0.300000, ratio rec: 0.729722, poscont: 8
MOTOR_0: Output value: 2285
MOTOR_0: NO ACT: PIDout: 2285, max_pw: 86 >= act_pw: 72 >= min_pw: 72

PID_MOTOR_0: posCtrl = 0.300000, sinc = "false", now_tk = 10525
MOTOR_0: Set point: 2552, Mesured value: 2151, Physical error: 252, ticks_dt: 8, act_pw
    : 72, Last : 5262, posCtrl: 0.300000, ratio rec: 0.730833, poscont: 9
MOTOR_0: Output value: 3000
MOTOR_0: NO ACT: PIDout: 3000, max_pw: 86 >= act_pw: 72 >= min_pw: 72

PID_MOTOR_0: posCtrl = 0.300000, sinc = "false", now_tk = 10542
POSCTRL_0: newmin: 70, newbase: 72, newmax : 81, gain multiplier: 3.500000

        *
        *
        *
```

As you can see position control starts acting when the `1-posCtrl %` of the travel has been run, every ten loops, you can see the tracking in the `poscont` field, the position control will increase the set point of the P.I.D algorithm, then it will take care of reducing the velocity to reach the new set point hence the motor will be breaking gradually till we arrive to the set point.

With our pseudo P.I.D control explained I'll finish the motor testing with motor synchronization, we will use test number 10 for this, test number 9 is designed for motor synchronization too, however it will turn indefinitely and it is more interesting to test synchro till a set point, so we can test all the functionalities. When motor synchronization is acting P.I.D will stop acting, else the further can affect the latter and vice versa, hence for this test we will disable P.I.D, and we will test motor synchronization for a 160 velocity which is a presumably high velocity, when our board will suffer the more, after this we will test motor synchronization and position control at a time to see if both motors brake at the same time. To run the test we will need the `-p / --port` option set to 2, since both motors are needed for the `mt_move_sinc_t` function, as mentioned we will set, for the first run, the velocity to 160, the synchronization of two motors will act the following, we will have a global structure dedicated to this functionality, in it will reside a flag, the first motor to arrive to the synchronization part of the code for first time will take the flag, after this, if it is needed, the motor will brake or accelerate, however the motor to change its velocity doesn't have to be necessarily the one that holds the flag. Lets then run the test

```
$> sudo ./mt_test -l -t10 -v160 -r20 -p2 > outs/t10
```

This time is interesting to redirect the output to a file, then with the file `t10` produced

```
$> cat outs/t10 | grep SINCRO
```

will produce an output similar to:

```
SINCRO: FLAG0 = TRUE, first -> 0
SINCRO: FLAG1 = TRUE, first -> 0
SINCRO_0: tticks > 980, difft/tticks : 21, t1 > 319, t2 > 307, delay1: 312626.000000,
    delay2: 291329.000000, diffdelay : 21297, minpw : 98, basepw : 161, maxpw : 200
SINCRO_0: tticks > 780, difft/tticks : 0, t1 > 1052, t2 > 1078, delay1: 820984.000000,
    delay2: 820512.000000, diffdelay : 472, minpw : 98, basepw : 161, maxpw : 200
SINCRO_0: tticks > 870, difft/tticks : 1, t1 > 1224, t2 > 1257, delay1: 1065825.000000,
    delay2: 1064841.000000, diffdelay : 984, minpw : 98, basepw : 161, maxpw : 200
SINCRO_0 ACEL: newmin: 116, newbase: 184, newmax : 200
SINCRO_0: tticks > 934, difft/tticks : 11, t1 > 1414, t2 > 1438, delay1: 1320969.000000,
     delay2: 1310187.000000, diffdelay : 10782, minpw : 98, basepw : 161, maxpw : 200
SINCRO_0 RES: newmin: 98, newbase: 161, newmax : 200
SINCRO_0: tticks > 982, difft/tticks : 10, t1 > 1594, t2 > 1622, delay1: 1566438.000000,
     delay2: 1556495.000000, diffdelay : 9943, minpw : 98, basepw : 161, maxpw : 200
SINCRO_0: tticks > 1024, difft/tticks : 9, t1 > 1768, t2 > 1803, delay1: 1811232.000000,
     delay2: 1801732.000000, diffdelay : 9500, minpw : 98, basepw : 161, maxpw : 200
SINCRO_0 ACEL: newmin: 116, newbase: 184, newmax : 200
SINCRO_0: tticks > 1056, difft/tticks : 19, t1 > 1956, t2 > 1984, delay1:
    2066760.000000, delay2: 2045659.000000, diffdelay : 21101, minpw : 98, basepw :
    161, maxpw : 200
SINCRO_0 RES: newmin: 98, newbase: 161, newmax : 200
SINCRO_0: tticks > 1081, difft/tticks : 18, t1 > 2137, t2 > 2167, delay1:
    2310246.000000, delay2: 2289864.000000, diffdelay : 20382, minpw : 98, basepw :
    161, maxpw : 200
SINCRO_0: tticks > 1104, difft/tticks : 17, t1 > 2312, t2 > 2348, delay1:
    2554460.000000, delay2: 2535261.000000, diffdelay : 19199, minpw : 98, basepw :
    161, maxpw : 200
SINCRO_0 ACEL: newmin: 116, newbase: 184, newmax : 200
SINCRO_0: tticks > 1121, difft/tticks : 28, t1 > 2506, t2 > 2528, delay1:
    2811309.000000, delay2: 2779669.000000, diffdelay : 31640, minpw : 98, basepw :
    161, maxpw : 200
SINCRO_0 RES: newmin: 98, newbase: 161, newmax : 200
SINCRO_0: tticks > 1137, difft/tticks : 27, t1 > 2686, t2 > 2713, delay1:
    3055916.000000, delay2: 3024352.000000, diffdelay : 31564, minpw : 98, basepw :
    161, maxpw : 200
SINCRO_0: tticks > 1152, difft/tticks : 26, t1 > 2863, t2 > 2894, delay1:
    3299921.000000, delay2: 3268900.000000, diffdelay : 31021, minpw : 98, basepw :
    161, maxpw : 200
SINCRO_0 ACEL: newmin: 116, newbase: 184, newmax : 200
SINCRO_0: tticks > 1163, difft/tticks : 35, t1 > 3056, t2 > 3076, delay1:
    3555918.000000, delay2: 3514471.000000, diffdelay : 41447, minpw : 98, basepw :
    161, maxpw : 200
SINCRO_0 RES: newmin: 98, newbase: 161, newmax : 200
SINCRO_0: tticks > 1173, difft/tticks : 34, t1 > 3237, t2 > 3260, delay1:
    3799775.000000, delay2: 3759031.000000, diffdelay : 40744, minpw : 98, basepw :
    161, maxpw : 200
SINCRO_0: tticks > 1184, difft/tticks : 34, t1 > 3413, t2 > 3440, delay1:
    4044197.000000, delay2: 4003005.000000, diffdelay : 41192, minpw : 98, basepw :
    161, maxpw : 200
SINCRO_0: tticks > 1194, difft/tticks : 34, t1 > 3590, t2 > 3621, delay1:
    4287723.000000, delay2: 4247123.000000, diffdelay : 40600, minpw : 98, basepw :
    161, maxpw : 200
SINCRO_0 ACEL: newmin: 116, newbase: 184, newmax : 200
SINCRO_0: tticks > 1201, difft/tticks : 41, t1 > 3778, t2 > 3803, delay1:
    4540658.000000, delay2: 4490653.000000, diffdelay : 50005, minpw : 98, basepw :
    161, maxpw : 200
SINCRO_0 RES: newmin: 98, newbase: 161, newmax : 200
SINCRO_0: tticks > 1207, difft/tticks : 40, t1 > 3962, t2 > 3989, delay1:
    4784214.000000, delay2: 4735332.000000, diffdelay : 48882, minpw : 98, basepw :
    161, maxpw : 200
SINCRO_0: tticks > 1214, difft/tticks : 39, t1 > 4140, t2 > 4175, delay1:
```

```
    5028268.000000, delay2: 4980444.000000, diffdelay : 47824, minpw : 98, basepw :
    161, maxpw : 200
SINCRO_0 ACEL: newmin: 116, newbase: 184, newmax : 200
SINCRO_0: tticks > 1219, difft/tticks : 47, t1 > 4332, t2 > 4361, delay1:
    5284193.000000, delay2: 5225925.000000, diffdelay : 58268, minpw : 98, basepw :
    161, maxpw : 200
SINCRO_0 RES: newmin: 98, newbase: 161, newmax : 200
SINCRO_0: tticks > 1225, difft/tticks : 48, t1 > 4512, t2 > 4542, delay1:
    5529140.000000, delay2: 5469901.000000, diffdelay : 59239, minpw : 98, basepw :
    161, maxpw : 200
SINCRO_0: tticks > 1231, difft/tticks : 48, t1 > 4688, t2 > 4728, delay1:
    5774923.000000, delay2: 5715219.000000, diffdelay : 59704, minpw : 98, basepw :
    161, maxpw : 200
SINCRO_0 ACEL: newmin: 116, newbase: 184, newmax : 200
SINCRO_0: tticks > 1235, difft/tticks : 56, t1 > 4880, t2 > 4908, delay1:
    6028891.000000, delay2: 5959450.000000, diffdelay : 69441, minpw : 98, basepw :
    161, maxpw : 200
SINCRO_0 RES: newmin: 98, newbase: 161, newmax : 200
SINCRO_0: tticks > 1239, difft/tticks : 55, t1 > 5063, t2 > 5095, delay1:
    6274455.000000, delay2: 6206091.000000, diffdelay : 68364, minpw : 98, basepw :
    161, maxpw : 200
SINCRO_0 ACEL: newmin: 116, newbase: 184, newmax : 200
SINCRO_0: tticks > 1242, difft/tticks : 66, t1 > 5256, t2 > 5276, delay1:
    6531647.000000, delay2: 6449427.000000, diffdelay : 82220, minpw : 98, basepw :
    161, maxpw : 200
SINCRO_0 RES: newmin: 98, newbase: 161, newmax : 200
SINCRO_0: tticks > 1246, difft/tticks : 65, t1 > 5436, t2 > 5457, delay1:
    6775065.000000, delay2: 6693304.000000, diffdelay : 81761, minpw : 98, basepw :
    161, maxpw : 200
SINCRO_0: tticks > 1250, difft/tticks : 64, t1 > 5612, t2 > 5639, delay1:
    7018831.000000, delay2: 6937615.000000, diffdelay : 81216, minpw : 98, basepw :
    161, maxpw : 200
SINCRO_0: tticks > 1254, difft/tticks : 66, t1 > 5790, t2 > 5822, delay1:
    7264369.000000, delay2: 7181278.000000, diffdelay : 83091, minpw : 98, basepw :
    161, maxpw : 200
SINCRO_0 ACEL: newmin: 116, newbase: 184, newmax : 200
SINCRO_0: tticks > 1256, difft/tticks : 73, t1 > 5985, t2 > 6006, delay1:
    7518636.000000, delay2: 7426237.000000, diffdelay : 92399, minpw : 98, basepw :
    161, maxpw : 200
SINCRO_0 RES: newmin: 98, newbase: 161, newmax : 200
SINCRO_0: tticks > 1258, difft/tticks : 73, t1 > 6170, t2 > 6190, delay1:
    7761936.000000, delay2: 7669526.000000, diffdelay : 92410, minpw : 98, basepw :
    161, maxpw : 200
SINCRO_0: tticks > 1261, difft/tticks : 73, t1 > 6346, t2 > 6372, delay1:
    8006074.000000, delay2: 7913372.000000, diffdelay : 92702, minpw : 98, basepw :
    161, maxpw : 200
SINCRO_0: tticks > 1264, difft/tticks : 72, t1 > 6523, t2 > 6554, delay1:
    8249638.000000, delay2: 8158252.000000, diffdelay : 91386, minpw : 98, basepw :
    161, maxpw : 200
SINCRO_0 ACEL: newmin: 116, newbase: 184, newmax : 200
SINCRO_0: tticks > 1267, difft/tticks : 79, t1 > 6712, t2 > 6734, delay1:
    8505107.000000, delay2: 8404756.000000, diffdelay : 100351, minpw : 98, basepw :
    161, maxpw : 200
SINCRO_0 RES: newmin: 98, newbase: 161, newmax : 200
SINCRO_0: tticks > 1269, difft/tticks : 77, t1 > 6892, t2 > 6917, delay1:
    8749308.000000, delay2: 8650776.000000, diffdelay : 98532, minpw : 98, basepw :
    161, maxpw : 200
SINCRO_0: tticks > 1272, difft/tticks : 77, t1 > 7070, t2 > 7097, delay1:
    8993199.000000, delay2: 8895200.000000, diffdelay : 97999, minpw : 98, basepw :
    161, maxpw : 200
```

As you can see this time the motor 0 arrived first, the library will allow till a gap up to 30 ticks, this gap however is computed in two ways, first we measure the ticks difference, after that we check two time stamps, updated by the motors before every re-stabilization, if the delay between the time stamps divided for the time between ticks spent is bigger than 30, as mentioned earlier, a re-stabilization will occur, depending if the motor that is being calibrated is going faster or

slower than the other one it will brake or accelerate. If a re-stabilization has occurred once both motors have a small enough gap of ticks the library will reset the velocity to its original state, from the output above the important fields are `difft/tticks`, the `t1` and `t2`, which stands for the difference between delays divided for the time between ticks, and the ticks for the motor 0 and 1 respectively, as you can see the synchronization is working pretty well for a high velocity, both motors ends with a difference of 27 ticks which is more than acceptable. Only to ensure it I'll make a new run at a velocity of 90

```
$> sudo ./mt_test -l -t10 -v90 -r20 -p2 > outs/t1090
```

As above

```
$> cat outs/t1090 | grep SINCRO > outs
```

will produce an output like:

```
SINCRO: FLAG1 = TRUE, first -> 1
SINCRO: FLAG0 = TRUE, first -> 1
SINCRO_1: tticks > 1387, difft/tticks : 26, t1 > 317, t2 > 354, delay1: 454800.000000,
    delay2: 491143.000000, diffdelay : 36343, minpw : 61, basepw : 91, maxpw : 121
SINCRO_1: tticks > 1623, difft/tticks : 24, t1 > 520, t2 > 554, delay1: 860104.000000,
    delay2: 899562.000000, diffdelay : 39458, minpw : 61, basepw : 91, maxpw : 121
SINCRO_1: tticks > 1780, difft/tticks : 21, t1 > 692, t2 > 732, delay1: 1266073.000000,
    delay2: 1303476.000000, diffdelay : 37403, minpw : 61, basepw : 91, maxpw : 121
SINCRO_1: tticks > 1871, difft/tticks : 19, t1 > 865, t2 > 912, delay1: 1671424.000000,
    delay2: 1706986.000000, diffdelay : 35562, minpw : 61, basepw : 91, maxpw : 121
SINCRO_1: tticks > 1933, difft/tticks : 16, t1 > 1037, t2 > 1092, delay1:
    2078767.000000, delay2: 2111533.000000, diffdelay : 32766, minpw : 61, basepw : 91,
     maxpw : 121
SINCRO_1 FRENA: newmin: 58, newbase: 87, newmax : 116
SINCRO_1: tticks > 1984, difft/tticks : 16, t1 > 1211, t2 > 1269, delay1:
    2485481.000000, delay2: 2518251.000000, diffdelay : 32770, minpw : 61, basepw : 91,
     maxpw : 121
SINCRO_1 FRENA: newmin: 56, newbase: 83, newmax : 110
SINCRO_1: tticks > 2035, difft/tticks : 17, t1 > 1384, t2 > 1439, delay1:
    2893465.000000, delay2: 2928384.000000, diffdelay : 34919, minpw : 61, basepw : 91,
     maxpw : 121
SINCRO_1 FRENA: newmin: 53, newbase: 78, newmax : 103
SINCRO_1: tticks > 2084, difft/tticks : 15, t1 > 1556, t2 > 1599, delay1:
    3301016.000000, delay2: 3333859.000000, diffdelay : 32843, minpw : 61, basepw : 91,
     maxpw : 121
SINCRO_1 RES: newmin: 61, newbase: 91, newmax : 121
SINCRO_1: tticks > 2102, difft/tticks : 16, t1 > 1731, t2 > 1779, delay1:
    3705699.000000, delay2: 3740755.000000, diffdelay : 35056, minpw : 61, basepw : 91,
     maxpw : 121
SINCRO_1 FRENA: newmin: 58, newbase: 87, newmax : 116
SINCRO_1: tticks > 2121, difft/tticks : 16, t1 > 1903, t2 > 1954, delay1:
    4110982.000000, delay2: 4146312.000000, diffdelay : 35330, minpw : 61, basepw : 91,
     maxpw : 121
SINCRO_1 FRENA: newmin: 56, newbase: 83, newmax : 110
SINCRO_1: tticks > 2141, difft/tticks : 17, t1 > 2074, t2 > 2126, delay1:
    4515545.000000, delay2: 4552631.000000, diffdelay : 37086, minpw : 61, basepw : 91,
     maxpw : 121
SINCRO_1 FRENA: newmin: 53, newbase: 78, newmax : 103
SINCRO_1: tticks > 2166, difft/tticks : 17, t1 > 2245, t2 > 2289, delay1:
    4921507.000000, delay2: 4959365.000000, diffdelay : 37858, minpw : 61, basepw : 91,
     maxpw : 121
SINCRO_1 RES: newmin: 61, newbase: 91, newmax : 121
SINCRO_1: tticks > 2180, difft/tticks : 18, t1 > 2416, t2 > 2461, delay1:
    5326625.000000, delay2: 5366996.000000, diffdelay : 40371, minpw : 61, basepw : 91,
     maxpw : 121
SINCRO_1: tticks > 2188, difft/tticks : 17, t1 > 2588, t2 > 2637, delay1:
    5733068.000000, delay2: 5771305.000000, diffdelay : 38237, minpw : 61, basepw : 91,
     maxpw : 121
SINCRO_1 FRENA: newmin: 58, newbase: 87, newmax : 116
SINCRO_1: tticks > 2197, difft/tticks : 17, t1 > 2758, t2 > 2811, delay1:
```

```
    6138403.000000, delay2: 6177513.000000, diffdelay : 39110, minpw : 61, basepw : 91,
     maxpw : 121
SINCRO_1 FRENA: newmin: 56, newbase: 83, newmax : 110
SINCRO_1: tticks > 2209, difft/tticks : 16, t1 > 2929, t2 > 2979, delay1:
    6546718.000000, delay2: 6583019.000000, diffdelay : 36301, minpw : 61, basepw : 91,
     maxpw : 121
SINCRO_1 FRENA: newmin: 53, newbase: 78, newmax : 103
SINCRO_1: tticks > 2223, difft/tticks : 15, t1 > 3101, t2 > 3144, delay1:
    6954249.000000, delay2: 6989655.000000, diffdelay : 35406, minpw : 61, basepw : 91,
     maxpw : 121
SINCRO_1 RES: newmin: 61, newbase: 91, newmax : 121
SINCRO_1: tticks > 2227, difft/tticks : 15, t1 > 3274, t2 > 3319, delay1:
    7359139.000000, delay2: 7393730.000000, diffdelay : 34591, minpw : 61, basepw : 91,
     maxpw : 121
SINCRO_1: tticks > 2230, difft/tticks : 15, t1 > 3446, t2 > 3497, delay1:
    7765076.000000, delay2: 7800249.000000, diffdelay : 35173, minpw : 61, basepw : 91,
     maxpw : 121
SINCRO_1 FRENA: newmin: 58, newbase: 87, newmax : 116
SINCRO_1: tticks > 2236, difft/tticks : 14, t1 > 3618, t2 > 3669, delay1:
    8173119.000000, delay2: 8204990.000000, diffdelay : 31871, minpw : 61, basepw : 91,
     maxpw : 121
SINCRO_1 FRENA: newmin: 56, newbase: 83, newmax : 110
SINCRO_1: tticks > 2244, difft/tticks : 13, t1 > 3791, t2 > 3837, delay1:
    8580415.000000, delay2: 8611360.000000, diffdelay : 30945, minpw : 61, basepw : 91,
     maxpw : 121
SINCRO_1 FRENA: newmin: 53, newbase: 78, newmax : 103
SINCRO_1: tticks > 2256, difft/tticks : 13, t1 > 3962, t2 > 3996, delay1:
    8986176.000000, delay2: 9017694.000000, diffdelay : 31518, minpw : 61, basepw : 91,
     maxpw : 121
SINCRO_1 RES: newmin: 61, newbase: 91, newmax : 121
SINCRO_1: tticks > 2261, difft/tticks : 15, t1 > 4134, t2 > 4168, delay1:
    9391050.000000, delay2: 9425227.000000, diffdelay : 34177, minpw : 61, basepw : 91,
     maxpw : 121
SINCRO_1: tticks > 2262, difft/tticks : 15, t1 > 4310, t2 > 4346, delay1:
    9796703.000000, delay2: 9831990.000000, diffdelay : 35287, minpw : 61, basepw : 91,
     maxpw : 121
SINCRO_1: tticks > 2262, difft/tticks : 15, t1 > 4481, t2 > 4524, delay1:
    10203010.000000, delay2: 10237792.000000, diffdelay : 34782, minpw : 61, basepw :
    91, maxpw : 121
SINCRO_1: tticks > 2264, difft/tticks : 16, t1 > 4654, t2 > 4700, delay1:
    10608025.000000, delay2: 10644786.000000, diffdelay : 36761, minpw : 61, basepw :
    91, maxpw : 121
SINCRO_1: tticks > 2266, difft/tticks : 16, t1 > 4826, t2 > 4876, delay1:
    11014046.000000, delay2: 11051057.000000, diffdelay : 37011, minpw : 61, basepw :
    91, maxpw : 121
SINCRO_1 FRENA: newmin: 58, newbase: 87, newmax : 116
SINCRO_1: tticks > 2268, difft/tticks : 15, t1 > 4998, t2 > 5050, delay1:
    11420468.000000, delay2: 11454787.000000, diffdelay : 34319, minpw : 61, basepw :
    91, maxpw : 121
SINCRO_1 FRENA: newmin: 56, newbase: 83, newmax : 110
SINCRO_1: tticks > 2272, difft/tticks : 14, t1 > 5168, t2 > 5219, delay1:
    11827997.000000, delay2: 11861630.000000, diffdelay : 33633, minpw : 61, basepw :
    91, maxpw : 121
SINCRO_1 FRENA: newmin: 53, newbase: 78, newmax : 103
SINCRO_1: tticks > 2279, difft/tticks : 15, t1 > 5340, t2 > 5382, delay1:
    12233291.000000, delay2: 12267661.000000, diffdelay : 34370, minpw : 61, basepw :
    91, maxpw : 121
SINCRO_1 RES: newmin: 61, newbase: 91, newmax : 121
SINCRO_1: tticks > 2281, difft/tticks : 15, t1 > 5510, t2 > 5554, delay1:
    12638823.000000, delay2: 12673240.000000, diffdelay : 34417, minpw : 61, basepw :
    91, maxpw : 121
SINCRO_1: tticks > 2282, difft/tticks : 15, t1 > 5683, t2 > 5729, delay1:
    13042856.000000, delay2: 13078246.000000, diffdelay : 35390, minpw : 61, basepw :
    91, maxpw : 121
SINCRO_1 FRENA: newmin: 58, newbase: 87, newmax : 116
SINCRO_1: tticks > 2283, difft/tticks : 16, t1 > 5854, t2 > 5905, delay1:
    13447615.000000, delay2: 13485832.000000, diffdelay : 38217, minpw : 61, basepw :
    91, maxpw : 121
```

```
SINCRO_1 FRENA: newmin: 56, newbase: 83, newmax : 110
SINCRO_1: tticks > 2287, difft/tticks : 17, t1 > 6025, t2 > 6073, delay1:
    13852160.000000, delay2: 13891530.000000, diffdelay : 39370, minpw : 61, basepw :
    91, maxpw : 121
SINCRO_1 FRENA: newmin: 53, newbase: 78, newmax : 103
SINCRO_1: tticks > 2294, difft/tticks : 17, t1 > 6196, t2 > 6231, delay1:
    14256748.000000, delay2: 14296412.000000, diffdelay : 39664, minpw : 61, basepw :
    91, maxpw : 121
SINCRO_1 RES: newmin: 61, newbase: 91, newmax : 121
SINCRO_1: tticks > 2294, difft/tticks : 16, t1 > 6366, t2 > 6406, delay1:
    14662350.000000, delay2: 14700313.000000, diffdelay : 37963, minpw : 61, basepw :
    91, maxpw : 121
SINCRO_1: tticks > 2295, difft/tticks : 15, t1 > 6539, t2 > 6580, delay1:
    15069699.000000, delay2: 15105470.000000, diffdelay : 35771, minpw : 61, basepw :
    91, maxpw : 121
SINCRO_1: tticks > 2295, difft/tticks : 16, t1 > 6709, t2 > 6758, delay1:
    15473487.000000, delay2: 15512355.000000, diffdelay : 38868, minpw : 61, basepw :
    91, maxpw : 121
SINCRO_1 FRENA: newmin: 58, newbase: 87, newmax : 116
SINCRO_1: tticks > 2297, difft/tticks : 18, t1 > 6882, t2 > 6930, delay1:
    15877676.000000, delay2: 15919828.000000, diffdelay : 42152, minpw : 61, basepw :
    91, maxpw : 121
SINCRO_1 RES: newmin: 61, newbase: 91, newmax : 121
SINCRO_1: tticks > 2296, difft/tticks : 18, t1 > 7054, t2 > 7110, delay1:
    16284407.000000, delay2: 16326706.000000, diffdelay : 42299, minpw : 61, basepw :
    91, maxpw : 121
SINCRO_1 FRENA: newmin: 58, newbase: 87, newmax : 116
```

This time as you can see the first motor to take the flag was the motor 1, and again everything is working pretty well, this time however both motors ends with a difference of 44 ticks, which is more than the allowed by the library, but trying to decrease it, it isn't the best but it is fine enough to me, 44 ticks in a travel of 7200 (per encoder line) is not bad at all. Well then to finish with motor testing I'll make one more run with position control enabled and, as we did for P.I.D, set to 0.3, I will set the velocity to 90.

```
$> sudo ./mt_test -l -t10 -v90 -r20 -p2 -b0.3 > outs/t10pos
```

As earlier

```
$> cat outs/t10pos | grep SINCRO
```

will produce, this time, an output like:

```
SINCRO: FLAG0 = TRUE, first -> 0
SINCRO: FLAG1 = TRUE, first -> 0
SINCRO_0: tticks > 1514, difft/tticks : 0, t1 > 301, t2 > 323, delay1: 455810.000000,
    delay2: 454947.000000, diffdelay : 863, minpw : 61, basepw : 91, maxpw : 121
SINCRO_1: tticks > 1613, difft/tticks : 0, t1 > 497, t2 > 534, delay1: 860130.000000,
    delay2: 861386.000000, diffdelay : 1256, minpw : 61, basepw : 91, maxpw : 121
SINCRO_1 FRENA: newmin: 58, newbase: 87, newmax : 116
SINCRO_1: tticks > 1788, difft/tticks : 1, t1 > 664, t2 > 709, delay1: 1265832.000000,
    delay2: 1268220.000000, diffdelay : 2388, minpw : 61, basepw : 91, maxpw : 121
SINCRO_1 FRENA: newmin: 56, newbase: 83, newmax : 110
SINCRO_1: tticks > 1903, difft/tticks : 1, t1 > 830, t2 > 879, delay1: 1670481.000000,
    delay2: 1672739.000000, diffdelay : 2258, minpw : 61, basepw : 91, maxpw : 121
SINCRO_1 FRENA: newmin: 53, newbase: 78, newmax : 103
SINCRO_1: tticks > 1999, difft/tticks : 1, t1 > 999, t2 > 1039, delay1: 2074832.000000,
    delay2: 2077798.000000, diffdelay : 2966, minpw : 61, basepw : 91, maxpw : 121
SINCRO_1 FRENA: newmin: 49, newbase: 74, newmax : 99
SINCRO_1: tticks > 2077, difft/tticks : 1, t1 > 1166, t2 > 1195, delay1: 2480888.000000,
     delay2: 2482987.000000, diffdelay : 2099, minpw : 61, basepw : 91, maxpw : 121
SINCRO_1 RES: newmin: 61, newbase: 91, newmax : 121
SINCRO_1: tticks > 2119, difft/tticks : 0, t1 > 1335, t2 > 1363, delay1: 2886786.000000,
     delay2: 2888246.000000, diffdelay : 1460, minpw : 61, basepw : 91, maxpw : 121
SINCRO_1: tticks > 2139, difft/tticks : 1, t1 > 1501, t2 > 1540, delay1: 3291731.000000,
     delay2: 3294490.000000, diffdelay : 2759, minpw : 61, basepw : 91, maxpw : 121
SINCRO_1 FRENA: newmin: 58, newbase: 87, newmax : 116
```

```
SINCRO_1: tticks > 2147, difft/tticks : 3, t1 > 1671, t2 > 1725, delay1: 3697049.000000,
     delay2: 3703927.000000, diffdelay : 6878, minpw : 61, basepw : 91, maxpw : 121
SINCRO_1 FRENA: newmin: 56, newbase: 83, newmax : 110
SINCRO_1: tticks > 2168, difft/tticks : 2, t1 > 1840, t2 > 1895, delay1: 4103717.000000,
     delay2: 4109662.000000, diffdelay : 5945, minpw : 61, basepw : 91, maxpw : 121
SINCRO_1 FRENA: newmin: 53, newbase: 78, newmax : 103
SINCRO_1: tticks > 2194, difft/tticks : 2, t1 > 2009, t2 > 2057, delay1: 4509243.000000,
     delay2: 4514549.000000, diffdelay : 5306, minpw : 61, basepw : 91, maxpw : 121
SINCRO_1 FRENA: newmin: 49, newbase: 74, newmax : 99
SINCRO_1: tticks > 2225, difft/tticks : 2, t1 > 2178, t2 > 2211, delay1: 4915526.000000,
     delay2: 4921404.000000, diffdelay : 5878, minpw : 61, basepw : 91, maxpw : 121
SINCRO_1 FRENA: newmin: 46, newbase: 70, newmax : 94
SINCRO_1: tticks > 2257, difft/tticks : 2, t1 > 2345, t2 > 2360, delay1: 5323016.000000,
     delay2: 5327782.000000, diffdelay : 4766, minpw : 61, basepw : 91, maxpw : 121
SINCRO_1 RES: newmin: 61, newbase: 91, newmax : 121
SINCRO_1: tticks > 2269, difft/tticks : 2, t1 > 2513, t2 > 2527, delay1: 5727796.000000,
     delay2: 5734308.000000, diffdelay : 6512, minpw : 61, basepw : 91, maxpw : 121
SINCRO_1: tticks > 2269, difft/tticks : 3, t1 > 2681, t2 > 2706, delay1: 6133851.000000,
     delay2: 6141945.000000, diffdelay : 8094, minpw : 61, basepw : 91, maxpw : 121
SINCRO_1: tticks > 2272, difft/tticks : 4, t1 > 2848, t2 > 2882, delay1: 6539699.000000,
     delay2: 6550103.000000, diffdelay : 10404, minpw : 61, basepw : 91, maxpw : 121
SINCRO_1: tticks > 2273, difft/tticks : 4, t1 > 3017, t2 > 3060, delay1: 6945949.000000,
     delay2: 6955689.000000, diffdelay : 9740, minpw : 61, basepw : 91, maxpw : 121
SINCRO_1 FRENA: newmin: 58, newbase: 87, newmax : 116
SINCRO_1: tticks > 2274, difft/tticks : 3, t1 > 3186, t2 > 3238, delay1: 7354808.000000,
     delay2: 7363252.000000, diffdelay : 8444, minpw : 61, basepw : 91, maxpw : 121
SINCRO_1 FRENA: newmin: 56, newbase: 83, newmax : 110
SINCRO_1: tticks > 2278, difft/tticks : 3, t1 > 3354, t2 > 3410, delay1: 7761245.000000,
     delay2: 7768615.000000, diffdelay : 7370, minpw : 61, basepw : 91, maxpw : 121
SINCRO_1 FRENA: newmin: 53, newbase: 78, newmax : 103
SINCRO_1: tticks > 2289, difft/tticks : 2, t1 > 3522, t2 > 3570, delay1: 8168198.000000,
     delay2: 8172953.000000, diffdelay : 4755, minpw : 61, basepw : 91, maxpw : 121
SINCRO_1 FRENA: newmin: 49, newbase: 74, newmax : 99
SINCRO_1: tticks > 2303, difft/tticks : 2, t1 > 3691, t2 > 3724, delay1: 8572786.000000,
     delay2: 8578276.000000, diffdelay : 5490, minpw : 61, basepw : 91, maxpw : 121
SINCRO_1 FRENA: newmin: 46, newbase: 70, newmax : 94
SINCRO_1: tticks > 2320, difft/tticks : 3, t1 > 3859, t2 > 3872, delay1: 8977880.000000,
     delay2: 8986245.000000, diffdelay : 8365, minpw : 61, basepw : 91, maxpw : 121
SINCRO_1 RES: newmin: 61, newbase: 91, newmax : 121
SINCRO_1: tticks > 2325, difft/tticks : 4, t1 > 4028, t2 > 4039, delay1: 9382983.000000,
     delay2: 9394218.000000, diffdelay : 11235, minpw : 61, basepw : 91, maxpw : 121
SINCRO_1: tticks > 2323, difft/tticks : 4, t1 > 4195, t2 > 4217, delay1: 9788752.000000,
     delay2: 9800242.000000, diffdelay : 11490, minpw : 61, basepw : 91, maxpw : 121
SINCRO_1: tticks > 2322, difft/tticks : 4, t1 > 4363, t2 > 4394, delay1:
    10195174.000000, delay2: 10206558.000000, diffdelay : 11384, minpw : 61, basepw :
    91, maxpw : 121
SINCRO_1: tticks > 2322, difft/tticks : 5, t1 > 4534, t2 > 4571, delay1:
    10603781.000000, delay2: 10615467.000000, diffdelay : 11686, minpw : 61, basepw :
    91, maxpw : 121
SINCRO_1 FRENA: newmin: 58, newbase: 87, newmax : 116
SINCRO_1: tticks > 2321, difft/tticks : 6, t1 > 4701, t2 > 4748, delay1:
    11009594.000000, delay2: 11023689.000000, diffdelay : 14095, minpw : 61, basepw :
    91, maxpw : 121
SINCRO_1 FRENA: newmin: 56, newbase: 83, newmax : 110
SINCRO_1: tticks > 2325, difft/tticks : 5, t1 > 4871, t2 > 4914, delay1:
    11416923.000000, delay2: 11429867.000000, diffdelay : 12944, minpw : 61, basepw :
    91, maxpw : 121
SINCRO_1 FRENA: newmin: 53, newbase: 78, newmax : 103
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.700972, cont: 0, actpw: " base ",
    arrived1: " true ", arrived2: " false ", ticks_rec : 10094
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.702986, cont: 1, actpw: " base ",
    arrived1: " true ", arrived2: " false ", ticks_rec : 10123
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.704931, cont: 2, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 10151
SINCRO_1: tticks > 2332, difft/tticks : 5, t1 > 5040, t2 > 5075, delay1:
    11823325.000000, delay2: 11836060.000000, diffdelay : 12735, minpw : 61, basepw :
    91, maxpw : 121
SINCRO_1 RES: newmin: 61, newbase: 91, newmax : 121
```

```
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.706875, cont: 3, actpw: " max ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 10179
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.708889, cont: 4, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 10208
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.710972, cont: 5, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 10238
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.713264, cont: 6, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 10271
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.715347, cont: 7, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 10301
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.717500, cont: 8, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 10332
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.719792, cont: 9, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 10365
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.722014, cont: 0, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 10397
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.724236, cont: 1, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 10429
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.726389, cont: 2, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 10460
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.728472, cont: 3, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 10490
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.730625, cont: 4, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 10521
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.732708, cont: 5, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 10551
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.734722, cont: 6, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 10580
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.736736, cont: 7, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 10609
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.738819, cont: 8, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 10639
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.740903, cont: 9, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 10669
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.742917, cont: 0, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 10698
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.744931, cont: 1, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 10727
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.747014, cont: 2, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 10757
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.748958, cont: 3, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 10785
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.750903, cont: 4, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 10813
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.752917, cont: 5, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 10842
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.754931, cont: 6, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 10871
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.756944, cont: 7, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 10900
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.758958, cont: 8, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 10929
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.760972, cont: 9, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 10958
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.762986, cont: 0, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 10987
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.764861, cont: 1, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11014
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.766736, cont: 2, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11041
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.768611, cont: 3, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11068
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.770486, cont: 4, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11095
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.772361, cont: 5, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11122
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.774167, cont: 6, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11148
```

```
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.776181, cont: 7, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11177
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.777917, cont: 8, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11202
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.779792, cont: 9, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11229
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.781736, cont: 0, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11257
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.783542, cont: 1, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11283
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.785417, cont: 2, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11310
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.787222, cont: 3, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11336
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.789028, cont: 4, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11362
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.790972, cont: 5, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11390
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.792847, cont: 6, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11417
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.794653, cont: 7, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11443
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.796528, cont: 8, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11470
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.798264, cont: 9, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11495
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.800069, cont: 0, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11521
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.801806, cont: 1, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11546
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.803611, cont: 2, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11572
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.805417, cont: 3, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11598
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.807222, cont: 4, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11624
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.808958, cont: 5, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11649
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.810833, cont: 6, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11676
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.812639, cont: 7, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11702
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.814375, cont: 8, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11727
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.816181, cont: 9, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11753
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.817986, cont: 0, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11779
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.819653, cont: 1, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11803
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.821458, cont: 2, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11829
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.823125, cont: 3, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11853
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.824861, cont: 4, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11878
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.826597, cont: 5, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11903
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.828333, cont: 6, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11928
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.830000, cont: 7, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11952
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.831667, cont: 8, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11976
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.833264, cont: 9, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 11999
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.834931, cont: 0, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12023
```

```
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.836597, cont: 1, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12047
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.838264, cont: 2, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12071
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.839931, cont: 3, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12095
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.841597, cont: 4, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12119
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.843472, cont: 5, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12146
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.845139, cont: 6, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12170
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.846875, cont: 7, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12195
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.848681, cont: 8, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12221
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.850347, cont: 9, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12245
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.851944, cont: 0, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12268
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.853611, cont: 1, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12292
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.855208, cont: 2, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12315
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.856806, cont: 3, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12338
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.858472, cont: 4, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12362
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.860000, cont: 5, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12384
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.861667, cont: 6, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12408
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.863333, cont: 7, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12432
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.865000, cont: 8, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12456
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.866597, cont: 9, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12479
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.868194, cont: 0, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12502
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.869722, cont: 1, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12524
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.871389, cont: 2, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12548
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.872917, cont: 3, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12570
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.874444, cont: 4, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12592
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.876042, cont: 5, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12615
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.877569, cont: 6, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12637
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.879167, cont: 7, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12660
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.880694, cont: 8, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12682
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.882222, cont: 9, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12704
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.883889, cont: 0, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12728
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.885625, cont: 1, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12753
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.887222, cont: 2, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12776
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.888819, cont: 3, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12799
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.890347, cont: 4, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12821
```

```
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.891806, cont: 5, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12842
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.893403, cont: 6, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12865
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.895000, cont: 7, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12888
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.896458, cont: 8, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12909
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.897986, cont: 9, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12931
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.899583, cont: 0, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12954
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.901111, cont: 1, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12976
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.902708, cont: 2, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 12999
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.904306, cont: 3, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13022
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.905833, cont: 4, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13044
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.907361, cont: 5, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13066
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.908889, cont: 6, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13088
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.910417, cont: 7, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13110
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.911944, cont: 8, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13132
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.913472, cont: 9, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13154
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.914931, cont: 0, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13175
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.916389, cont: 1, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13196
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.917917, cont: 2, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13218
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.919375, cont: 3, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13239
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.920833, cont: 4, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13260
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.922292, cont: 5, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13281
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.923819, cont: 6, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13303
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.925278, cont: 7, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13324
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.926736, cont: 8, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13345
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.928194, cont: 9, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13366
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.929653, cont: 0, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13387
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.931042, cont: 1, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13407
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.932431, cont: 2, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13427
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.933889, cont: 3, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13448
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.935278, cont: 4, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13468
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.936736, cont: 5, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13489
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.938194, cont: 6, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13510
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.939583, cont: 7, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13530
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.941042, cont: 8, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13551
```

```
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.942431, cont: 9, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13571
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.943819, cont: 0, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13591
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.945278, cont: 1, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13612
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.946667, cont: 2, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13632
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.948194, cont: 3, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13654
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.949583, cont: 4, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13674
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.951042, cont: 5, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13695
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.952431, cont: 6, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13715
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.953889, cont: 7, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13736
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.955278, cont: 8, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13756
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.956806, cont: 9, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13778
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.958194, cont: 0, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13798
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.959583, cont: 1, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13818
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.960972, cont: 2, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13838
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.962361, cont: 3, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13858
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.963750, cont: 4, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13878
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.965139, cont: 5, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13898
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.966528, cont: 6, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13918
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.967917, cont: 7, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13938
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.969306, cont: 8, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13958
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.970694, cont: 9, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13978
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.972014, cont: 0, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 13997
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.973333, cont: 1, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 14016
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.974722, cont: 2, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 14036
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.976111, cont: 3, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 14056
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.977431, cont: 4, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 14075
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.978750, cont: 5, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 14094
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.980139, cont: 6, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 14114
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.981528, cont: 7, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 14134
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.982917, cont: 8, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 14154
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.984236, cont: 9, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 14173
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.985556, cont: 0, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 14192
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.986944, cont: 1, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 14212
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.988194, cont: 2, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 14230
```

```
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.989583, cont: 3, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 14250
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.990972, cont: 4, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 14270
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.992292, cont: 5, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 14289
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.993611, cont: 6, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 14308
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.995000, cont: 7, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 14328
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.996319, cont: 8, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 14347
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.997708, cont: 9, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 14367
SINCRO_1_POS: first: 1,  posCtrl: 0.300000, spend: 0.998958, cont: 0, actpw: " base ",
    arrived1: " true ", arrived2: " true ", ticks_rec : 14385
```

As you can see, synchronization is working pretty well, it does its job, and both motors arrive to the start breaking point with a difference of about 44 ticks again, then the motor 1, the one holding the flag will be the one in charge of breaking both motors at a time, this time the output is not very explanatory, but the motors break at the same time.

As a final note say that all this tests has been performed powering the montage with a transformer 220V to 9V and without applying any weight to the motors so the results may be some kind of ideal, anyway weigh will probably disturb for the behavior of the motors but it will probably reduce little errors at not very high velocities.

4.2 Analog sensors testing

For the analog testing the file containing the sources can be found in the `tests` directory of LEGO-Pi git-hub , and it is named `analog_tests.c`, the same Makefile that take care of the motor test compilation will produce the executable file in a straightforward way for you, once in the directory containing the sources

```
$> make ag_test
```

This test provides us with 6 different tests, to choose the one we want to run we must use the mandatory `-t / --test` option, additionally there is three more optional parameter that we can set, some of them will apply to some tests only, I leave you here a printing of the usage instructions for the test:

```
$> sudo ./ag_test

Usage: ./ag_test -t<test_num> [-pid]
  -t --test      test number to perform [1-6]           {mandatory}
  -p --port      analog port [0-3]                      {0}
  -i --times     iterations, only apply to some tests   {6}
  -d --dbg       Set the library in debug mode [no arg] {disabled}
```

The default options are what you can see at the end of every line enclosed in brackets.

Lets then begin with the first test, for this tests the `-p / --port` option will be ignored, instead you will need two LEGO light sensors plugged to the analog port numbers 1 and 3 and a LEGO push sensor plugged to the analog port 0, in this tests we will wait for the user to push the LEGO push sensor, once this is done, if the led of the LEGO light sensors are off it will enable it, if the leds are enabled, it will disable it, after this the test will perform a voltage reading on every port with a LEGO light sensor attached and print the results and the voltage difference between the readings, the procedure will be repeated the times requested trough the `-i / --times`

option. For the light sensors the higher the voltage reading is the less light is being detected , so, to check the behavior of the light sensors I will expose the light sensor on port 1 to a white cardboard, while the light sensor attached to port 3 will be towards a green colored cardboard, if everything works well the light sensor faced to the white cardboard must report a lower voltage reading, since white reflects more light than green, and the readings must be more precise when the led diode that provides extra light is enabled on both light sensors, since I'm running the test in a cloudy gloomy day an no extra ambient light is being provided. To run the test I will use the default setting for the `-i / --times` option, so

```
$> sudo ./ag_test -t1
```

has produced this time the following output:

```
Setting ligh ON
PUSH_VAL: 0.98, LIGHT_1 says: 2.07, LIGHT_3 says: 3.72, VDIFF = 1.65

Setting ligh OFF
PUSH_VAL: 1.66, LIGHT_1 says: 4.09, LIGHT_3 says: 4.11, VDIFF = 0.02

Setting ligh ON
PUSH_VAL: 0.91, LIGHT_1 says: 2.08, LIGHT_3 says: 3.73, VDIFF = 1.66

Setting ligh OFF
PUSH_VAL: 0.91, LIGHT_1 says: 4.09, LIGHT_3 says: 4.09, VDIFF = 0.00

Setting ligh ON
PUSH_VAL: 1.29, LIGHT_1 says: 2.06, LIGHT_3 says: 3.71, VDIFF = 1.65

Setting ligh OFF
PUSH_VAL: 1.78, LIGHT_1 says: 4.09, LIGHT_3 says: 4.11, VDIFF = 0.03
```

As you can see everything is working as expected, every block of the output stands for a push sensor pushing, when no extra light is provided the sensors detect almost no light on both cases, as expected, however when we enable the led of the sensors the voltage provided for the light sensor on port 1, the one facing the white cardboard, is smaller than the voltage reading of the light sensor on port 3, the one towards the green colored cardboard, and we are getting about 1,65 V of difference between the readings, which is nice because we will be able to difference colors easily with our light sensor. Nevertheless I've to say that the sensors assembly is critical here for the correct behavior of it, I'm putting the cardboards separated a few millimeters of the light sensor and with a little inclination, maybe 5 or 10 degrees respect the vertical axis, however with a proper assembly we can get a fair enough assumption of the light detected for the sensors. As a final test I'll change the green colored cardboard for a light orange colored one, to see how the voltage readings from the sensor varies, and what will be the voltage range between colors, it is, if with a green colored cardboard we have a voltage difference of 1,65 V lets see what a difference we are able to get with an orange colored cardboard, it must be less, since the light emitted for the led is red, the command to run the test is the same that above, and the output produced this time was:

```
Setting ligh ON
PUSH_VAL: 1.51, LIGHT_1 says: 2.05, LIGHT_3 says: 2.46, VDIFF = 0.41

Setting ligh OFF
PUSH_VAL: 1.32, LIGHT_1 says: 4.07, LIGHT_3 says: 4.09, VDIFF = 0.02

Setting ligh ON
PUSH_VAL: 1.87, LIGHT_1 says: 2.05, LIGHT_3 says: 2.48, VDIFF = 0.42

Setting ligh OFF
PUSH_VAL: 1.07, LIGHT_1 says: 4.09, LIGHT_3 says: 4.10, VDIFF = 0.01
```

```
Setting ligh ON
PUSH_VAL: 1.72, LIGHT_1 says: 2.06, LIGHT_3 says: 2.46, VDIFF = 0.40

Setting ligh OFF
PUSH_VAL: 0.90, LIGHT_1 says: 4.10, LIGHT_3 says: 4.10, VDIFF = 0.00
```

As you can see we still vewing a difference between both light sensors, however this time ts less than the above execution, as expected, as a final test, I will run the test with a blue colored cardboard towards the light sensor on port 3, then if the voltage readings difference, respect the readings of the light reflected for a white cardboard, aren't the same than for the green we will be able to say that we have a range of voltage that allow us to difference R.G.B (red, green and blue) palette, the light sensor isn't meant to be used for this purpose, however as a test, to ensure that the light sensors behave as expected is not bad, again the line to run the test was the same as the above two cases, this time, with the blue colored cardboard towards the light sensor on port 3, the results was the following:

```
Setting ligh ON
PUSH_VAL: 1.50, LIGHT_1 says: 2.09, LIGHT_3 says: 3.65, VDIFF = 1.56

Setting ligh OFF
PUSH_VAL: 1.84, LIGHT_1 says: 4.09, LIGHT_3 says: 4.11, VDIFF = 0.02

Setting ligh ON
PUSH_VAL: 1.88, LIGHT_1 says: 2.10, LIGHT_3 says: 3.62, VDIFF = 1.52

Setting ligh OFF
PUSH_VAL: 1.61, LIGHT_1 says: 4.09, LIGHT_3 says: 4.11, VDIFF = 0.02

Setting ligh ON
PUSH_VAL: 1.22, LIGHT_1 says: 2.09, LIGHT_3 says: 3.63, VDIFF = 1.54

Setting ligh OFF
PUSH_VAL: 1.08, LIGHT_1 says: 4.10, LIGHT_3 says: 4.09, VDIFF = 0.01
```

So as you can see this time we are getting a difference of 1,51 V between the light reflected for a white cardboard and the light reflected for a blue colored cardboard, so the behavior of the light sensors is fair, however if we want to use it, as an instance, to detect different colors we must set the ranges for each color very accurately, as you can see the difference for green was about 1.65 V while the difference for blue is about 1.55 V, it means than the difference between green and blue will be about 0.10 V, what may lead us to mistake a color hue when colors are similar.

I'll skip test number two, it is designed to check if the user can control the led of a LEGO light sensor on demand, however the latter tests proves it and, actually, the outputs are more explanatory than with test number 2, Lets then run test number 3, this test will loop as many times as demanded trough the `-i / --iter` option, at every loop will make a dB reading of a sound sensor attached to the analog port requested with the option `-p / --port` and sleep for a second. Here I've to say that I don't have a dB sound meter to check if the decibels reading are accurate or not, however I will run the test without exposing the sensor to any noise first, and, after this I'll run the test exposing the sensor to different volume noises, actually, what I will do is enclose the sensor between a surround headphones, with a siren sound playing at different volumes. So, for the first run, without applying any noise to the sensor but the ambient sounds, which I can say that aren't very loud

```
$> sudo ./ag_test -t3 -p2
```

The results are:

```
SOUND says: 8 dB, read_int = 3743, read_volt = 4.575946
SOUND says: 7 dB, read_int = 3801, read_volt = 4.661294
SOUND says: 7 dB, read_int = 3787, read_volt = 4.642979
SOUND says: 6 dB, read_int = 3815, read_volt = 4.657998
SOUND says: 7 dB, read_int = 3814, read_volt = 4.654090
SOUND says: 7 dB, read_int = 3803, read_volt = 4.650672
```

As you can see the more voltage reported for the sensor the less noise is being detected, as you will intuit the `read_volt` field stands for the voltage received from the sensor, while the `read_int` is the raw value returned by the A/D converter, which will range from 0 to 4095 with our 12 bits of precision. From the LEGO sound sensor specification we know that the maximum dB reading will be of 100 dB, so, assuming that in this case the voltage returned will be about 0 V, and if no noise detected the voltage returned will be about 5 V, the voltage reference, we can make an assumption of the decibels detected by the sensor, this is what LEGO-Pi does at least.

For the further tests the sound used to test the sensor was downloaded from here, the volume control has been done with the alsamixer program available on any GNU-Linux distribution, to run the test this time the line was

```
$> sudo ./ag_test -t3 -p2 -i20
```

and to play the sound repeatedly from your computer

```
$> for i in `seq 1 10`; do mpg123 siren.mp3; done
```

the results at maximum volume was the following:

```
SOUND says: 92 dB, read_int = 302, read_volt = 0.377534
SOUND says: 81 dB, read_int = 753, read_volt = 0.933700
SOUND says: 85 dB, read_int = 610, read_volt = 0.731136
SOUND says: 89 dB, read_int = 430, read_volt = 0.531868
SOUND says: 93 dB, read_int = 280, read_volt = 0.337241
SOUND says: 91 dB, read_int = 364, read_volt = 0.443956
SOUND says: 76 dB, read_int = 955, read_volt = 1.162393
SOUND says: 86 dB, read_int = 553, read_volt = 0.675336
SOUND says: 87 dB, read_int = 523, read_volt = 0.624420
SOUND says: 84 dB, read_int = 618, read_volt = 0.788645
SOUND says: 67 dB, read_int = 1315, read_volt = 1.613065
SOUND says: 82 dB, read_int = 724, read_volt = 0.892308
SOUND says: 86 dB, read_int = 566, read_volt = 0.688278
SOUND says: 89 dB, read_int = 424, read_volt = 0.512332
SOUND says: 93 dB, read_int = 270, read_volt = 0.327106
SOUND says: 91 dB, read_int = 351, read_volt = 0.433333
SOUND says: 76 dB, read_int = 955, read_volt = 1.164957
SOUND says: 86 dB, read_int = 542, read_volt = 0.672772
SOUND says: 87 dB, read_int = 499, read_volt = 0.622222
SOUND says: 82 dB, read_int = 736, read_volt = 0.911233
```

As you can see we are getting a higher decibels reading, as expected, and readings are more or less according to the fluctuations of sounds, now lets decrease the volume to the half to see if the readings decrease accordingly, to run the test an play the sound you can do as in the latter test, to decrease the volume, as mentioned above, alsamixer utility has been used you can do it this way, or whatever way you want while the volume decreases, the results found are the following:

```
SOUND says: 39 dB, read_int = 2499, read_volt = 3.039438
SOUND says: 24 dB, read_int = 3089, read_volt = 3.793773
SOUND says: 23 dB, read_int = 3146, read_volt = 3.823443
SOUND says: 60 dB, read_int = 1593, read_volt = 1.926984
SOUND says: 72 dB, read_int = 1104, read_volt = 1.348718
SOUND says: 72 dB, read_int = 1142, read_volt = 1.407326
SOUND says: 29 dB, read_int = 2916, read_volt = 3.537607
SOUND says: 30 dB, read_int = 2799, read_volt = 3.368010
```

```
SOUND says: 63 dB, read_int = 1485, read_volt = 1.809035
SOUND says: 56 dB, read_int = 1797, read_volt = 2.190110
SOUND says: 38 dB, read_int = 2514, read_volt = 3.071673
SOUND says: 24 dB, read_int = 3107, read_volt = 3.782051
SOUND says: 23 dB, read_int = 3129, read_volt = 3.808425
SOUND says: 60 dB, read_int = 1600, read_volt = 1.938950
SOUND says: 72 dB, read_int = 1119, read_volt = 1.382662
SOUND says: 73 dB, read_int = 1069, read_volt = 1.324054
SOUND says: 25 dB, read_int = 3075, read_volt = 3.753480
SOUND says: 11 dB, read_int = 3636, read_volt = 4.459585
SOUND says: 64 dB, read_int = 1455, read_volt = 1.789499
SOUND says: 56 dB, read_int = 1772, read_volt = 2.168254
```

As you can see the readings decrease, and the sound fluctuations are more notable this time, however the dB value is not very reliable, this is because I don't know where the boundaries of the voltage are for 0 and 100 (max) decibels are, and the sensor is in dB mode, not dBA, so maybe the decibels aren't very reliable, but we can discriminate between louder noise or softer noise in a pretty fair way.

Lets test the last analog device LEGO-Pi is giving support to, the Hitechnic Gyroscope sensor. For this sensor we will run a couple of tests, the first is the one I used to find the default value for the gyroscope, it is the value that the sensor will return when it is stationary. As explained in the appendix B of this document the gyroscope sensor will return always the same value when it is in stationary position, we gonna call this value the default value, when the gyroscope detects a rotation a different value will be returned, if the rotation detected by the sensor is clockwise a value greater than the default value will be returned, otherwise, a value smaller than the default will be returned. Lets then try to figure out what will be our default value with the A/D converter we are using, to do this we will run the test number 5, I will use analog port 2, and we need to set a big iterations option value, the bigger we set it the more reliable the default value found will be. First of all we need to make sure that the sensor is in a stationary position, then

```
$> sudo ./ag_test -t5 -p2 -i10000
```

must output:

```
average reading for 10000 samples: 2335
```

You can easily what this test does, it takes makes as many readings as requested trough the `-i/--times` option and compute the average of all the readings. So with our default value computed we can carry on with test number 4. This test is a very simple test that will, indefinitely, take a reading of the gyroscope sensor every 200 milliseconds and print the results, for the gyroscope sensor we count with the function `ag_gyro_get_val` that will subtract the reading from the A/D converter to the default value found above to the reading of the gyroscope sensor, returning then positive values for a clockwise rotation or negative values for a counter clockwise rotation, we don't have any notion of units here, the bigger the value returned is, in absolute values, the faster we are rotating. Here I've to say that, since I'm interested in testing the library in standalone mode for each of the three big blocks, motors, analog sensors and digital sensors, hence there is no motors involved in this test source file so the rotations will be done manually, just to see that the behavior is the expected, then if we leave the sensor in a stationary position

```
$> sudo ./ag_test -t4 -p2
```

To finish the test you will need to send a signal from keyboard, the output produced will be something like:

```
GYRO says: -5, read_volt = 2.849939, read_int = 2328
```

```
GYRO says: 4, read_volt = 2.850183, read_int = 2332
GYRO says: -16, read_volt = 2.850427, read_int = 2334
GYRO says: -2, read_volt = 2.840904, read_int = 2332
GYRO says: 1, read_volt = 2.837118, read_int = 2332
GYRO says: 4, read_volt = 2.854945, read_int = 2335
GYRO says: 0, read_volt = 2.826618, read_int = 2344
GYRO says: -11, read_volt = 2.841636, read_int = 2331
GYRO says: -17, read_volt = 2.836630, read_int = 2340
GYRO says: -1, read_volt = 2.846398, read_int = 2331
GYRO says: 6, read_volt = 2.824420, read_int = 2324
GYRO says: 3, read_volt = 2.852259, read_int = 2331
GYRO says: 0, read_volt = 2.849573, read_int = 2339
GYRO says: -5, read_volt = 2.849328, read_int = 2340
GYRO says: -2, read_volt = 2.856654, read_int = 2338
GYRO says: -3, read_volt = 2.851770, read_int = 2331
GYRO says: -3, read_volt = 2.849695, read_int = 2329
GYRO says: 1, read_volt = 2.845421, read_int = 2332
GYRO says: 4, read_volt = 2.836996, read_int = 2330
GYRO says: 5, read_volt = 2.859096, read_int = 2328
GYRO says: 6, read_volt = 2.843834, read_int = 2314
GYRO says: 0, read_volt = 2.846886, read_int = 2341
GYRO says: -6, read_volt = 2.854945, read_int = 2322
GYRO says: -4, read_volt = 2.827473, read_int = 2323
^C
```

As you can see there is a little range of values that can be considered negligible readings from the sensor, after some testing I can say that this range goes from -15 to 15 approximately, The default value however can vary a bit when the sensor has been used for a while, it can be re-calibrated with the `ag_gyro_cal` function. Now I will treat to rotate the sensor in both directions to see the results, so this time, as I'm rotating the sensor manually I'll paste here only the interesting parts of the output, these are:

```
        *
        *
        *

GYRO says: 1259, read_volt = 4.403053, read_int = 3594
GYRO says: 635, read_volt = 3.634676, read_int = 2949
GYRO says: 111, read_volt = 2.979731, read_int = 2444

        *
        *
        *
```

Here you have a clockwise rotation, and

```
        *
        *
        *

GYRO says: -1928, read_volt = 0.477534, read_int = 390
GYRO says: -400, read_volt = 2.379243, read_int = 1965

        *
        *
        *
```

Here you have a counter clockwise rotation. So with all our devices tested the only thing that remains is try our library for unknown analog devices. To do this we will use a light sensor as an unknown analog device. The test will loop as many times as demanded with the `-i / --times` option, in each loop it will switch the yellow pin line, it is, it will deliver 3.3V and 0V to the yellow pin alternatively, that in this case will enable the led of the sensor, after this a reading of the port will be performed and printed, after every loop the test will ask to the user if it must

continue or break. The light sensor will be plugged to analog port 1 and, as in the first test, will be towards a white cardboard, so you can compare the results with test 1 if you want, I will use the default settings again for the `-i / --times` option.

```
$> sudo ./ag_test -t6 -p1
```

has produced this output:

```
Delivering yellow wire 0 V
UNKNOWN SAYS: volt = 4.12, int = 3367
Continue?1

Delivering yellow wire 3.3 V
UNKNOWN SAYS: volt = 2.21, int = 1791
Continue?1

Delivering yellow wire 0 V
UNKNOWN SAYS: volt = 4.09, int = 3340
Continue?1

Delivering yellow wire 3.3 V
UNKNOWN SAYS: volt = 2.16, int = 1800
Continue?1

Delivering yellow wire 0 V
UNKNOWN SAYS: volt = 4.10, int = 3358
Continue?1

Delivering yellow wire 3.3 V
UNKNOWN SAYS: volt = 2.18, int = 1787
Continue?1
```

So as you can see the results are the expected, everything must work properly for unknown analog devices too, however the only analog LEGO compliant devices tested are the ones explained above, LEGO push sensor, LEGO light sensor, Hitechnic gyroscope sensor and LEGO sound sensor. With this I finish the analog testing and I'll carry on with digital devices.

4.3 Digital sensors testing

In this section I'm gonna test the digital sensors, the source file containing the tests can be found in the directory `tests` of the provided LEGO-Pi git-hub , it is named `digital_test.c`. To produce the executable file easily once inside the test directory

```
$> make dg_test
```

will produce the executable for you.

This file will provide us 6 test, one for each digital device plus one more designed for unknown devices, to chose the one we want to run we will use the mandatory `-t / --test` option, besides there is four more optional parameters that we can set, these have all their default values, I leave you here the brief usage instructions printed for the test, you will find the default values for the optional parameters at the end of each line enclosed in brackets

```
$> sudo ./dg_test
```

```
Usage: ./dg_test -t<test_num> [-peTd]
  -t --test      test number to perform [1-6]             {mandatory}
  -p --port      digital port [0-1]                       {0}
  -e --tdep      extra paramter, test dependant           {1}
  -T --titled    Print titles before info fields [no arg] {true}
  -d --dbg       Set the library in debug mode [no arg]   {false}
```

The first five tests for digital sensors follows the same pattern, first they ask for the state of the sensor at the beginning of the test, after this they read the sensor info, usually the product version, sensor type, and that kind of information, after this each test will do more specific tasks depending on the sensor it are trying to test. The test will be executed without the debug option, however you can use it if you want, it will make the library print information of the failed transactions of the devices.

Lets begin with the first test, this one is meant to test Hitechnic color sensor, the version of the sensor I have is 1.23, so version 2 of the device is supported but untested. We will use the digital port 0, the test dependant option must evaluate to something different of 0 if we want to calibrate the white color at the end of the test, this is not very interesting, since we won't see any results, and we don't want the test to calibrate the white color while other colors are exposed to the sensor, so we gonna disable it. I will use the titled option in all the tests, since the information printed will be more explanatory. First I will run the test with the color sensor exposed to a blue colored cardboard, later on I will make several executions with different colors to see if it behave as expected. For the readings to work as expected the sensor must be positioned as in the following picture, and this time positioning and ambient light are really critical:

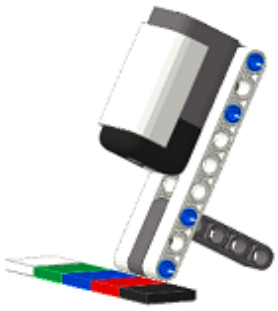


Figure 4.8: Color sensor position

Lets then begin with the blue color,

```
$> sudo ./dg_test -T -t1 -e0
```

has produced the following output:

```
New device successfully created: type >> 4, version >> 1, port >> 0
Initial state: 0
Product Version: V1.23
Manufacturer: HiTechnc
Sensor Type: Color

Col. number: 2
Col. index: 000011, 0x03
Red: 0, Green: 0, Blue: 127
Red_nrm: 0, Green_nrm: 0, Blue_nrm: 255
Red_raw: 1, Green_raw: 9, Blue_raw: 67
Final state: 0
```

As you can see the test inform the user that the device was created successfully, this is because there is some I2C transactions involved in the device creation, after this you can see what will be the default header of a digital sensor test, the initial state and, later, the information fields, this fields will vary for Hitechnic devices and LEGO ultrasonic sensor. After this the specific

test information will be printed. As you can see the color number received is 2, which is fine, if you want you can check out the palette recognized by the sensor in the figure in the section B.2 of the appendix B, concretely in the explanation of the function `dg_col_get_num` or here, the color index stands for a six bit value the 2 more significant bits will be used to indicate the red reading strength, the two following bits will be used to indicate the green reading strength and the least significant bits will indicate the blue reading strength, it is explained in the appendix B of this document in the function `dg_col_get_idx` too. After this you can see the R.G.B readings, the normalized readings and the raw readings for the red, green and blue, which are fine too. After this you can see the final state, it must be for this sensor 0, like the initial state, if no white calibration was performed.

Lets now expose our sensor to a green colored cardboard, the line to run the test will be the same always for this test, the output generated with the green cardboard was:

```
New device successfully created: type >> 4, version >> 1, port >> 0
Initial state: 0
Product Version: V1.23
Manufacturer: HiTechnc
Sensor Type: Color

Col. number: 4
Col. index: 001100, 0x0c
Red: 0, Green: 24, Blue: 0
Red_nrm: 0, Green_nrm: 255, Blue_nrm: 0
Red_raw: 0, Green_raw: 27, Blue_raw: 0
Final state: 0
```

As you can see the color number varies accordingly, and the color index is showing that the green reading is strong, as expected, now lets try it with a red colored cardboard:

```
New device successfully created: type >> 4, version >> 1, port >> 0
Initial state: 0
Product Version: V1.23
Manufacturer: HiTechnc
Sensor Type: Color

Col. number: 9
Col. index: 110000, 0x30
Red: 52, Green: 0, Blue: 0
Red_nrm: 255, Green_nrm: 0, Blue_nrm: 0
Red_raw: 305, Green_raw: 39, Blue_raw: 11
Final state: 0
```

Again color number and color index are showing the correct values, so with our "basic" colors being detected for the sensor lets try with color mixes, first we will try with an orange colored cardboard, the results are the following:

```
New device successfully created: type >> 4, version >> 1, port >> 0
Initial state: 0
Product Version: V1.23
Manufacturer: HiTechnc
Sensor Type: Color

Col. number: 8
Col. index: 110100, 0x34
Red: 54, Green: 23, Blue: 0
Red_nrm: 255, Green_nrm: 109, Blue_nrm: 0
Red_raw: 320, Green_raw: 90, Blue_raw: 11
Final state: 0
```

Pretty fair, as you can see the color number has decreased a unit respect the red cardboard, and the color index shows a little bit of green too, let's keep going with a yellow colored cardboard, this is the output:

```
New device successfully created: type >> 4, version >> 1, port >> 0
Initial state: 0
Product Version: V1.23
Manufacturer: HiTechnc
Sensor Type: Color

Col. number: 5
Col. index: 011100, 0x1c
Red: 58, Green: 95, Blue: 0
Red_nrm: 156, Green_nrm: 255, Blue_nrm: 0
Red_raw: 340, Green_raw: 188, Blue_raw: 50
Final state: 0
```

Again the reading is the correct, color index shows a bit of red and a bunch of green and the color number is the expected, lets try now with a brown cardboard:

```
New device successfully created: type >> 4, version >> 1, port >> 0
Initial state: 0
Product Version: V1.23
Manufacturer: HiTechnc
Sensor Type: Color

Col. number: 9
Col. index: 110000, 0x30
Red: 18, Green: 0, Blue: 0
Red_nrm: 255, Green_nrm: 0, Blue_nrm: 0
Red_raw: 112, Green_raw: 0, Blue_raw: 0
Final state: 0
```

As you can see light brown is detected as a red color, if you take a look at the palette recognized by the sensor there is no number for a brown hue, so the reading is pretty fair. To finalize I'll try with a light purple, almost pink, colored cardboard and with a more darker purple, to see how are this similar colors detected, first the lighter purple cardboard:

```
New device successfully created: type >> 4, version >> 1, port >> 0
Initial state: 0
Product Version: V1.23
Manufacturer: HiTechnc
Sensor Type: Color

Col. number: 9
Col. index: 110000, 0x30
Red: 45, Green: 0, Blue: 0
Red_nrm: 255, Green_nrm: 0, Blue_nrm: 0
Red_raw: 261, Green_raw: 9, Blue_raw: 33
Final state: 0
```

Again this color is detected as red mostly, however if you take a look at the palette is the appropriate reading, I mean that the color exposed is in the range of the number 9, and the red reading varies from the red and brown colored cardboards, so it is fine too. Finally the darker purple cardboard:

```
New device successfully created: type >> 4, version >> 1, port >> 0
Initial state: 0
Product Version: V1.23
Manufacturer: HiTechnc
Sensor Type: Color

Col. number: 9
Col. index: 110000, 0x30
Red: 25, Green: 0, Blue: 0
Red_nrm: 255, Green_nrm: 0, Blue_nrm: 0
Red_raw: 151, Green_raw: 0, Blue_raw: 5
Final state: 0
```

As you can see the color number remains the same, however the red reading is different again so the behavior of the sensor is the expected.

Lets now test the ultrasonic sensor now, in this test we will read all the distances that the sensor provides, what will be the distance to the eight closest objects, in a two different ways, first we will use the function `dg_us_get_dist` and later we'll use the function `dg_us_get_alldist`, the first will ask for the distances one by one, while the latter will request all the distance at once. I've to say here that I can only ensure the first distance, it is, I will put the sensor towards an object at a certain distance making sure that is the first object the sensor will find, the rest of the distances will be assumed correct if this first one is correct, this is because I'm running this test in a small and, believe me, very untidy room and the sensor have a range of 60 degrees and 250 cm, so I can be sure of what object will detect after the first controlled object. To run the test I will use the port 1 this time, the extra parameter will be ignored in this test and the `--titled / -T` option will be enabled as usual. Then with the first object at a distance of about 43cm

```
$> sudo ./dg_test -T -t2 -p1
```

has produced the following output:

```
New device successfully created: type >> 1, version >> 1, port >> 1
Initial state: 2
Product Version: V1.0
Product ID: LEGO
Sensor Type: Sonar
Factory Zero: 0x00
Factory Scale Factor: 0x01
Factory Scale Divisor: 0x0e
Measurement Units: 10E-2m
Measurement Interval: 0x01

Attempting to get all distances one by one
Distance 0: 42
Distance 1: 49
Distance 2: 99
Distance 3: 155
Distance 4: 255
Distance 5: 255
Distance 6: 255
Distance 7: 0

Attempting to get all distances at once
Distance 0: 43
Distance 1: 48
Distance 2: 151
Distance 3: 255
Distance 4: 255
Distance 5: 255
Distance 6: 0
Distance 7: 0

Final state: 2
```

As you can see in this case the info fields are a little bit more extensive, this is the information that the ultrasonic sensor provides, after this the distances taken are fair enough, the sensor have an error of one cm approximately, values of 0 or 255 must be considered error, it means that the sensor wasn't ready in the time of the query or the object was further than the maximum distance the sensor can achieve, however we are usually interested in the first distance so this is the one that matters, now I will put the sensor closer to the object, to a distance of about 28cm, lets see:

```
New device successfully created: type >> 1, version >> 1, port >> 1
```

```
Initial state: 2
Product Version: V1.0
Product ID: LEGO
Sensor Type: Sonar
Factory Zero: 0x00
Factory Scale Factor: 0x01
Factory Scale Divisor: 0x0e
Measurement Units: 10E-2m
Measurement Interval: 0x01

Attempting to get all distances one by one
Distance 0: 29
Distance 1: 32
Distance 2: 48
Distance 3: 65
Distance 4: 81
Distance 5: 255
Distance 6: 255
Distance 7: 255

Attempting to get all distances at once
Distance 0: 29
Distance 1: 32
Distance 2: 48
Distance 3: 65
Distance 4: 81
Distance 5: 255
Distance 6: 255
Distance 7: 255

Final state: 2
```

Well, pretty fine, the distances are fair enough for a mobile robot, as we can see the initial and final state is 2 in both runs, in this sensor the state stands for the mode which the sensor is set, the default 2, is continuous measurement mode, the library change it to be able to get the seven further distances, and once the info is gathered, it set the sensor back to continuous measurement mode, so the state is fine too.

I will continue now with the Hitechnic infrared seeker sensor. The version I have of this sensor is 1.2 so version two, as happens with the color sensor, is supported but untested. Besides I've to say that I don't have a proper infrared emitter, so the tests are a little bit poor this time. I'm using as a emitter a remote controller for a TV, this devices use to function via signals, it is, the infrared signal won't be stable, it will raise and fall to send ones or zeroes to the device in question, in this case my TV, and I want to keep being able of controlling my TV from my coach at any price, so I decided not to hack the remote controller. However to make an idea of how the infrared seeker will work we can make a test and see if we catch a one emission from the RC. To perform the test the digital port 0 will be used, the `-T / --titled` option will be set as usual and the extra parameter must evaluate to true, it is, something different of 0, if we want the sensor to work in DC mode, otherwise it'll work in AC mode. Here you may think, why you was thinking of hacking the RC, set the sensor in AC mode and try, but, unfortunately version 1 of the sensor does not support AC signal detection, so the extra parameter must be true here for me. Lets run the test and see if we are able to catch something. I'm pointing the RC controller to the direction 2 of the sensor, if you want you cant check the explanation of the function, `dg_irs_get_dir` in the appendix B of this document where you will find a graphic of the direction numbers, you can also reach the graphic here.

```
$> sudo ./dg_test -t3 -e1 -p0 -T
```

has produced this output:

```
New device successfully created: type >> 5, version >> 1, port >> 0

Initial state: 0
Product Version: V1.2
Manufacturer: HiTechnc
Sensor Type: IR Dir.

DC direction = 2

Getting stregths one by one
DC stregth 1 = 1
DC stregth 2 = 0
DC stregth 3 = 0
DC stregth 4 = 0
DC stregth 5 = 0

Getting all stregths at once
DC stregth 1 = 25
DC stregth 2 = 12
DC stregth 3 = 0
DC stregth 4 = 0
DC stregth 5 = 0

Attempting to get DC average on version 1...

send_message: Unsupported sensor version.
dg_irs_get_dcavg: DC Average reading failed.
Error getting DC average, as expected.
Final state: 0
```

As you can see, the test behave like the latter test, first it takes the distances one by one, and later, it takes the distances all at once. The direction number as you can see has been successfully detected however when asking for the distances one by one no signal strength was detected, this is because of the RC signaling. Nevertheless when all the distances was requested at once we've been lucky an we were able to see the strength in the direction registers 1 and 2, which holds the strength signal for the directions of the sensor 1 and 3, what is the expected behavior of the sensor, when even directions are returned from the sensor it must be interpreted as if a part of the signal is being detected in the two adjacent odd numbered directions. After the readings are performed the sensor attempts to get the DC average, which is a functionality available only to the version 2 of the sensor, hence it fails. The state will hold the mode the sensor is set to, as we use the version one of the sensor only DC mode is available. As a last test I will try to point the RC to the direction number five to see if we can get the readings, the results are, this time:

```
New device successfully created: type >> 5, version >> 1, port >> 0

Initial state: 0
Product Version: V1.2
Manufacturer: HiTechnc
Sensor Type: IR Dir.

DC direction = 5

Getting stregths one by one
DC stregth 1 = 0
DC stregth 2 = 5
DC stregth 3 = 77
DC stregth 4 = 0
DC stregth 5 = 0

Getting all stregths at once
DC stregth 1 = 0
DC stregth 2 = 1
```

```
DC stregth 3 = 0
DC stregth 4 = 0
DC stregth 5 = 0

Attempting to get DC average on version 1...

send_message: Unsupported sensor version.
dg_irs_get_dcavg: DC Average reading failed.
Error getting DC average, as expected.
Final state: 0
```

Pretty fair, even with the RC controller which is not an ideal emitter, as we can see we didn't have fortune when attempting to get the strengths all at once, however, we can say that with a proper infrared emitter the sensor must work properly.

Lets now test the Hitechnic accelerometer sensor, as mentioned in the analog testing section I'm interested for now in testing the library in standalone mode, then we don't have motors management in this test, so we can't move the sensor in a stable manner to see how it behave, however the Hitechnic accelerometer can be used as a tilt detector too, it have its drawbacks, if the robot encounters a pothole ore a bump in the way the readings will be affected, however this will come in handy for this test since we will put the sensor in different angles and make readings to see the axis G forces. To do this test number 4 will be used, the test dependant option will be ignored this time and we will plug the sensor to port 0, then with the sensor in a horizontal position:

```
$> sudo ./dg_test -t4 -p0 -T
```

has produced the following output:

```
New device successfully created: type >> 3, version >> 1, port >> 0
Initial state: 0
Product Version: V1.1
Manufacturer: HITECHNC
Sensor Type: Accel.

Attempting to get axis:
X: -10
Y: -7
Z: 194

Final state: 0
```

The state is useless here, you can see the info fields and the readings of the G forces on the axis the sensor will report a 200 units reading for each G unit on the axis, so for a horizontal position the reading is fair enough, lets now tilt the sensor till we get an angle of about 45 degrees with the vertical and horizontal axis with the head of the sensor pointing up, to see how the gravity distributes around the three axis, the results are:

```
New device successfully created: type >> 3, version >> 1, port >> 0
Initial state: 0
Product Version: V1.1
Manufacturer: HITECHNC
Sensor Type: Accel.

Attempting to get axis:
X: 124
Y: -3
Z: 150

Final state: 0
```

As you can see the X axis is detecting a greater G force now while Z axis is detecting less G force, as expected for the position we set the sensor to. Finally we will set the sensor totally vertical with the head pointing down, we must see a negative X value near 200 and almost no G force in the Z axis, the results are:

```
New device successfully created: type >> 3, version >> 1, port >> 0
Initial state: 0
Product Version: V1.1
Manufacturer: HITECHNC
Sensor Type: Accel.

Attempting to get axis:
X: -200
Y: 1
Z: 2

Final state: 0
```

Almost perfect, here we can see how all the gravity falls into the X axis, so everything is going pretty well. The Y axis will react to linear movements, it is, when we move the robot forward or backwards.

Lets test our last known digital device the Hitechnic magnetic compass. This digital sensor will provide us with two heading readings ranging from 0 (north) to 359, which ideally must coincide. So 180 will be south, 90 east and 270 west. We gonna use test 5, the test dependant option is used in this test to calibrate the sensor, so we gonna disable it and the sensor will be plugged to port 0, then with the device facing north

```
$> sudo ./dg_test -t5 -p0 -e0 -T
```

produced the following option

```
New device successfully created: type >> 2, version >> 1, port >> 0
Initial state: 0
Product Version: V1.23
Manufacturer: HiTechnc
Sensor Type: Compass

Attempting to get heading:
2DG_head: 0
WDG_head: 0

Final state: 0
```

Now, if we turn the sensor about 90 degrees to the east:

```
New device successfully created: type >> 2, version >> 1, port >> 0
Initial state: 0
Product Version: V1.23
Manufacturer: HiTechnc
Sensor Type: Compass

Attempting to get heading:
2DG_head: 92
WDG_head: 93

Final state: 0
```

Another 90 degrees more must take us to south, it is a 180 reading:

```
New device successfully created: type >> 2, version >> 1, port >> 0
Initial state: 0
Product Version: V1.23
Manufacturer: HiTechnc
Sensor Type: Compass
```

```
Attempting to get heading:
2DG_head: 180
WDG_head: 180

Final state: 0
```

Finally we must find west, a 270 heading value, rotating the device 90 degrees more.

```
New device successfully created: type >> 2, version >> 1, port >> 0
Initial state: 0
Product Version: V1.23
Manufacturer: HiTechnc
Sensor Type: Compass

Attempting to get heading:
2DG_head: 271
WDG_head: 271

Final state: 0
```

So everything is going pretty well at the end.

I will make a final test now to see if unknown devices are well handled by the library, for this test you will need a LEGO ultrasonic sensor attached to the port requested trough the parameter `-p/--port` and the tests dependant option will indicate the port where a Hitechnic color sensor will be plugged to. Both devices will be declared as unknown devices, once the devices are created a reading of the sensor type for the ultrasonic device and the color number for the Hitechnic color sensor will be performed, the `-T / --titled` option will be ignored for this test, then with a LEGO ultrasonic sensor plugged to port 1 and a color sensor plugged to port 0 and exposed to the orange colored cardboard:

```
$> sudo ./dg_test -t6 -p1 -e0
```

has produced this output:

```
Device plugged to port 1 successfully created
Device plugged to port 0 successfully created
Attempting to get sensor type on port 1 [gessing LEGO_US ...]
Sensor Type: Sonar
Attempting to get color number on port 0 [gessing HT_COLOR ...]
Color number: 8
```

So unknown devices must be well handled too.

With this I finalize the digital sensor testing, keep in mind that version two of the sensors that count with it, it is, Hitechnic IR seeker and color sensor aren't tested, however the drivers has been implemented too in the hope that it will work fine.

4.4 Conclusions

Besides that the tests done in the latter section are pretty explanatory, and, I think, that the conclusions are pretty clear I will try to summarize a bit in this section to see the strengths and weakness of the project.

To me the election of a Raspberry Pi to act as a PBX is fine, one of the best points is that there is a big community growing day by day involved and developing very interesting software, and it is always good to have people that may be interested in the project, actually feedback will be very appreciated.

4.4. Conclusions

Respect to the technical part, as mentioned in sections above the project may be very minimalist, however it aims to capture the interest of the community, and hopefully, be the target of some improvements some day. The motor management is pretty fair to control velocity and distances traveled, however, as mentioned in the above sections the project is lacking some hardware for the control of the motors can be considered "professional". The pulses generated for the LEGO-Pi library have a frequency of about 334 Hz, and DC motors use to work at a frequency of, at least a few KHz, however to make a simple robot and reuse old LEGO peripherals I think that it is a good approach, or at least a good starting point.

Respect to the sensors management I think that it are very well handled by our Raspberry Pi, the communications with the A/D converter via the SPI interface are very good, and digital sensors are pretty well handled too, even with the "homemade" I2C library that takes care of the communications, however some versions of the sensors remains untested, and the drivers has been developed blindly, so, again, feedback would be very appreciated here.

To me, the main strength of this project is the library developed to interact with the peripherals. Having a native C ANSI library to control LEGO peripherals is, under my point of view, a very desirable capability to anyone interested on LEGO peripherals management, since the library will take care of getting the data of the sensors, and the user still have the power of C ANSI, and all the libraries suited for it, to treat this data as he likes. We have to thank our Raspberry Pi for it, since it boots into a proper programming environment, what allowed me to develop this library. In fact LEGO peripherals just are a particular case of motor and sensors, so, probably, the library can be rearranged to interact with other analog sensors, I2C compliant digital devices and, hopefully, motors, in other words, robotics with a Raspberry Pi acting as a "brain" seems a pretty achievable goal. In top of that all the LEGO peripherals control is done via the GPIO pins of the board, so the CSI buses to plug cameras, the USB ports to plug wifi or bluetooth dongles, and the ethernet port remains free and available for the user benefit, and with an operating system on running in the board the communication with such devices is an straightforward and standardized task.

As a final note, I've to say that in the meantime of the development of this project a new board was released, I'm referring to ODRIOD-U3, which will provide a lot more power to the PBX and a more handy hardware interface, with its IO shield, however, as mentioned in the section 3.4, with a simple PIC microcontroller the motor management will be solved pretty well and we will still have the Raspberry Pi community, which, to me, is one of the strengths of the project.

Chapter 5

Miscellaneous

5.1 Budget

As mentioned in the section 1.3, at the very beginning of this document, one of the strengths of LEGO-Pi project is its simple hardware, hence it can be achieved with a very reduced budget and implemented for whoever interested, so here I leave a summary of all the hardware used to implement our project and its prices, to finally get a fair price of all the hardware, all the LEGO peripherals and shipment prices won't be accounted here.

- Raspberry Pi model B revision 2: 35$
- H-bridge sn754410: 2.95$
- A/D converter MCP3204: 3.95$
- Voltage regulator L2595T: 2.84$
- NXT-Breadboard adapter (optional): 3.25$ (x8)
- All the rest of the electronic components can be estimated in about: 5$

Then the total price of the project is, since LEGO-Pi have 8 NXT shaped ports, $35 + 2.95 + 3.95 + 2.84 + 3.25 \times 8 + 5 = 75.74$ $ Raspberry Pi included, plus shipments if you need it. Not bad.

5.2 Bibliography

As you can see a lot of sites have been referenced in this document, however there is some of them that doesn't appear in the document and have been consulted for the implementation of the project, either to inspire some of the designs of the project, to get information about the peripherals or to get code I reused for LEGO-Pi project, here you have what I think is a fairly complete list of all the sites, books and documents consulted:

- Extreme NXT book
- Dexter industries

- RPIO project by Chris Hager
- WiringPi project by Gordon Henderson
- Philippe "Philo" Hurbain site
- alanbarr git-hub
- ROBOTC project, you can find the ROBOTC drivers sources here
- Embedded Linux Wiki
- Raspberry Pi data sheet
- I2C bus protocol specification from Philips
- LEGO Hardware Developer Kit
- LEGO sound sensor survey

And, of course, Wikipedia and Google have been severely used at every step of this project. I think I'm not forgetting about anyone, however if you think there is somebody missing here, please make me know.

5.3 Acknowledgments

Here, there is a lot of people to mention, however I would like to give special thanks to all the people responsible for the documents, sites or projects mentioned in the last section (5.2) since the project wouldn't be possible without all the information they provide.

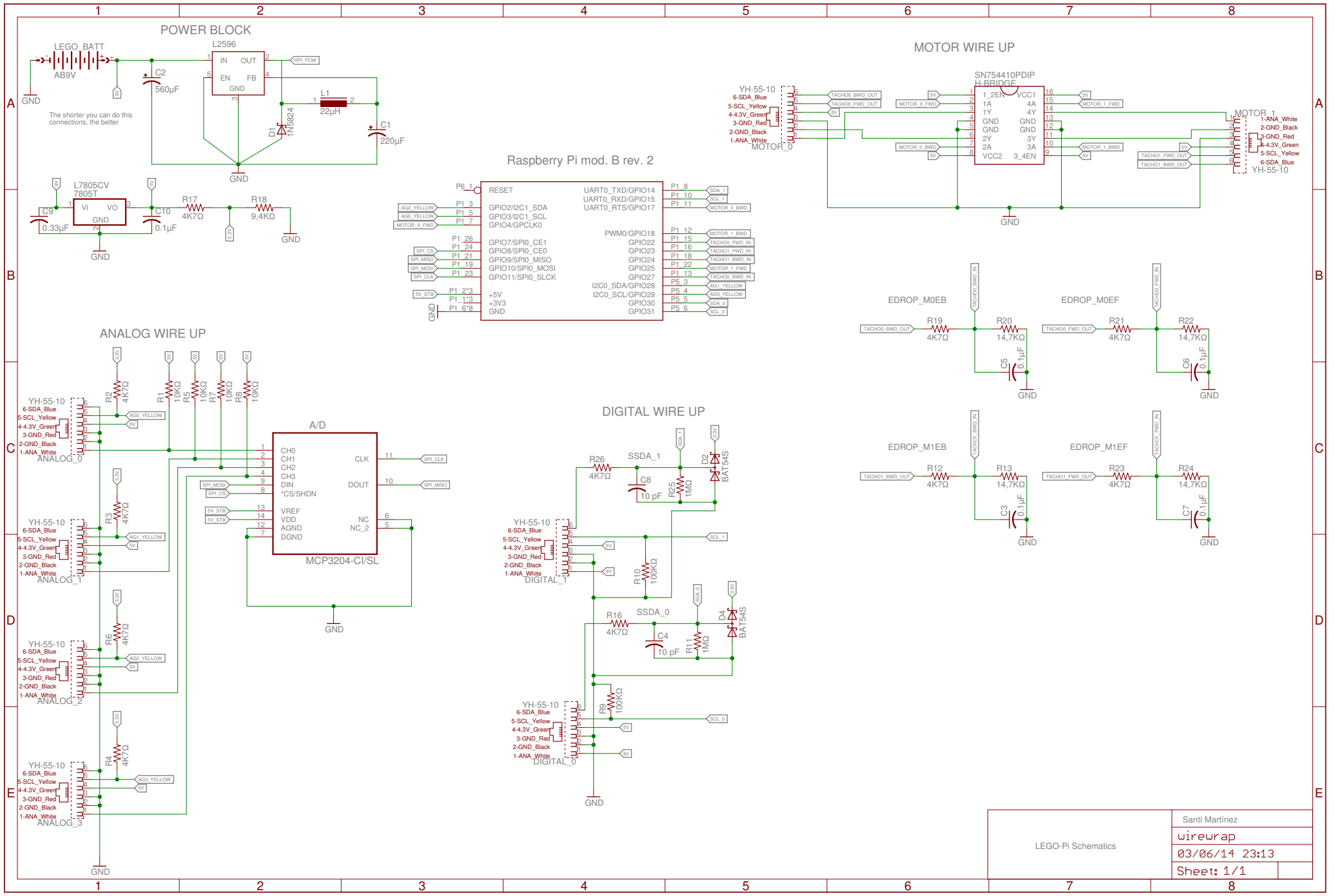
And, if you arrived till this point of the document, I owe you a very very big THANK YOU!

Appendix A

Project Schematics

These are the LEGO-Pi hardware schematics, in the `docs/schematics/eagle` directory of the LEGO-Pi git-hub you will find the schematics files for eagle CAD soft, this is a free software PCB design tool, it can be found here, a pdf file of the schematic is available too, the libraries used to make the schematics can be found at:

- A/D converter, you will need an account in element14 to get this one.
- H-Bridge, thanks to Alex Whittemore
- RaspberryPi model B revision 2, thanks to stv0g.


POWER BLOCK
LEGO_BATT
AB9V
GND
The shorter you can do this connections, the better
L2596
IN
OUT
EN
FB
GND
RPI_POW
C2
560µF
L1
22µH
D1
1N5824
C1
220µF
L7805CV
7805T
VI
VO
C9
0.33µF
C10
0.1µF
R17
4K7Ω
R18
9,4KΩ
Raspberry Pi mod. B rev. 2
P6_1
RESET
GPIO2/I2C1_SDA
GPIO3/I2C1_SCL
GPIO4/GPCLK0
GPIO7/SPI0_CE1
GPIO8/SPI0_CE0
GPIO9/SPI0_MISO
GPIO10/SPI0_MOSI
GPIO11/SPI0_SLCK
+5V
+3V3
GND
UART0_TXD/GPIO14
UART0_RXD/GPIO15
UART0_RTS/GPIO17
PWM0/GPIO18
GPIO22
GPIO23
GPIO24
GPIO25
GPIO27
I2C0_SDA/GPIO28
I2C0_SCL/GPIO29
GPIO30
GPIO31
MOTOR WIRE UP
SN754410PDIP
H-BRIDGE
1_2EN
1A
1Y
GND
2Y
2A
VCC2
VCC1
4A
4Y
3Y
3A
3_4EN
YH-55-10
6-SDA_Blue
5-SCL_Yellow
4-4.3V_Green
3-GND_Red
2-GND_Black
1-ANA_White
MOTOR_0
MOTOR_1
ANALOG WIRE UP
ANALOG_0
ANALOG_1
ANALOG_2
ANALOG_3
A/D
MCP3204-CI/SL
CH0
CH1
CH2
CH3
DIN
*CS/SHDN
VREF
VDD
AGND
DGND
CLK
DOUT
NC
NC_2
DIGITAL WIRE UP
DIGITAL_1
DIGITAL_0
SSDA_1
SSDA_0
BAT54S
EDROP_M0EB
EDROP_M0EF
EDROP_M1EB
EDROP_M1EF
LEGO-Pi Schematics
Santi Martínez
wirewrap
03/06/14 23:13
Sheet: 1/1

Appendix B

Library Specification

This appendix is dedicated to the C ANSI software library specification of the LEGO-Pi project, here you will find a more detailed explanation of the functionalities that the library provides to the user, the code of the library won't be added to this section, however you can always reach it at LEGO-Pi git-hub . LEGO-Pi uses the GPIO numbering scheme, so every time I refer to a GPIO number I'll be meaning the number of the GPIO, not the number of the pin on the board. The LEGO-Pi library makes use of the WiringPi and the gls (GNU Scientific Library) libraries.

B.1 lego_motor.c

In this section the functionalities and the structures provided by the file `lego_motor.c` will be explained.

B.1.1 Defaults

For the motor interface, according to the official LEGO-Pi assembly you can find in the section 3.1, the default pins will be, for the motor port 0:

```
enum mot1 {M1_PINF = 4, M1_PINR = 17, M1_ENC1 = 27, M1_ENC2 = 22, M1_CHANN = 0};
```

And for the motor port 1:

```
enum mot2 {M2_PINF = 25, M2_PINR = 18, M2_ENC1 = 24, M2_ENC2 = 23, M2_CHANN = 7};
```

This values stands for:

- `MX_PINF`: The pin responsible of the forward motion on the port `X-1`
- `MX_PINR`: The pin responsible of the backward motion on the port `X-1`
- `MX_ENC1`: The pin responsible of the encoder first line on the motor port `X-1`
- `MX_ENC2`: The pin responsible of the encoder second line on the motor port `X-1`
- `MX_CHANN`: The DMA channel used by the motor port `X-1`

If you want to rearrange the hardware this are the values you may want to modify.

B.1.2 Structures and types

Type `dir`

```
typedef enum {
  FWD,
  BWD
} dir;
```

Enumeration defining the directions that we can set on a motor, `FWD` stands for forward motion while `BWD` stands for backward motion. To LEGO-Pi, with the connections done as showed in the project schematics (Appendix A of this document), when we set a motor in forward motion the motor will turn as in the following picture:

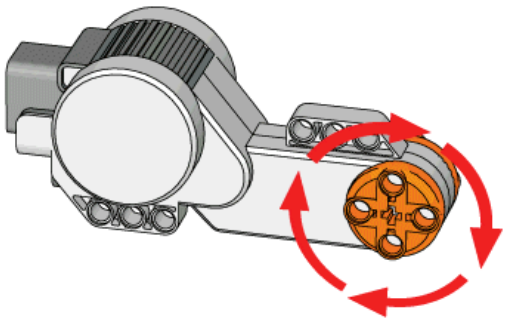


Figure B.1: LEGO-NXT motor forward motion

Struct `MOTOR`

```
struct motor {
  int id;
  int pinf;
  int pinr;
  int chann;
  bool moving;
  int ticsxturn;
  ENC * enc1;
  ENC * enc2;
  PID * pid;
};

typedef struct motor MOTOR;
```

This is the logical representation of a motor. The `id` field is used internally for the library, and is not meant to be modified by the user. The `pinf` and `pinr` fields are the GPIOs responsible for the motor motion, being the `pinf` field the one that holds the GPIO number of the pin responsible for the forward motion, and the `pinr` the one holding the GPIO number of the pin responsible for the backward motion, this fields are set by the library too. The field `chann` stands for the DMA channel used by the motor in question, this field is, again, set by the library, and the user must not modify it. The `moving` field will be true when the motor is in motion, while will evaluate to false when the motor is stationary, it is set by the library. The field `ticsxturn` is the amount of tics we will receive for each turn of the output hub, it will vary depending on the number of encoders we use, either 360 for one encoder or 720 for two encoders. The fields `enc1` and `enc2` are pointers

to encoder structures that will hold the information of each of the encoder lines of the motor, note here that LEGO-Pi uses interchangeably the two signals, without worrying about the direction, if you want a more precise explanation about this you can refer to the section 2.2.2 of this document. The `pid` field is a pointer to a `PID` structure that will hold all the data needed to P.I.D control, this functionality as mentioned in the section 2.2.2 is considered experimental.

Struct `ENC`

```
typedef struct timespec TSPEC;

typedef pthread_mutex_t MUTEX;

struct encoder {
  int pin;
  int tics;
  TSPEC tmp;
  void (*isr)(void);
  MUTEX mtx;
};

typedef struct encoder ENC;
```

This is our representation of an encoder line. The field `pin` holds the GPIO number of the pin connected to the encoder output, it is set by the library, and mustn't be altered. The field `tics` will hold the amount of ticks received in this encoder line since the last reset. The field `tmp` is an struct of type timespec, it is used for the default ISRs to store a time stamp that will be overwritten every time a tick comes. the field `*isr` is a pointer to a void function, this will be the code executed every time a tick comes in from this line, there is four default ISRs (Interrupt Service Routine) in the library that will update the ticks field of the encoder they are in charge of, however the function can be redefined by the user, note that the communication with it must be done via global variables. The field `mtx` is a mutex that will be locked before any tick update in the default ISRs, and unlocked after the update has taken place.

Struct `PID`

```
#define MAX_COEF        21

struct pid {
  bool active;
  int svel;
  double cp[MAX_COEF];
  double cd[MAX_COEF];
  double kp;
  double ki;
  double kd;
  double ttc;
  gsl_interp_accel *accelM;
  gsl_interp_accel *accelD;

};

typedef struct pid PID;
```

This is the structure that will hold all the info to implement the P.I.D control on a motor. The first field `active` is used to activate or deactivate the P.I.D control, as mentioned

previously P.I.D is an experimental feature, so it defaults to false. The field `svel` is used internally for the library to store the "velocity step", it is the difference between adjacent velocities for which we have the error and expected time between ticks values, if you want more info on how P.I.D works in the LEGO-Pi library you can refer to the section 2.2.2 of this document. The fields `cp` and `cd` are arrays of floating point values that will hold the time between ticks (`cp`) and the physical error (`cd`) values for the amount of velocities we hold data, the number of velocities for what we can store data is limited to 20, and the arrays will always be ended with a "0", this fields can be recomputed with the function `mt_calibrate`. The fields `kp`, `ki` and `kd` are the proportional gain, integral gain and derivative gain for the P.I.D control. The field `ttc` stands for "ticks to configure" and it is the amount of ticks that the P.I.D control will wait to check if an error happened. The two last fields `accelM` and `accelD` are pointers to a `gsl_interp_accel`, this fields are used internally by the library to speed up the interpolations on our data set, it is, in the values held in the `cp` and `cd` fields.

B.1.3 Functions

In this section will be explained the functions that this file will provide to interact with the motors.

mt_init

```
extern bool mt_init ();
```

This function doesn't take any parameters, it must be called once before calling any of the other functions provided for this file, it will initialize the motor interface. If the function succeed true will be returned, false otherwise.

mt_set_verbose

```
#define LOG_LVL_DBG   2
#define LOG_LVL_ADV   1
#define LOG_LVL_FATAL 0

extern bool mt_set_verbose(int lvl);
```

This function takes as a parameter an integer value `lvl`. It will set the log level for the motor library, the `lvl` must range from 0 to 2, being 2 the most verbose and 0 the less verbose. If succeed the function will return true, false otherwise.

mt_new

```
extern MOTOR * mt_new (ENC * e1, ENC * e2, int port);
```

This function takes as a parameters two pointers to a `ENC` struct `e1`, `e2` and an integer `port` representing the port where we want to load the configuration that the structures pointed by the `e1` and `e2` parameters holds. The way to specify a configuration is a little bit tricky however. If any of the pointers `eX` is NULL the default configuration will be loaded into the encoder line `X` of the `port` passed as a parameter, otherwise, if the `pin` field of the structure pointed for the parameters `eX`, where `X` is the number of the encoder to configure, holds

the pin number matching the MP_ENCX from the motor defaults (B.1.1), where P is the port+1, the function pointed by the eX->isr field will be loaded into the X encoder line of the motor port port, if the pin field of the structure pointed by the eX parameter doesn't match the default pin number the encoder will be disabled. As a result the function will return a pointer to a MOTOR struct pointing to a structure holding the configuration specified as explained for the motor port port. To easily disable encoders a macro ECNULL is provided in the library, if you set one of the eX parameters to ECNULL the encoder line in question will be disabled, this macro is a pointer to an ENC struct with the pin field equal to -1. As you may intuit this function will initialize a motor port, so it must be called once before we use a motor port, to allocate all the structures needed by the motor.[1] If the function succeed the pointer returned will point to a valid motor structure, otherwise NULL will be returned.

mt_reconf

```
extern bool mt_reconf (MOTOR * m, ENC * e1, ENC * e2);
```

If you previously configured a motor with the function mt_new and you want to load a different configuration at some step of your program this is the function you may look for. This function takes as a parameters a pointer to a motor struct m, typically returned by the mt_new function, and apply the configuration hold by the structures pointed by e1 and e2. The way to specify a configuration is, again a little bit tricky. If both eX pointers are NULL the default configuration will be loaded into the motor port pointed by m, if only one of the eX parameters is NULL the other encoder line configuration will be loaded and the first will remain untouched, otherwise, if both eX pointers are different to NULL both encoder lines of the motor port pointed by m will be modified. mt_reconf loads a configuration as mt_new does, again you can use the above mentioned macro ECNULL to easily disable encoders.[1] If the function succeed true will be returned, false otherwise.

mt_enc_is_null

```
extern int mt_enc_is_null (MOTOR * m, int eid);
```

This function takes as a parameters a pointer to a MOTOR structure m and integer eid representing the encoder line id (either 1 or 2) for what we want to check the availability with the configuration in use. The function will return a negative integer if an error occurs, otherwise, will return either true (1) if the encoder line eid is active for the motor port pointed by m or false (0) if the encoder line eid is disabled for the motor port pointed by m.

[1] *WARNING:* The library assumes that any function provided to act as an ISR will, at least, update the tics field of the encoder line it is responsible of, if you refuse to update this field misbehavior will occur. A possible workaround for this can be, if 2 encoder lines are active and you don't want to update one of the tics field you can manually set the value of the ticsxturn field to 360, and everything must work properly, however, if you want to use just one encoder line in one port, it is, one of the encoders will be disabled, the enabled encoder must necessarily update the tics field of the line it is in charge.

mt_enc_count

```
extern int mt_enc_count (MOTOR * m);
```

This function takes as a parameter a pointer to a struct `MOTOR m`, it will return a negative value if an error occurred, otherwise, it will return the total number of encoders enabled for the motor port pointed by `m`.

mt_reset_enc

```
extern bool mt_reset_enc (MOTOR * m);
```

This function takes as a parameter a pointer to a struct `MOTOR m`, it will return false if an error occurred, otherwise, it will reset the encoder `tics` field of each encoder line of the motor port pointed by `m`, it is, it will set the fields to 0, and return true.

mt_get_ticks

```
extern int mt_get_ticks (MOTOR * m);
```

This function takes as a parameter a pointer to a `MOTOR` struct `m`. It will return a negative value if an error occurred, otherwise, it will return the amount of tics accumulated in the motor port pointed by `m`. If only one encoder line is enabled in the motor port pointed by `m` the amount of ticks stored in this line will be returned, else, if the two lines are enabled, the sum of the tics stored by the two lines of the motor port pointed by `m` will be returned. Note here that the case of no encoder lines enabled in the motor port pointed by `m` is treated as an error, so a negative value will be returned.

mt_get_time

```
extern TSPEC * mt_get_time (MOTOR * m, int eid);
```

This function takes as a parameters a pointer to a `MOTOR` structure `m` and an integer `eid` representing an encoder id, either 1 or 2. If an error occurs it will return NULL, otherwise, it will return the time stamp held in the encoder line `eid` of the motor port pointed by `m`. If the encoder line passed in the `eid` field is disabled an error will arise, hence NULL will be returned.

mt_lock

```
extern bool mt_lock (MOTOR * m);
```

This function takes as a parameter a pointer to a `MOTOR` struct `m`. If an error occurs false will be returned, otherwise the two mutex objects in the encoder lines pointed by `m` will be locked and an true will be returned. This function is useful to stop counting ticks for a while when the default ISRs are in use, otherwise the user can use it for his benefit.[2]

[2] If you want more info on pthread mutexes you can refer to this site

mt_unlock

```
extern bool mt_unlock (MOTOR * m);
```

This function takes as a parameter a pointer to a MOTOR struct m. If an error occurs false will be returned, otherwise the two mutex objects in the encoder lines pointed by m will be unlocked and an true will be returned.[2]

mt_calibrate

```
extern bool mt_calibrate (int samples, double wbs);
```

This function takes as a parameters an integer samples and a floating point value wbs. This function is meant to be used before we use P.I.D control, to get the values of the cp and cd arrays held in the PID structure of each motor initialized when this function is called. It is implemented this way because the values vary a lot when two motors are moving than when there is only one motor in motion, you can refer to the section 2.2.2 for a more detailed explanation. The integer samples stands for the number of velocities we will get info of, it is the number of velocities for which we will compute the time between ticks and the physical error values that we will store in the cp and cd arrays respectively. Note that we will interpolate this arrays later on, when P.I.D control is demanded by the user, so the more velocities we compute the more precise will be the interpolation. The velocities shift will be the same for adjacent velocities, so the points we will interpolate are the best distributed. The integer samples must range from 5 to 20. The floating point value wbs stands for wait between samples, it is exposed for the user to control the time (in seconds) we will wait between every sample taken. If no error occurs the function will return true, otherwise false will be returned.

mt_pid_conf

```
extern bool mt_pid_conf (MOTOR * m , double micros [], double desv []);
```

This function take as a parameters two pointers to a floating point value arrays micros and desv, and one more pointer to a MOTOR struct m. The latter function mt_calibrate is an expensive function since we need to do a lot of computations, hence this function is provided to load a pre-computed values for the cp and cd arrays of the PID structure held by the motor port pointed by m. The values stored in the array pointed by micros will be loaded in the cp array of the PID structure held by the motor port pointed by m, while the values stored in the array pointed by desv will be loaded in the cd array of the PID structure held in the motor pointed by m. If everything went right true will be returned, false otherwise.

mt_pid_is_null

```
extern int mt_pid_is_null (MOTOR * m);
```

This function takes as a parameter a pointer to a MOTOR struct m. The function will return a negative value if an error occurs, else, it will return true (1) if the active field of the PID structure held by the motor pointed by m evaluates to false, it is, if the P.I.D control isn't active for the motor pointed by m, otherwise, the function will return false (0).

mt_pid_off

```
extern bool mt_pid_off (MOTOR * m);
```

This function takes as a parameter a pointer to a MOTOR struct m. It will set the active field of the PID structure held by the motor pointed by m to false, it is it will disable the P.I.D control for the motor pointed by m. If the function succeed true will be returned, false otherwise.

mt_pid_on

```
extern bool mt_pid_on (MOTOR * m);
```

This function takes as a parameter a pointer to a MOTOR struct m. It will set the active field of the PID structure held by the motor pointed by m to true, it is, it will enable the P.I.D control for the motor pointed by m. If the function succeed true will be returned, false otherwise.

mt_pid_set_gains

```
extern bool mt_pid_set_gains (MOTOR * m, double Kp, double Ki, double Kd);
```

This function takes as a parameter a pointer to a MOTOR struct m and three floating point values Kp, Ki and Kd. It will set the fields kp, ki and kd of the PID structure held by the motor pointed by m, where Kp is the proportional gain, Ki is the integral gain and Kd is the derivative gain for P.I.D control. If the function succeed true will be returned, false otherwise. Note that this must be the only way to interact with the P.I.D control, however, due to LEGO-Pis implementation of the P.I.D algorithm there is another field that will affect the behavior of the P.I.D control, it is the ttc field of the PID structure of each motor, there is no function to modify it so the user must set it manually. To a proper explanation about the ttc field you can check the section B.1.2 of this appendix, if you want a more detailed explanation about the P.I.D algorithm LEGO-Pis implements you can check out the section 2.2.2 of this document.

mt_move

```
extern bool mt_move (MOTOR * m, dir dr, int vel);
```

This function takes as a parameters a pointer to a MOTOR struct m, a direction dr, and an integer value vel. This function will make the motor pointed by m turn in the direction specified through the dr parameter and at a vel velocity.[3] If P.I.D control is active for the motor pointed by m P.I.D control will be launched, however the program flow will be returned to the user program. The motor won't stop until it is required. If the function succeed, true will be returned, false otherwise.

[3] The velocity is represented with an integer which value must range between 0 and 200

mt_move_t

```
extern bool mt_move_t (MOTOR * m, int ticks, dir dr, int vel, double posCtrl);
```

This function takes as a parameters a pointer to a `MOTOR` structure `m`, an integer `ticks`, a direction `dr`, an integer `vel` and a floating point value `posCtrl`. It will make the motor pointed by `m` move in the direction `dr` at a `vel` velocity[3] and till we receive `ticks` ticks from the motor. The last parameter `posCtrl` is a floating point value indicating if we want position control in this travel, the maximum value it must have is 1 representing that we want position control all along the travel. If `posCtrl` is 0 no position control will be done. If `posCtrl` is a value between 0 and 1 the velocity of the motor will start decreasing when the motor has reached the `(1-posCtrl)*100%` of the travel, it is when the motor has received `(1-posCtrl)*ticks` ticks, the smaller `posCtrl` is the more drastic will be the velocity decrement. It is implemented to help to reduce the motor error when a set point is defined. Besides if P.I.D control is active for the motor pointed by `m` P.I.D control will be launched at the same time. If the function succeed true will be returned, otherwise false will be returned. The control flow will be returned to the user code once the motor has been set.

mt_tticks

```
extern int mt_tticks (MOTOR * m, int turns);
```

This function takes as a parameter a pointer to a `MOTOR` structure `m` and an integer value `turns`. It will return a negative value if an error occurs or 0 if no encoder lines are enabled in the motor pointed by `m`, else it will return an integer representing the amount of ticks expected for `turns` turns of the output hub with the configuration that the motor pointed by `m` holds. What the function will do at the end is return `turns*m->ticsxturn`.

mt_move_sinc

```
extern bool mt_move_sinc (dir dr, int vel);
```

This function takes as a parameter a direction `dr` and an integer `vel`. For this function two motors are needed, it will attempt to move both motors synchronously in the direction `dr` and at a `vel` velocity,[3] it is, it will increment or decrement the velocity of the motors depending on the difference between the ticks received by both motors, trying to make this difference the smallest possible. Besides if P.I.D control is active it will be launched, however, it will stop acting when one of the motors correct the initial acceleration error, from that point, velocity will be only modified to achieve synchrony between motors. Once the motors has been set the control flow will be returned to the user program and the motors won't stop until the user requires it. If the function succeed true will be returned, false otherwise.

mt_move_sinc_t

```
extern bool mt_move_sinc_t (dir dr, int vel, int lim, double posCtrl);
```

This function takes as a parameters a direction `dr`, an integer `vel`, another integer `lim` and a floating point value `posCtrl`. This function will behave as the last function commented

mt_move_sinc but with a set point specified through the lim parameter, it is, it will attempt to move the motors synchronously in a direction dir and at a velocity[3] vel until both motors receive lim ticks, the first to arrive to the desired set point will keep on turning till the other arrives. The floating point value posCtrl will indicate what position control the library must apply to this travel, it will behave as explained in the function mt_move_t but making sure that both motors decrement the velocity at the same time. Once the motors has been set the control flow will be returned to the user program. If the function succeed true will be returned, false otherwise.

mt_wait

```
extern bool mt_wait (MOTOR * m);
```

This function takes as a parameter a pointer to a MOTOR struct m. This function is mean to be called while the motor pointed by m is in motion, it will block the user program execution until the motor pointed by m is stationary. This function will return true if succeed, false without blocking the user program if an error occurred.

mt_wait_all

```
extern int mt_wait_all ();
```

This function takes no parameters. It is meant to be called when both motors are in motion, it will block the user program execution till both motors that has stopped. It will return an integer value representing the port of the motor has stopped first if succeed, if the value returned is 2 will mean that both motors were stopped when the function was called, if an error occurred a negative value will be returned without blocking the user program.

mt_stop

```
extern int mt_stop (MOTOR * m, bool reset);
```

This function takes as a parameters a pointer to a MOTOR struct m and a boolean value reset. It will stop the motor pointed by m and, if the boolean value reset evaluates to true, it will reset the encoder lines of the motor. This function will return a negative value if an error occurs or, if everything succeed, the amount of ticks accumulated when the motor has been stopped, independently of the value of the reset field.

mt_wait_for_stop

```
extern bool mt_wait_for_stop (MOTOR * m, double delay);
```

This function takes as a parameters a pointer to a MOTOR struct m and a floating point value delay. This function is meant to be called when the motor just stopped, it will wait delay seconds and if no tick came, the function will return, if a tick comes while waiting, it will restart waiting till no tick comes in delay seconds. The function will return either true if succeed or false if an error occurred.

WARNING: For this function to work properly the ISRs of the encoder lines must necessarily update the field `tmp` of the encoder line it are in charge of via the function `clock_gettime` you can find in the `time.h` header, otherwise misbehavior will occur. The clock identifier used by the library is defined in the file `lego_motor.h` as `CLK_ID`.

`mt_shutdown`

```
extern void mt_shutdown ();
```

This function takes no parameters. It is meant to be called when the work with the motors library is all done, it will free all the allocated structures.

B.2 lego_anlog.c

In this section the functionalities provided by the file `lego_analog.c` will be explained.

B.2.1 Defaults

For the analog interface, according to the official LEGO-Pi assembly you can find in the section 3.1, the default connections will be:

```
enum lports { L_PORT0 = 3, L_PORT1 = 28, L_PORT2 = 2, L_PORT3 = 29 };
```

Where `L_PORTX` is the GPIO plugged to the yellow pin of the analog port `X`, connected, at the same time, to the analog A/D converter channel `X`. If you want to rearrange the hardware this are the definitions you must look for. Besides the library will assume that the analog port `Y` is plugged to the A/D converter channel `Y`.

B.2.2 Structures and types

Type `agType`

```
typedef enum {
  PUSH,
  LIGHT,
  SOUND,
  HT_GYRO,
  AG_OTHER
} agType;
```

Enumeration defining the kinds of analog sensors supported by LEGO-Pi, the names of the types defines clearly the sensor type it references. The type `AG_OTHER` is added to support other LEGO compliant devices not listed in the latter enumeration.

Struct `ANDVC`

```
struct analog_device {
  agType type;
  int port;
};

typedef struct analog_device ANDVC;
```

This is our representation of a LEGO analog device, the field `type` will be filled by the library and will define the analog kind of sensor plugged to this port, while the field `port` will hold the port number where the device is plugged in. Both fields will be defined dynamically by the user when initializing a port.

B.2.3 Functions

ag_init

```
extern bool ag_init(int times);
```

This function takes as a parameter an integer `times`. It must be called once before call any other function provided for this file. It will initialize the analog interface, the parameter `times` stands for how many readings of the value returned from the A/D converter must be done before returning the average of the `times` readings performed. If it succeeds true will be returned, false otherwise.

ag_set_verbose

```
#define LOG_LVL_DBG   2
#define LOG_LVL_ADV   1
#define LOG_LVL_FATAL 0

extern bool ag_set_verbose(int lvl);
```

This function takes as a parameter an integer value `lvl`. It will set the log level for the analog library, the `lvl` must range from 0 to 2, being 2 the most verbose and 0 the less verbose. If succeed the function will return true, false otherwise.

ag_new

```
extern bool ag_new (ANDVC* dvc, int port, agType type);
```

This function takes as a parameters a pointer to an analog device struct `dvc`, an integer `port`, and an analog sensor type `type`. This function will initialize the analog port `port`, setting the analog sensor type `type` as the type plugged in this port, this information will be left in the struct `dvc` for the user to be able to interact with the port. If the function succeed true will be returned, false otherwise.

ag_lgt_set_led

```
extern bool ag_lgt_set_led (ANDVC* dvc, bool on);
```

This function takes as a parameter a pointer to an analog device struct `dvc` and a boolean value `on`. It is meant to be used only for devices of type `LIGHT`, is it, LEGO light sensor. The function will set the led that provides extra light on or off depending on the value of the parameter `on`. If the function succeed true will be returned, false otherwise.

ag_lgt_get_ledstate

```
extern int ag_lgt_get_ledstate (ANDVC* dvc);
```

This function takes as a parameter a pointer to an analog device struct `dvc`. It is meant to be used only for devices of type `LIGHT`, is it, LEGO light sensor. If the function succeed it will return the state of the led as a boolean value, if the led is on true (1) will be returned, false (0) otherwise, if the function fails a negative value will be returned.

ag_psh_is_pushed

```
extern bool ag_psh_is_pushed (ANDVC * dvc, double * volt);
```

This function takes as a parameters a pointer to an analog device struct `dvc` and a pointer to a floating point value `volt`. Its meant to be used only for devices of type `PUSH`, it is, LEGO Push sensor. The function will return true if the sensor is pushed, false otherwise. If an error occurred the value left in the floating point variable pointed by `volt` will be -1, otherwise it will contain the voltage reading done to check the sensor state, it is, a value between 0 and 5, since the voltage reference of the A/D is 5V.

ag_snd_get_db

```
extern int ag_snd_get_db (ANDVC * dvc);
```

This function takes as a parameter a pointer to an analog device structure `dvc`. It is meant to be used only for devices of type `SOUND`, it is, LEGO sound sensors. The function will return a negative value if an error occurred, otherwise, it will return an integer ranging from 0 to 100, representing the dB reading from the sensor.

ag_gyro_get_val

```
extern int ag_gyro_get_val (ANDVC * dvc, bool * error);
```

This function takes as a parameter a pointer to an analog device structure `dvc` and a pointer to a boolean value `error`. It is meant to be used only for devices of type `HT_GYRO`, it is, Hitechnic gyroscope sensors. The Hitechnic gyroscope sensor readings are always the same value when stationary, the default value for LEGO-Pi is defined in the library as `HT_GYRO_DEF` and equals to 2335, however, if the sensor detects a rotation it will return a bigger value if the rotation is counter clockwise or an smaller value if the rotation is clockwise. This function will abstract the raw value read from the A/D and return negative values for an anticlockwise rotation or positives values for clockwise rotation, the bigger the value returned is the faster we are turning. If an error occurs the boolean value pointed by `error` will be set to true and -1 will be returned, else the boolean value pointed by `error` will evaluate to false. After some testing I found that values ranging from -15 to 15 can be considered a negligible reading.

ag_gyro_cal

```
extern bool ag_gyro_cal (ANDVC * dvc, int times);
```

This function takes as a parameter a pointer to an analog device structure `dvc` and an integer `times`. It is meant to be used only for devices of type `HT_GYRO`, it is, Hitechnic gyroscope sensors. As mentioned above, in the latter function, the value read from the A/D is always the same when the sensor detects no rotation, however this value may vary a little bit while using the sensor, this function is meant to be used to recalibrate the sensor, it is, reset the value expected from the A/D when stationary, so when this function is called the user must be sure that the robot is stationary, The calibration will be performed by reading the value returned for the A/D converter as many times as requested

through the `times` parameter and computing the average of the `times` readings, the bigger the `times` parameter is the more reliable will be the default value computed. If the function succeed true will be returned, false otherwise.

ag_oth_set_y

```
extern bool ag_oth_set_y (ANDVC* dvc, bool high);
```

This function takes as a parameter a pointer to an analog device structure `dvc` and a boolean value `high`. It is meant to be used only for devices of type `AG_OTHER`. If the parameter `high` is true the pin plugged to the yellow wire of analog port pointed by `dvc` will receive 3.3V, if `high` is false the pin plugged to the yellow wire of analog port pointed by `dvc` will receive 0V. If the function succeed true will be returned, false otherwise.

ag_read_volt

```
extern double ag_read_volt (ANDVC * dvc);
```

This function takes as a parameter a pointer to an analog device structure `dvc`. This function can be used for whatever analog sensor, it will return the voltage read from the A/D in the channel specified for the `port` field of the structure pointed by `dvc` as a floating point value if it succeed, else it will return a negative value. The value returned if the function succeed will range from 0 to 5.

ag_read_int

```
extern int ag_read_int (ANDVC * dvc);
```

This function takes as a parameter a pointer to an analog device structure `dvc`. This function can be used for whatever analog sensor, it will return the value read from the A/D in the channel specified for the `port` field of the structure pointed by `dvc` as an integer ranging from 0 to 4095 if succeed, otherwise it will return a negative value.

ag_shutdown

```
extern void ag_shutdown ();
```

This function takes no parameters. It must be called when all the work with analog sensors is done. It will shutdown the analog interface.

B.3 lego_digital.c

In this section the functionalities provided by the file `lego_digital.c` will be explained.

B.3.1 Defaults

For the digital interface, according to the official LEGO-Pi assembly you can find in the section 3.1, the default connections will be, for the digital port 0:

```
#define SDA_0   30
#define SCL_0   31
```

And for the digital port 1:

```
#define SDA_1   14
#define SCL_1   15
```

If you are willing to rearrange the hardware these are the definitions you must look for.

B.3.2 Structures and types

Type `dgType`

```
typedef enum {
  DG_OTHER,
  LEGO_US,
  HT_COMPASS,
  HT_ACCEL,
  HT_COLOR,
  HT_IRS
} dgType;
```

Enumeration defining the different kinds of digital sensors supported by the library, the device names stands for:

- `LEGO_US`: Lego ultrasonic sensor.
- `HT_COMPASS`: Hitechnic compass sensor.
- `HT_ACCEL`: Hitechnic acceleration / tilt sensor.
- `HT_COLOR`: Hitechnic color sensor.
- `HT_IRS`: Hitechnic infrared seeker sensor.

The type `DG_OTHER` is added to support other LEGO compliant devices that communicates via I2C protocol.

Type cmdIdx

```
typedef enum {
  US_OFF,
  US_SIN_SHOT,
  US_CONT_MES,
  US_EVENT_CAP,
  US_WARM_RESET,
  HTCS_CAL_WHITE,
  HTCS_ACT_MD,
  HTCS_PAS_MD,
  HTCS_RAW_MD,
  HTCS_50_MD,
  HTCS_60_MD,
  HTCM_MES_MD,
  HTCM_CAL_MD,
  HTIS_DSP_12,
  HTIS_DSP_6,
} cmdIdx;
```

Enumeration defining the commands we can send to some of the devices supported. There are some of the supported devices that accept commands to modify its behavior, the functions of the library will send this commands to the sensors when strictly needed, however the user may want to configure a sensor in a particular way, that's why this command types are made for. From the list of known devices for the LEGO-Pi library these are the commands an its behaviors:

LEGO ultrasonic sensor

- US_OFF: Set the ultrasonic sensor off.
- US_SIN_SHOT: Set the ultrasonic sensor in single shot mode, in this mode the sensor will make a reading of the distances every time the command is sent, and we will be able to get, hopefully, the eight closest objects to the sensor.
- US_CONT_MES: Set the ultrasonic sensor in continuous measurement mode, in this mode the sensor will be updating the closest distance permanently, theh will be left in the register 0x42 of the sensor. Only the closest distance will be available.This mode is the default for the ultrasonic sensor.
- US_EVENT_CAP: This command will set the ultrasonic sensor in event capture command, the user won't get distances in this mode, it is used to check if there is other ultrasonic devices acting near, to know when is safer to request data from the sensor.
- US_WARM_RESET: This command will request the sensor to reset, after the reset has performed the sensor will be set in continuous measurement mode.

Hitechnic color sensor

- HTCS_CAL_WHITE: Calibrate the white color. Only meant to be used by the version 1 of the sensor.
- HTCS_ACT_MD: Use ambient light cancellation. This command is only meant to be used by the version 2 of the sensor. This is the default mode of the version 2 of the sensor.

- HTCS_PAS_MD: Do not use ambient light cancellation. This command is only meant to be used by the version 2 of the sensor.
- HTCS_RAW_MD: Set the sensor in raw mode, it is, we will get the raw readings from the light sensors mounted inside the Hitechnic color sensor. This command is only meant to be used by the version 2 of the sensor.
- HTCS_50_MD: Use 50 Hz ambient light cancellation. This command is only meant to be used by the version 2 of the sensor.
- HTCS_60_MD: Use 60 Hz ambient light cancellation. This command is only meant to be used by the version 2 of the sensor. Either this command or the latter pointed (HTCS_50_MD) needs to be performed before the user can work with the Hitechnic color sensor version 2.[4]

Hitechnic compass sensor

- HTCM_MES_MD: Set the sensor in measurement mode, this is the default mode for the sensor.
- HTCM_CAL_MD: Set the sensor in calibration mode, In this mode the sensor will wait for a calibration, it is will wait till a rotation from 440°to 720°, after the rotation is done the user must set the sensor back to measurement mode and check the state of the sensor via `dg_get_state` function, if the sensor was successfully calibrated a 0 will be returned, 2 will be returned otherwise. Note that the rotation must be done in at least twenty seconds for the sensor to calibrate successfully, so the rotation mustn't be done fast.

Hitechnic IR seeker sensor

- HTIS_DSP_12: This command will set the modulation filter to 1200 Hz, it is only useful when AC readings are requested. It is meant to be used only for the version 2 of the sensor.
- HTIS_DSP_6: This command will set the modulation filter to 600 Hz, it is only useful when AC readings are requested. It is meant to be used only for the version 2 of the sensor.

If the user attempts to send a command to a sensor type not matching the command sensor type an error will occur, same thing will happens if the command is not supported by the sensor version. If you want more information of the sensor modes for each sensor you can check this site for Hitechnic devices, for the LEGO ultrasonic sensor you may refer to the Appendix 7 of the LEGO HDK.

[4] Note that what command have to be performed will vary depending on the region, you can find a list of the configuration needed on each region here

Struct DGDVC

```
struct digital_device {
  dgType type;
  int port;
  int vers;
  I2C_DVC * dvc;
};

typedef struct digital_device DGDVC;
```

This is the main struct to communicate with LEGO compliant digital devices, it is our representation of a LEGO compliant digital device. The field `type` will hold the type of sensor and will be defined dynamically by the user when initializing a digital port. The `port` field will hold the digital port the structure references an will be defined dynamically by the user too. The field `vers` stands for the version of the sensor plugged in the port `port`, this field will be filled for the library. The last field `dvc` is a pointer to the "low-level" I2C device that will be used to communicate with the sensor in question, if you want more information of the struct `I2C_DVC` you can refer to the section C.1.1 of the appendix C of this document.

B.3.3 Functions

dg_init

```
extern bool dg_init(int retry);
```

This function takes as a parameter an integer `retry`. It will initialize the digital interface, the parameter `retry` will indicate how many times an I2C transference must be retried, in case of failure, before be considered an error. If the function succeed true will be returned, false otherwise.

dg_set_verbose

```
#define LOG_LVL_DBG   2
#define LOG_LVL_ADV   1
#define LOG_LVL_FATAL 0

extern bool dg_set_verbose(int lvl);
```

This function takes as a parameter an integer value `lvl`. It will set the log level for the digital library, the `lvl` must range from 0 to 2, being 2 the most verbose and 0 the less verbose. If succeed the function will return true, false otherwise.

dg_new

```
extern bool dg_new (DGDVC * dev, dgType type, int port);
```

This function takes as a parameter a pointer to a digital device `dev`, a digital device type `type` and an integer `port`. It will initialize the digital port `port` with a device of type `type`, All the information needed will be left in the structure pointed for `dev`, for the user to be able to communicate with the device. This function is meant to be used for all known devices

to the LEGO-Pi library, it is, all the devices with type different to DG_OTHER. If the function succeed true will be returned, false otherwise.

dg_new_unknown

```
extern bool dg_new_unknown (DGDVC * dev, uint8_t addr, int freq, int port);
```

This function takes as a parameters a pointer to a digital device dev, an 8 bits unsigned integer addr, an integer freq and an integer port. This function is meant to be used for unknown devices, it is, devices with type DG_OTHER. It will initialize the digital port port with a digital device of type DG_OTHER a write address addr and a working frequency of freq.[5] All the information needed by the library to comunicate with the sensor further on will be left in the structure pointed by dev. If the function succeed true will be returned, false otherwise.

dg_send_cmd

```
extern bool dg_send_cmd (DGDVC * dev, cmdIdx cmd);
```

This function takes as a parameter a pointer to a digital device dev and a command index cmd. This function will attempt to to send the command cmd to the digital device pointed by dev, for a complete list of the commands available for each sensor you can check the type definition cmdIdx in this section. If the function succeed true will be returned, false otherwise.

dg_get_state

```
extern bool dg_get_state (DGDVC * dvc, uint8_t * state);
```

This function takes as a parameters a pointer to a digital device dvc and a pointer to an 8 bit unsigned integer state. This function can only be used for known devices, it is all the devices but devices of type DG_OTHER. It will check the sensor state for the sensor pointed by dvc, it is, it will read the LEGO "standard" state register, which is held in the address 0x41 of the sensors, the result of the reading will be left in the 8 bit unsigned integer pointed by state. The resulting state will vary depending on the sensor type and the situation.[6] usually the state will be the mode the sensor is set when this function is called, however it can vary. If the function succeed it will return true, false otherwise.

[5] Note here that the I2C library is only tested for devices that works at a frequency of 9600 Hz, which is a very low frequency for an I2C device, if any LEGO compliant device have a higher frequency the correct behavior of the library can be ensured

[6] If you want more information of the sensor states for each sensor you can check this site for Hitechnic devices, for the LEGO ultrasonic sensor you may refer to the Appendix 7 of the LEGO HDK.

dg_col_get_rgb

```
extern bool dg_col_get_rgb (DGDVC * dvc, uint8_t * red, uint8_t * green,
                            uint8_t * blue);
```

This function takes as a parameters a pointer to a digital device `dvc` and three more pointers to 8 bit unsigned integers `red`, `green` and `blue`. This function is meant to be used only for devices of type `HT_COLOR`. It will leave the read, green and blue readings from the color sensor pointed by `dvc` in the three variables pointed for the parameters `red`, `green` and `blue`. The function will return true if succeed, false otherwise.

dg_col_get_norm

```
extern bool dg_col_get_norm (DGDVC * dvc, uint8_t * red, uint8_t * green,
                             uint8_t * blue);
```

This function takes as a parameters a pointer to a digital device `dvc` and three more pointers to 8 bit unsigned integers `red`, `green` and `blue`. This function is meant to be used only for devices of type `HT_COLOR`. The function will leave the normalized red, green and blue readings from the sensor pointed by `dvc` in the three variables pointed for the parameters `red`, `green` and `blue`. The normalized readings stands for the readings of the three colors but the biggest value of the three readings will be set to 255 (the maximum value we can get in a byte) and the other two values will be rearranged proportionally. The function will return true if succeed, false otherwise.

dg_col_get_raw

```
extern bool dg_col_get_raw (DGDVC * dvc, uint16_t * red, uint16_t * green,
                            uint16_t * blue, bool passive);
```

This function takes as a parameters a pointer to a digital device `dvc`, three more pointers to 16 bits unsigned integer values `red`, `green` and `blue` and a boolean value `passive`. This function is meant to be used only for devices of type `HT_COLOR`. The function will leave the raw readings of the red, green and blue values detected for the sensor pointed for `dvc` in the three variables pointed by `red`, `green` and `blue`. If the version of the sensor is 1 the parameter `passive` will be ignored, otherwise, passive mode will be activated before the reading is performed. It will return true if succeed, false otherwise.

dg_col_get_index

```
extern bool dg_col_get_index (DGDVC * dvc, uint8_t * idx);
```

This function takes as a parameters a pointer to a digital device `dvc` and one more pointer to an 8 bit unsigned integer `idx` this time. This function is meant to be used only for devices of type `HT_COLOR`. The function will leave the color index read for the sensor pointed by `dvc` in the variable pointed for the `idx` parameter. The color index stands for a 6 bits unsigned integer indicating the strengths of the colors, the two least significant bits will be used to indicate the strength of the blue reading, the two following bits, the bits on the center, will indicate the strength of the green reading, and the most significant two bits will indicate the strength of the red reading, so the more bits are one the for each

color the bigger will be the color reading, usually, it can be used to check how dark or clear a color reading is. So for example `XXXX11` will mean a dark blue reading while `XXXX01` will mean a more light blue reading, `XXXX00` will mean almost no blue presence in the color detected by the sensor, the same pattern can be applied for the two centered bits and green reading, and the two more significant bits and the red reading. If the function succeed it will return true, false otherwise.

dg_col_get_number

```
extern bool dg_col_get_number (DGDVC * dvc, uint8_t * num);
```

This function takes as a parameters a pointer to a digital device `dvc` and one more pointer to an 8 bit unsigned integer `num`. This function is meant to be used only for devices of type `HT_COLOR`. The function will leave the value of the color number read for the sensor pointed by `dvc` in the variable pointed by `num`. The color number will vary depending on the color hue, in the picture showed you can find the list of color hues recognized by the sensor an its associated numbers. If the function succeed true will be returned, false otherwise.

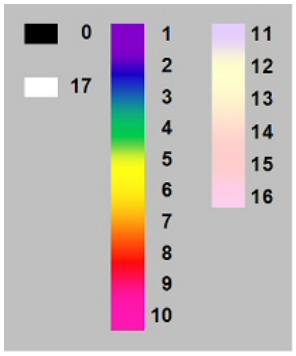


Figure B.2: Color hue numbers

dg_col_get_white

```
extern bool dg_col_get_white (DGDVC * dvc, uint16_t * white, bool raw, bool passive);
```

This function takes as a parameters a pointer to a digital device `dvc`, a pointer to a 16 bit unsigned integer `white` and two boolean values `raw` and `passive`. This function is meant to be used only for devices of type `HT_COLOR` and only for the version 2 of the sensor.[7] The function will leave the reading of the white channel of the sensor pointed by `dvc` in the variable pointed by `white`. If the `raw` parameter evaluates to true raw mode will be activated before the reading is done, hence a 16 bits value will be left in the variable pointed by `white`, otherwise, an 8 bit value will be left in the variable pointed by `white`. If the `passive` parameter is true, passive mode will be activated before perform the reading, hence ambient light cancellation will be deactivated. If the function succeed true will be returned, false otherwise.

[7] As mentioned in the testing chapter the version 2 of the Hitechnic color sensor is not tested, hence this function isn't tested either

dg_irs_get_dir

```
extern bool dg_irs_get_dir (DGDVC * dvc, uint8_t * dir, bool dc);
```

This function takes as a parameter a pointer to a digital device `dvc`, a pointer to an 8 bites unsigned integer `dir` and a boolean value `dc`. This function is meant to be used only for devices of type `HT_IRS`. The function will leave the direction number of the infrared signal detected by the sensor pointed by `dvc` in the variable pointed by `dir`. The result of the direction will range from 0 to 9, meaning 0 no signal found and values from 1 to 9 the direction number, note here that although the sensor directions may be values from 1 to 9 the sensor will hold only 5 strength registers, even direction numbers must be interpreted as if the sensor is detecting a part of the signal in each of the odd numbered adjacent directions. If the sensor version is 1 the parameter `dc` must always evaluate to true, otherwise, an `"Unsupported sensor version"` error will occur, if the version of the sensor is 2 the parameter `dc` will indicate if the user is looking for the direction of a DC signal, true in this case, or for the direction of an AC signal, false in this case. If the function succeed true will be returned, false otherwise.

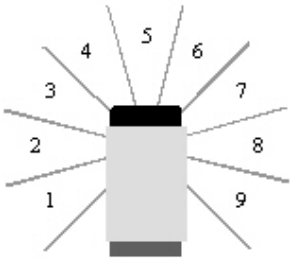


Figure B.3: IR seeker direction numbers

dg_irs_get_str

```
extern bool dg_irs_get_str (DGDVC * dvc, uint8_t * str, int num, bool dc);
```

This function takes as a parameter a pointer to a digital device `dvc`, a pointer to an 8 bites unsigned integer `str` an integer `num` and a boolean value `dc`. This function is meant to be used only for devices of type `HT_IRS`. This function will leave the strength of the signal detected for the sensor pointed by `dvc` in the direction number `num` in the variable pointed by `str`. If the sensor version is 1 the parameter `dc` must always evaluate to true, otherwise, an `"Unsupported sensor version"` error will occur, if the version of the sensor is 2 the parameter `dc` will indicate if the user is looking for the strength in the direction `num` of a DC signal, true in this case, or for the strength in the direction `num` of an AC signal, false then. The `num` parameter must range between 1 and 5, both included, representing 1 the strength in the direction 1, 2 the strength in the direction 3, 3 the strength in the direction 5, 4 the strength in the direction 7 and 5 the strength in the direction 9, as shown in the table B.4. If the function succeed true will be returned, false otherwise.

LEGO-PI num	IR Seeker direction
1	1
2	3
3	5
4	7
5	9

Figure B.4: LEGO-Pi num to IRS dir

As mentioned above, if dg_irs_get_dir returns an even direction number it must be interpreted as if a part of the signal is being detected in the two adjacent odd numbered directions, so to get the strength on an even numbered direction you must combine the strengths in the two adjacent odd numbered directions. You can refer to the figure B.3 to get the IR Seeker direction numbers.

dg_irs_get_allstr

```
#define IRS_STR_TABLE_TAM       5

extern bool dg_irs_get_allstr (DGDVC * dvc, uint8_t * strt, bool dc);
```

This function takes as a parameter a pointer to a digital device dvc, a pointer to an 8 bits unsigned integer array strt and a boolean value dc. This function is meant to be used only for devices of type HT_IRS. This function will leave the strengths for all the directions in the array pointed by strt, the array must be of length 5. If the sensor version is 1 the parameter dc must always evaluate to true, otherwise, an "Unsupported sensor version" error will occur, if the version of the sensor is 2 the parameter dc will indicate if the user is looking for the strengths of a DC signal in all the directions, true in this case, or for the strength of an AC signal in all the directions, false then. The strengths will be ordered in the table as explained in the latter function (dg_irs_get_str). If the function succeed true will be returned, false otherwise.

dg_irs_get_dcavg

```
extern bool dg_irs_get_dcavg (DGDVC * dvc, uint8_t * avg);
```

This function takes as a parameters a pointer to a digital device dvc and a pointer to an 8 bits unsigned integer avg. This function is meant to be used only for devices of type HT_IRS and only for the version 2 of the sensor. It will leave the average strength of the DC signal in all the directions in the variable pointed by avg. If the function succeed true will be returned, false otherwise.

dg_us_get_dist

```
extern bool dg_us_get_dist (DGDVC * dvc, uint8_t * dist, int num);
```

This function takes as a parameter a pointer to a digital device dvc a pointer to an 8 bit unsigned integer dist and an integer num. This function is meant to be used only for devices of type LEGO_US. It will leave the distance to the num closest object in the variable pointed

by dist. Note here that, if the parameter num is greater than 0, single shot mode will be activated to get the required distance, once the data is returned continuous measurement mode will be enabled again. If the function succeed true will be returned, false otherwise.

dg_us_get_alldist

```
#define US_DIST_TABLE_TAM 8

extern bool dg_us_get_alldist (DGDVC * dvc, uint8_t tdist []);
```

This function takes as a parameters a pointer to a digital device dvc and a pointer to an 8 bits unsigned integer array tdist. This function is meant to be used only for devices of type LEGO_US. It will get all the distances to the 8 closest objects and leave it in the array pointed by tdist, the array must be of length 8. Note here that I2C communications are slow for the ultrasonic sensor, so, if you need more than one distance is faster to use this function than the latter one explained dg_us_get_dist. This function will return true if succeed, false otherwise.

dg_com_get_head

```
extern bool dg_com_get_head (DGDVC * dvc, uint16_t * h1, uint16_t * h2);
```

This function takes as a parameter a pointer to a digital device dvc, a pointer to a 16 bits unsigned integer h1, and one more pointer to a 16 bits unsigned integer h2. This function is only meant to be used by devices of type HT_COMPASS. It will get the heading readings from the sensor and leave it in the variables pointed by h1 and h2. Note here that two heading values will be returned, the sensor provides two heading values, one of them will be read in two bytes from two registers, while the other will be read in one byte from a register (and multiply it for 2) plus a one degree adder read from another register, both values are returned because in some situations there is a little difference between the values, specially when the sensor heading is closest to the north, so the user can choose the value used. The values left in the variables pointed for h1 and h2 will range from 0 to 359, being 0 north, 180 south, 270 west and 90 est.

dg_acc_get_axis

```
extern bool dg_acc_get_axis (DGDVC * dvc, int * x, int * y, int * z);
```

This function takes as a parameters a pointer to a digital device dvc and three more pointers to integer values x, y and z. This function is meant to be used only for devices of type HT_ACCEL. The function will leave the acceleration values for the x, y and z axis in the variables pointed by x, y and z. The values left in the variables will be approximately 200 units per G, it is, since the gravity will be distributed between the axis it can be used as a tilt sensor too, however, bumps or holes in the way of the robot will affect the sensor readings too. This function will return true if succeed, false otherwise.

dg_get_info

```
#define DG_INFO_TABLE_TAM 9

extern bool dg_get_info (DGDVC * dvc, char * info [], bool titled);
```

This function takes as a parameters a pointer to a digital device `dvc`, a pointer to an array of strings `info` and a boolean value `titled`. This function can be used for whatever known sensor, it is, all the sensor types but `DG_OTHER`. The function will fill the table pointed by `info` with information of the sensor pointed by `dvc` such as the product version, sensor type, etc. Since we get more info fields from the LEGO ultrasonic than from Hitechnic devices the table pointed by `info` needs to be of length 9, however, the table will be filled with void strings (`""`) from the last useful info field till the end of the table. If the parameter `titled` evaluates to true it will concatenate a title before each info field explaining what means the returned string, else, only the information itself will be left in the array pointed by `info`. This function will return true if succeed, false otherwise.

dg_transfer

```
extern bool dg_transfer (DGDVC * dev, uint8_t data_out [], int len_out, bool rs,
                         uint16_t data_in [], int len_in, bool word_read, int cse);
```

This function can be used for whatever sensor type, however, it is meant to be used for devices of type `DG_OTHER`, since there is no other functions available for it. What this function does is to expose the "low-level" I2C library `i2c_transfer` functionality for the user to be able to communicate with the unknown digital sensor. The library will use the I2C device (`I2C_DVC`) held in the structure pointed by the `dev` parameter to perform the communication, if you want a more detailed explanation on the parameters of the function and return values you may refer to the section C.1.2 of the appendix C of this document.

dg_write

```
extern bool dg_write (DGDVC * dev, uint8_t data_out [], int len_out, int cse);
```

This function can be used for whatever sensor type, however, it is meant to be used for devices of type `DG_OTHER`, since there is no other functions available for it. What this function does is to expose the "low-level" I2C library `i2c_write` functionality for the user to be able to communicate with the unknown digital sensor. The library will use the I2C device (`I2C_DVC`) held in the structure pointed by the `dev` parameter to perform the communication, if you want a more detailed explanation on the parameters of the function and return values you may refer to the section C.1.2 of the appendix C of this document.

dg_read

```
extern bool dg_read (DGDVC * dev, uint16_t data_in [], int len_in, bool word_read,
                     int cse);
```

This function can be used for whatever sensor type, however, it is meant to be used for devices of type `DG_OTHER`, since there is no other functions available for it. What this function does is to expose the "low-level" I2C library `i2c_read` functionality for the user to be able to

communicate with the unknown digital sensor. The library will use the I2C device (`I2C_DVC`) held in the structure pointed by the `dev` parameter to perform the communication, if you want a more detailed explanation on the parameters of the function and return values you may refer to the section C.1.2 of the appendix C of this document.

dg_shutdown

```
extern void dg_shutdown ();
```

This function takes no parameters, it is meant to be called when all the work with digital devices is done, it will generate an stop condition on every port initialized and exit.

B.4 lego.c

In this section the functionalities provided by the file `lego.c` will be explained, this functionalities are not strictly needed to interact with the peripherals, the files explained above `lego_motor.c`, `lego_analog.c` and `lego_digital.c` provides all the methods to communicate with the sensors and motors and can be used in "standalone" mode, however this file adds some functionalities in the hope of it will be useful to the user, it are meant to be used when all the interfaces are required for the user.

B.4.1 Definitions

Macro DELAY_US

```
typedef struct timespec TSPEC; /* < From time.h > */

#define DELAY_US(t)         nanosleep((TSPEC*)&(TSPEC){0, t*1000}, NULL)
```

This definition will block the user program for `t` microseconds, since we are meant to be control robots short delays are often useful in all kind of situations, so this definition is provided to the user for him to be able to do this in a straightforward way.

Macro DIFFT

```
typedef struct timespec TSPEC; /* < From time.h > */

#define DIFFT(ti,tf)        (((*tf).tv_nsec - (*ti).tv_nsec)/1000 +\
                             ((*tf).tv_sec   - (*ti).tv_sec)*1000000)
```

This macro expects as arguments two pointers to a TSPEC structure, being `ti` a time stamp taken sooner in time and `tf` a time stamp taken later in time, it will compute the time difference in microseconds between the two time stamps.

B.4.2 Functions

lego_init

```
extern bool lego_init (int dg_retires, int ag_avg);
```

This function takes as a parameter two integer values `dg_retires` and `ag_avg`. This function will initialize all the interfaces provided by the library, it is, the motor interface, the analog interface and the digital interface. The parameter `dg_retries` will be passed to the `dg_init` function and the parameter `ag_avg` will be passed to the `ag_init` function, so you can refer to the specification of this functions to get a more detailed explanation on how it will behave. If the function succeed it will return true, false otherwise.

lego_set_verbose

```
#define LOG_LVL_DBG   2
#define LOG_LVL_ADV   1
#define LOG_LVL_FATAL 0

extern bool lego_set_verbose (int level);
```

This function takes as a parameter an integer `lvl`. This function will set the log level of all the initialized interfaces to `lvl`, being 2 the most verbose and 0 the less verbose. if the function succeed true will be returned, false otherwise.

lego_shutdown

```
extern void lego_shutdown ();
```

This function takes no parameters. It will shutdown all the interfaces provided by the library previously initialized.

Appendix C

I2C Library Specification

This appendix is dedicated to the I2C library used by the LEGO-Pi project.

C.1 lego_i2c.c

In this section the functionalities and the structures provided by the file `lego_i2c.c` will be explained. No code will be added to this section, however you can reach it in the directory `lib` of the LEGO-Pi git-hub . As mentioned in the section 2.4.2 of this document this is a very simple implementation of the I2C protocol that will be able to run from user space, There is no clock stretching support, only 7 bit address are supported, and no arbitration is supported neither. The I2C library uses the GPIO numbering scheme, so every time I refer to a GPIO number I'll be meaning the GPIO number, not the number of the pin on the board. This library must not be used by the user willing to control LEGO peripherals, LEGO-Pi will use it however, hence the specification is populated. For this implementation a word will be an integer of 16 bits.

C.1.1 Structures and types

Struct `I2C_DVC`

```
struct i2c_device {
  int sda;
  int scl;
  double thold;
  uint8_t addr;
};

typedef struct i2c_device I2C_DVC;
```

This is the main struct for the I2C library, it represents an I2C slave device. The `sda` field is the GPIO plugged into the SDA line of the slave, while the field `scl` is the GPIO plugged to the SCL line of the slave. The field `thold` is used internally for the library, it is the delay time we will wait between clocks. The field `addr` is the write address of the slave device.

C.1.2 Functions

i2c_init

```
#define LOG_PRINT 1
#define LOG_QUIET 0

extern bool i2c_init (int log_lvl);
```

This function takes as a parameter an integer `log_level`. It will initialize the I2C interface, the parameter `log_lvl` will indicate the log level desired for the user, being 1 more verbose than 0. If the function succeed true will be returned, false otherwise.

i2c_set_loglvl

```
#define LOG_PRINT 1
#define LOG_QUIET 0

extern bool i2c_set_loglvl (int log_lvl);
```

This function takes as a parameter an integer `log_level`. It is used to reset the log level of the I2C library at runtime, being 1 more verbose than 0. If the function succeed true will be returned, false otherwise.

i2c_new_device

```
#define GPPUD_DISABLE   0x0  /* < Disables the resistor > */
#define GPPUD_PULLDOWN  0x1  /* < Enables a pulldown resistor > */
#define GPPUD_PULLUP    0x2  /* < Enables a pullup resistor > */

extern bool i2c_new_device (I2C_DVC * dvc, uint8_t addr, int freq, int sda, int scl,
                            uint32_t sda_pud, uint32_t scl_pud);
```

This function takes as a parameters a pointer to an I2C device structure `dvc`, an 8 bits unsigned integer `addr`, an integer `freq`, two more integers `sda,scl` and two 32 bits unsigned integers `sda_pud, scl_pud`. This function will initialize a new I2C slave device with a write address `addr`, with a working frequency of `freq` and with the SDA and SCL lines plugged into the GPIOs `sda` and `scl` respectively. The parameters `sda_pud` and `scl_pud` are used to indicate if the user wants pull-up or pull-down resistor in the SDA or SCL lines, the values it can take are 0 to disable the resistor, 1 to enable the pull-down resistor, or 2 to enable the pull-up resistor.[1] If the function succeed it will return true, false otherwise.

i2c_read

```
extern bool i2c_read (I2C_DVC * dvc, uint16_t * data_in, int len_in, bool word_read,
                      int cse);
```

This function takes as a parameters a pointer to an I2C device struct `dvc`, a pointer to an array of 16 bit unsigned integers `data_in`, an integer `len_in`, a boolean value `word_read` and an integer `cse`. Whats the function will do is perform an I2C read to the slave device pointed by `dvc`, the result of the read must be of `len_in` bytes or words that will be placed

[1] if you want more information about the internal pull-up/down resistors of the Raspberry Pi GPIOS you can refer to this site

in the array pointed by `data_in`. The parameter `word_read` is used to indicate the library if the responses will be byte-sized (8 bits per block), false in this case, or word-sized (16 bits per block), true then. Once the read is performed the function will wait `cse` microseconds before returning the program flow to the user program. If everything went right true will be returned, false will be returned otherwise.

i2c_write

```
extern bool i2c_write (I2C_DVC * dvc, uint8_t * data_out, int len_out, int cse);
```

This function takes as a parameters a pointer to a I2C device struct `dvc` a pointer to an array of 8 bit unsigned integers `data_out`, an integer `len_out` and one more integer `cse`. What the function will do is perform an I2C write transaction with the slave device pointed by `dvc`, sending `len_out` bytes held in the array pointed for `data_out`. Once the transaction is done the function will wait `cse` microseconds before returning the program flow to the user program. If everything went right true will be returned, false will be returned otherwise.

i2c_transfer

```
extern bool i2c_transfer (I2C_DVC * dvc, uint8_t * data_out, int len_out, bool rs,
                          uint16_t * data_in, int len_in, bool word_read, int cse);
```

This function takes as a parameters a pointer to an I2C device struct `dvc`, a pointer to an array of 8 bit unsigned integers `data_out`, an integer `len_out`, a boolean value `rs`, a pointer to an array of 16 bit unsigned integers `data_in`, an integer `len_in`, a boolean value `word_read` and an integer `cse`. This function will behave as the if the two latter functions explained are called in streak, it is, it will perform a write transference with the I2C slave device pointed by `dvc` followed for a read transference. The bytes to be sent must be placed in the array pointed by `data_out` which must be of length `len_out`. The data the device send back will be placed in the array pointed by `data_in` which must be of length `len_in`. The parameter `word_read` will indicate that the responses of the sensor will be byte-sized (8 bits per block), false in this case, or word-sized (16 bits per block), true then. If the user wants to generate a repeated start condition (STOP + START) between the two transactions the parameter `rs` must be true, otherwise an start condition will be generated. Once the transactions are done the function will wait `cse` microseconds before returning the program flow to the user program. If everything went right true will be returned, false will be returned otherwise.

i2c_shutdown

```
extern void i2c_shutdown (I2C_DVC * dvc);
```

This function takes as a parameter a pointer to an I2C device struct `dvc`. This function must be called when all the I2C work is done, it will generate an stop condition on the device pointed by `dvc` and exit.