UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

FABIEN CHIFFRE

Architecture review and test tool development for the eBikeCortex

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Science

Porto Alegre
December 2017

RESUMO

O eBikeCortex é um controlador para bicicletas elétricas que permite ao usuário configurar a assistência elétrica da sua bicicleta. O eBikeCortex tem muitas vantagens sobre outros produtos no mercado. Porém, do lado técnico ele precisa evoluir.
Seu firmware é desenvolvido em um bloco único. Fica difícil reusar os modulos dele, principalmente a parte logica responsável pela computação da assistência elétrica.
Uma arquitetura modular permite criar um firmware mais robusto, que pode ser testado de uma maneira mais fácil e também de reusar esta parte logica.
Esta parte logica também tem que ser testado antes de ser usado em produtos reals. Porque ela é integrado em um sistema de circuito fechado, a ferramenta de teste tem que simular o ambiente do sistema.
Em fim, a parte de integração no Hardware tem que ser totalmente revista para ter uma nova versão do eBikeCortex que respeita esta nova arquitetura modular.

Palavras-chave: E-bikes, Sistemas embarcados, Software engineering, Test.

ABSTRACT

The eBikeCortex is an electric bicycle controller, which allows to the users to configure the power assist of their bicycles. The eBikeCortex has many advantages over other products on the marke. But, on the technical side, it needs to evolve.
Its firmware is developed in a single block. It is difficult to reuse its modules, especially the logical part responsible for computing the power assist.
A modular architecture allows to create a more robust firmware, which can be tested in an easier way and also to reuse the power assist computation part.
This logical part also has to be tested before being used in real products. Because it is integrated into a closed loop system, the testing tool has to simulate the system environment.
Finally, the integration part in Hardware has to be entirely reviewed to have a new version of eBikeCortex, which respects this new modular architecture.

Keywords: E-bikes. Embedded System. Software engineering. Tests.

ACKNOWLEDGMENT

I would like to thank all members of the eBikeLabs team for hosting me during this 6-month internship.
I would like to especially thank Maël Bosson and Raphaël Marguet, co-founders of eBike-Labs for giving me the chance to work in their company and to discover the startup world from the inner side.
I would also thank the members of the R&D team: Julien Fleche, Pierre-Louis Jourdan and Théophile Alary for the help they provided to accomplish my project.
Finally, I would like to thank Erika Cota and Claudio Geyer from the Universidade Federal do Rio Grande do Sul for their support during my year in Brazil.

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS AND ACRONYMS

ADC	Analog to Digital Converter
BLE	Bluetooth Low Energy
DMA	Direct Memory Access
DAC	Digital to Analog Converter
HIL	Hardware-in-the-loop
RPM	Rotation Per Minute

CONTENTS

1 INTRODUCTION

1.1 The eBikeCortex

The eBikeCortex is an e-bike controller used to regulate the e-bike power assist. It is developped by the french startup eBikeLabs[2].
Current e-bike references do not have a power assist control system that really meets the users' needs. In the worst case, this system is a simple button that will activate or not the motor. Most of the time, the system offers only a limited range of power assist levels. It rarely allows to totally customize how the power assist works and the user has to adapt himself to his e-bike when it should be the contrary. Moreover, power assist control is a thing that is specific to an e-bike brand and when the user decides to change his e-bike, he has to learn again how his new product works.
The eBikeCortex solves this issue by offering a fully customizable power assist based on two different modes:

- With the first one, the user configures a pedaling frequency that he would like to reach. Then, the power assist helps him to reach this target. If the user slows down (for example if he goes through an ascent, his pedaling frequency will drop down), the power assist will increase as compensation. Otherwise, if the user pedals faster than his target frequency, the power assist will reduce.

- The second one is more recent and only works on e-bike with an engine brake. In this mode, the user also configures a target pedalling frequency. If his pedaling frequency is too low, the controller acts like as described in the first mode and helps the user. But, when his pedalling frequency is too high, the motor starts to brake. Used with gear levers, it allows to quickly adapt the bike speed. Indeed, if the user selects a lower gear ratio, his pedaling speed will increase and the engine will start to brake. Selecting a higher gear ratio will reduce his pedalling frequency and the assistance increases. This mode is very comfortable for users, who are biking in city because they can change their speed without having to change an assistance level.

Once the mode has been selected, the user determines his target pedaling frequency, the maximum speed of his bike, a speed from which the engine brake should be activated and

also the maximum power the motor is allowed to provide.
This last parameter offers a great deal of flexibility. It can be used to regulate the physical effort the user wishes to provide. For instance, in the case of a typical home-work trip, the user wishes probably probably to assign a high power value to do the trip easily and with low physical effort. On the other hand, on a sporting outing with friends with non-electric bikes, one may choose a lower power value. Inthis case, the user needs to make more physical effort to follow his friends.
With all these parameters, the controller determines the power assist to be provided at each moment. It also meets some safety rules: the power assist must shutdown immediately if the user brakes or stops pedaling. For comfort purposes, the power assist increases quickly at the start and then ensures that changes are not too sudden.
In short, the eBikeCortex is designed to work on the largest number of e-bike, to ensure user safety and to offer a customizable power assistance. The fact that it cannot work with pedal-assist is currently its major limitation.

1.2 Problem to solve

Even with all these advantages, the eBikeCortex has a big problem: its firmware is built in a monolithic architecture. All the logical blocks responsible for computing the power assist are mixed with the hardware parts responsible for integrating it to the microcontroller.
This architecture has two big consequences:

- First, the logical part cannot be reused in other products and now that eBikeLabs start to develop a new controller it becomes a critical point.
- Secondly, tests are very complicated to realize as may have to process a single enormous block. Identify the error source is, consequently, very difficult.

This work proposes an architecture review for this firmware, which aiming at solving these two problems.

1.3 Objectives

The goal of this project is to move the eBikeCortex firmware from a monolithic architecture to a modular one. This change implies:

- The development of a library responsible for the power assist computation. This library should be independant of any part directly related to the hardware allowing its use on different microcontrollers.
- The conception of a test tool to validate this logical library independently of a microcontroller platform. This tool should be used for non-regression tests too.
- The development of a new eBikeCortex firmware which respects the new system architecture. This final point allows to demonstrate how the chosen architecture can be applied on a real system.

1.4 Text organization

Chapter 2 describes the eBikeCortex, with its global system architecture, the advantages over the other product of the market and then concludes with the current firwmare architectures with its porblems. It includes some information about global e-bike concepts which are useful for a better comprehension of other Chapters.
Chapter 3 presents a new architecture proposal.
Chapter 4 explains how is it possible to test the power assist library.
Next, Chapter 5 presents the implementation of the solution described in Chapter 2 and 3.
Finally, Chapter 6 concludes with the results and also describes some perspectives for the eBikeCortex.

2 THE EBIKECORTEX

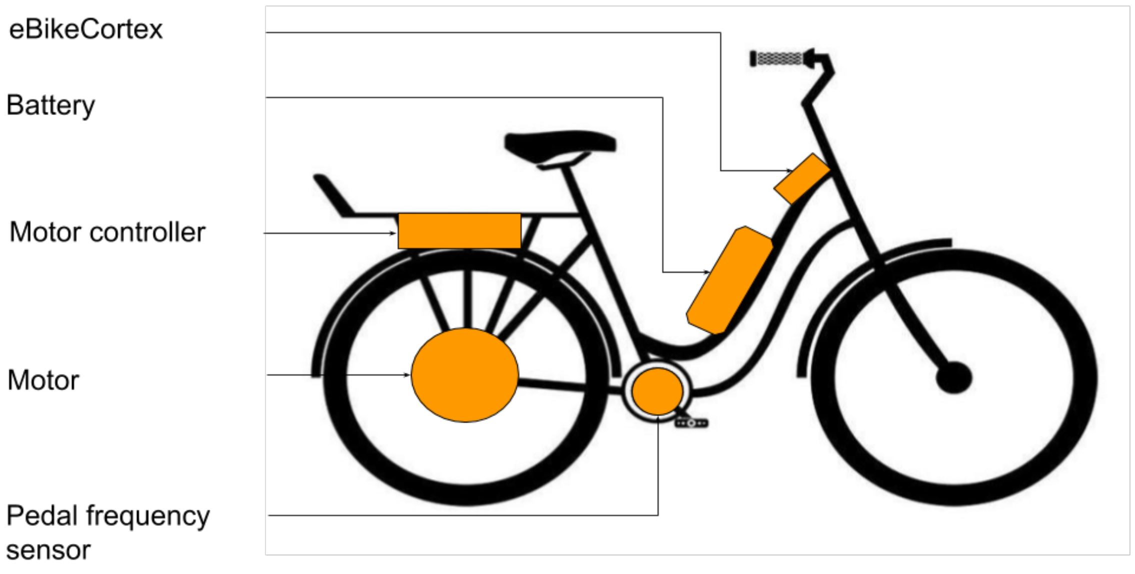
2.1 System architecture

The controller eBikeCortex is built over a STM32F412 and a HM10, Bluetooth Low Energy chip. Other hardware elements are needed to get it working:

- A motor controller which is connected to the motor, to the eBikeCortex through an electric wire.
- A motor that must be set on the wheel. The current eBikeCortex does not work with pedal motor.
- A battery physically connected to the motor and to the volmeter and the amperemeter of the motor controler.
- A pedal frequency sensor set on the pedal and connected to the eBikeCortex through an electirc wire.
- A brake sensor which is also connected to the eBikeCortex through a wire.

The Figure 2.1 diagram shows where are set up the different elements.

Figure 2.1: e-bike mapping



All these hardware elements allow to capture the following data:

- The pedaling frequency, expressed in revolutions per minute (RPM).
- The bike speed, in meters per hour.

- The battery voltage, in millivolts.
- The battery intensity, expressed in milliamperes
- The occurrence of a braking event.

The computation of the power assist is done in a 100Hz loop. At each iteration, the eBikeCortex collects the data described above. It then uses the power assist library which produces two outputs: an assistance command, which is used to control the motor, and a braking command, which is used to regulate the braking engine intensity. Both commands should obviously not be activated at the same time. These commands are then transmitted to the motor controller which applies it on the motor.

The battery is one of the key elements of an e-bike. Each battery has its own voltage profile and capacity. The voltage profile is a curve that gives the remaining battery capacity with a given voltage. Indeed, the battery voltage drops down when the battery discharges. When the eBikeCortex is turned on, it reads this battery voltage and gets the battery level. Because it is able to get the battery intensity provided by the battery at each moment, the eBikeCortex can compute the total consumption and so it gets the remaining capacity at each moment. In addition, as it gets the battery consumption and the battery voltage, the eBikeCortex can also deduce the battery power provided to the motor. It can then regulate the power to an acceptable level for the motor and to respect the power constraint configured by the user.

For the brake event, a simple sensor is placed on the brake controller of the bike. Thus, when the user breaks, a signal is sent to the eBikeCortex to notify the action.

Concerning the pedaling frequency, the sensor is placed on the crankset. This sensor is simply a disk with some magnets and a magnet detector. When the user pedals, the crankset turns and the magnets pass in front of the sensor. Each magnet detection allows to know the time between two magnets and, as the number of magnets is known the system is able to determine the pedaling frequency.

The computation of the bike speed is done according to a similar principle. Hall effect sensors[1] are set up in the motor. The time between each tick of a sensor allows to determine the rotation speed of the motor. As the motor is placed on the wheel it is possible to know the wheel rotation speed, and so with the size of the wheel the bike speed.

The eBikeCortex has two different user interfaces. The first one is a simple LED display, which shows the current battery level and can indicate some problem like a failure on a sensor or a too high motor temperature. The second one is built over the BLE. An

[1]A Hall sensor effect is used to detect changes in a magnetic field.

Android application can connect to the bike. This application allows the user to change his configuration. It can also collect all the sensor data and presents it through a display screen. So, when the user is biking he can know his current pedaling frequency, his current speed, etc...
The Android application is also able to send a new firmware over the BLE which allows the user to update his product easily.
The organization described in this part was already determined by the development team and was not changed by this project.

2.2 Other solutions

The eBike market is quite young but grows very quickly. 1 188 000 eBikes were imorted into the European Union. Compare to 2015, this number increased by 63%[3].
Based on this strong growth, a lot of constructors start to developped their own solutions. Bosch is one of the biggest eBike motors constructor in Europe and sell its own "on-boards"[1].
Main advantage of this solution is the perfect compatibility with the motor manufactured by Bosh. But, on the other hand such controller cannot be used in other model. In addition, it only provides some level of assistance that the user has to select. Thus, even if it gives all the indications to help the user to choose the right level, this level will not change automatically.
Another exemple is Piaggio with its Wi-Bike. It has a very complete Android application and an adaptive power assit. This offer is closer than the one provide by the eBikeLabs but it is still specific to the ebikes manufactured by Piaggio.
Most of the time, smaller ebikes manufacturers do not provide a such developped controller with few power assist level and specific to the model or to the brand.
eBikeLabs differs from the previous example because it is not an eBike manifaturer and only develop a controller. Thus, the development s focused on the power assist management. This controller is also supposed to be compatible with a large range of motors which is something quite new.

2.3 The initial firmware architecture

As mentionned in Chapter 1, the eBikeCortex firmware was built on a monolithic architecture. In such software architecture, the data input, data processing and use of the user interface are often mixed and cannot be clearly isolated.
In the eBikeCortex, the parameters saving logic, the BLEBLE: Bluetooth Low Energy communication and the power assist computation do not have clear limits. Plus, these blocks do not have clear entry and exit points.
Thus, the power assist computation directly uses some sensor drivers to get the needed inputs. It then processes immediately these inputs and can produce errors, which are directly notified to the user through to LED drivers. Finally, the output is directly converted into voltage for the motor controller.
For the eBikeCortex, this architecture is the result of two years of iterative developments without a firmware design clearly estabilshed. It allowed to develop the eBikeCortex quickly but it is not possible to reused single part as easy as it should be.
Still, with the eBikeCortex as en example: the eBikeLabs started to develop a new controler integrating some other feature like motor control. The company would like to reuse the power assist computation but realized soon that it was not so easy. Difficulties come mainly from the output handling and the fact that these output are not clearly isolated.
Moreover, in embedded hardware components can change during the product life. And this change can impact the sensor driver that is used too. On a single block application, this change will impact all the block and can introduce some new errors which can be difficult to catch.

3 PROPOSED ARCHITECTURE

3.1 General concepts

The idea is to move from a monolithic architecture to a more modular one, composed of different blocks. To do so, each main feature has to be considered as a feature block which could be used in another system. Thus, it should be independent of the hardware drivers. Port functions could be necessary to ensure a good interface between the feature blocks and the drivers. Then the application code is responsible for connecting each feature blocks to the hardware layer.
The main advantage of such an architecture is to build blocks that can be easily reused in other applications. On one hand, the development time for each block increases. But, this time quickly pays off if the module is reused. On the other hand, build an application using too much of modules can also increase the code size, which can be problematic in embedded system.

3.2 Modularizing the eBikeCortex

For the eBikeCortex, three feature blocks were designed:

- A module including all the logical part of the power assist computation. It should allow to get an assistance command from input given by the application.
- A memory management module used to save data used and produced by the application.
- A last module responsible for the BLE communication. Difficulty here is to clearly separate what is part of the BLE stack, part of the BLE hardware layer and part of the eBikeCortex communication protocol. The module should include only these last elements.

Due to time restriction, the project was focused on the creation of the main feature block, the power assist library. The library creation process could be appied to create the other feature blocks.

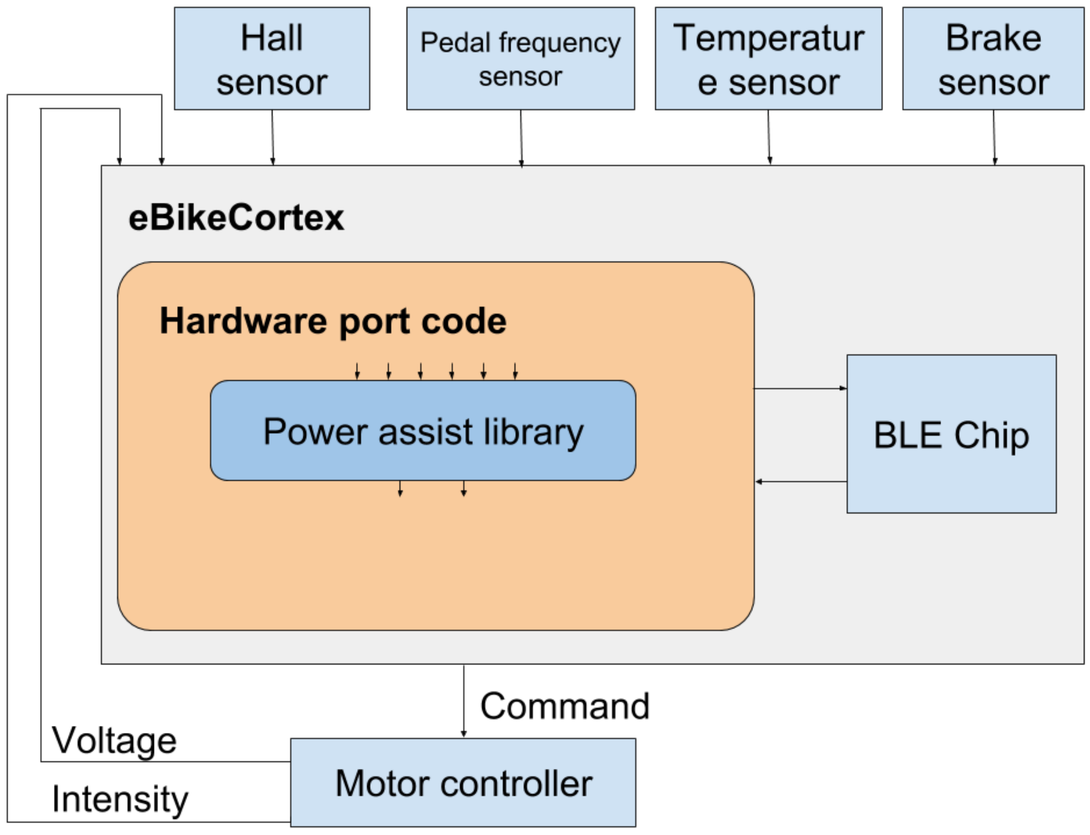
3.3 Power assist library module implementation

The power assist library is the most important feature block used by the eBikeCortex controller. It allows to compute the power assist to be provided to the user according to the current system state.
To be independent from the platform, the application has to be responsible for capturing system state through sensors. The library is only responsible for the logical computation. So, to be able to build an isolated block, inputs have to be clearly identified. Most of the time the inputs could be classified in two categories:

- Inputs needed for the initialization. They are used to properly configure the library. It could be hardware specific element, user configurations, etc. For the power assist library, the inputs are the targetted RPM value, the motor power limits configured by the user, some motor configurations, the indication if the motor has an engine brake, etc.
- Data to be provided during runtime. These data are needed for the power assist computation when the system is running and corresponds to the system current state. For the e-bike power assist library the following parameters are used: the current pedaling frequency, the current battery voltage and intensity, the current bike speed and the occurrence of a braking event.

With these input lists defined, the module workflow has to be determined. For the power assist library it is quite simple. When the system starts or when the BLE part notifies a reset event, the application layer initializes the library with the proper parameters. Then it runs the library in a loop. At each iteration, the system state is captured through sensors by the application. These data are then passed to the library which produces a command in output. Application is then responsible for applying this command.
Following this workflow, all parts related to the hardware are handled by the application or other modules. Consequently, the power assist module can be reused in other applications but can also be tested independently.
Figure 3.1 represents ther architecture implemented. Only the power assist library was modularized.

Figure 3.1: New eBikeCortex system architecture



4 PROPOSED TEST SOLUTION

The new eBikeCortex architecture also allows to review the test process. Now that the architecture clearly separated the logical part from the hardware layer both can be tested individually. More precisely, the logical part can be tested independently and once it is validated, its integration to the hardware can be tested. If an error occurs,during this last step, by elminiation, the error comes from the hardware layer or from the interface between the different system's elements.

4.1 Tesing the logical part

4.1.1 Concept

The power assist library uses the system current state to compute a command which is then transmitted to a motor controller, which applies the command to the motor. This action direclty affects the current system state: the intensity provided by the battery increases if the motor has to provide more power or decreases otherwise. The bike speed changes too and the pedaling frequency of the user can be modified.
But, as the system has sensors to captured all of these data, the effect of the previous command is known and the power assist library can take another action with this feedback. Shortly, the output of the power assist library has an influence on its own inputs. It is the definition of a closed-loop system.
In this project, the test of this system has two main goals:

- Being able to simulate the behavior of this system.
- Being able to validate this behavior.

4.1.2 Environment simulation

As mentionned, the system influences its own inputs. Consequently, to be able to simulate a coherent behavior this influence has to be simulated. Otherwise, the system will produce a command, check gain the inputs, see there is no influence, increase the command and so on...
So the problem is clear: to be able to test the power assist library, only the inputs which

can be directly changed by the user or external parameters can be fixed. The other have to be simulated from the system initial state and the output of the power assist library simulation.
The Hardware-in-the-loop (HIL) method[9] is a well known method to simulate complex control systems and is very used for embedded systems. It uses both hardware and software elements to build a realistic system simulation. For instance the actuator can be the real controller and all the sensor can be simulated.
Even if this method is initially not built for software components, like the power assist library, it provides a lot of concept about the simulation of a closed-loop systems that can be reused.
Indeed, in the majority of HIL implementation, the sensors and a big part of the environment are simulated. It helps to reduce costs and allows to observe the system behavior in a laboratory.
The sensor simulation represents the way the control system acquires its data. It should match the way a real sensor works to be more realistic. For instance, for the eBikeCortex, the pedaling frequency sensor should be simulated as it is a mechanic sensor with its own limitations.
The environment simulation represents the control system global state at each instant. For an ebike, it includes its speed and the motor power. This environment is impacted by the simulation of the actuator which correspond to the output of the control system.
For all these elements, closer simulation is to a real behavior the better will be the whole system simulation will be.
In this project, the last element of the simulation is the power assist library which corresponds to the element to observe and is the control part. All the simulation is realized through software components.
But, even if HIL is not used at this stage because we want to validate the software component, it could be used to check if the behavior is still the same after the library has been integrated to the hardware. Sadly, due to time restriction, this point has not been explored furthermore.

4.1.3 Behavior validation

The second goal is to check if the behavior of the simulated system corresponds to the expected one. As the system is complex and critical to the user health, this part is very important. But, because the objectives had a lot of objectives, there was not enough time to do a complete literature review about real-time component validation and develop and validate the test tool. Consequently, this part reuses only some part of the existing literature and only focuses on the software behavior validation. The real-time part validation was not explored.

On a such system, we cannot simply compare the output of the simulation with known values. Indeed, due to the system complexity, it is very difficult to establish a collection of valid output vectors. Moreover, a little change on the system can impact all the library outputs and the test tools will produce an error while the behavior stay the same.

The idea is to use a list of scenarios to validate the system requirements [10]. A scenario corresponds to a list of actions done by the user, for our tools it corresponds to the input of the simulation. Then, when the scenario is executed, a validation will be done over the output to determine which software requirements have to be applied and if they have been applied correctly.

For the power assist library validation, the software requirements corresponds to a check-list of constraints extracted from the specifications of our power assist mechanism. Once the scenario is simulated, the patterns which have to trigerred these contraints are looked for into the scenario input and if it is detected, the output pattern is looked for into the simulation output. If it is correctly detected, the constraint application is validated otherwise, an error is detected.

For each constraint of our list some elements have to be determined:

- When the constraint has to be checked? This is defined by the current system state. In our case it is simple as there are only three: during the start phase, the stop pĥase or the normal one.
- What triggers the constraint? It can be a data which goes over a threshold, a braking event...
- What is the effect of the constraint? For instance, the command can drop down to 0 or grow up, the system state changes...
- And then what is the constraint priority? Indeed, some have an higher constraints and should be applied first. Consequently, constraint with lower priorities will not be applied.

As these constraints are related to the system state, the power assist library state machine has to be determined.

4.2 The hardware part

The hardware part testing is greatly limited limited by the company's resources. So tests are more empirical here.
The company's already has the hardware to test basic system reactions. This platform simulates the brake and the pedaling frequency sensors to check if the controller correctly generate a command and stops it during the braking action.
Once these basic elements are checked, the controller can be setup on a bike on a home-trainer to test it with different user actions: different pedaling frequency, braking actions...
Finally, when there are no more aberrations (e.g an ebike which never stop, an assistance with too much variation, etc.), it is possible to use this bike on real conditions. As it is used daily in the company, it allows to detect big integration problems.

5 WORK IMPLEMENTATION

5.1 Power assist library creation

The code extraction was not so difficult to do, but it was a prerequisite for all the other tasks. It gave me a good knowledge about how the power assist works, which was useful for the test tool development.
Once the list described in Section 3.3 was established, the separation was easier than it seemed at the first look.
Some parts needed additional port functions for application or other module interactions. It is the case of the pedaling frequency sensor. Indeed, the power assist library includes the logical mechanism to determine if the biker stops to pedal or not. So, in addition to the classic pedaling frequency, the library has to know if the frequency value is a new one or an old one stored in a buffer.
To get a similar behavior on different platforms, the library integrates also a filtering part for data provided by the application layer.

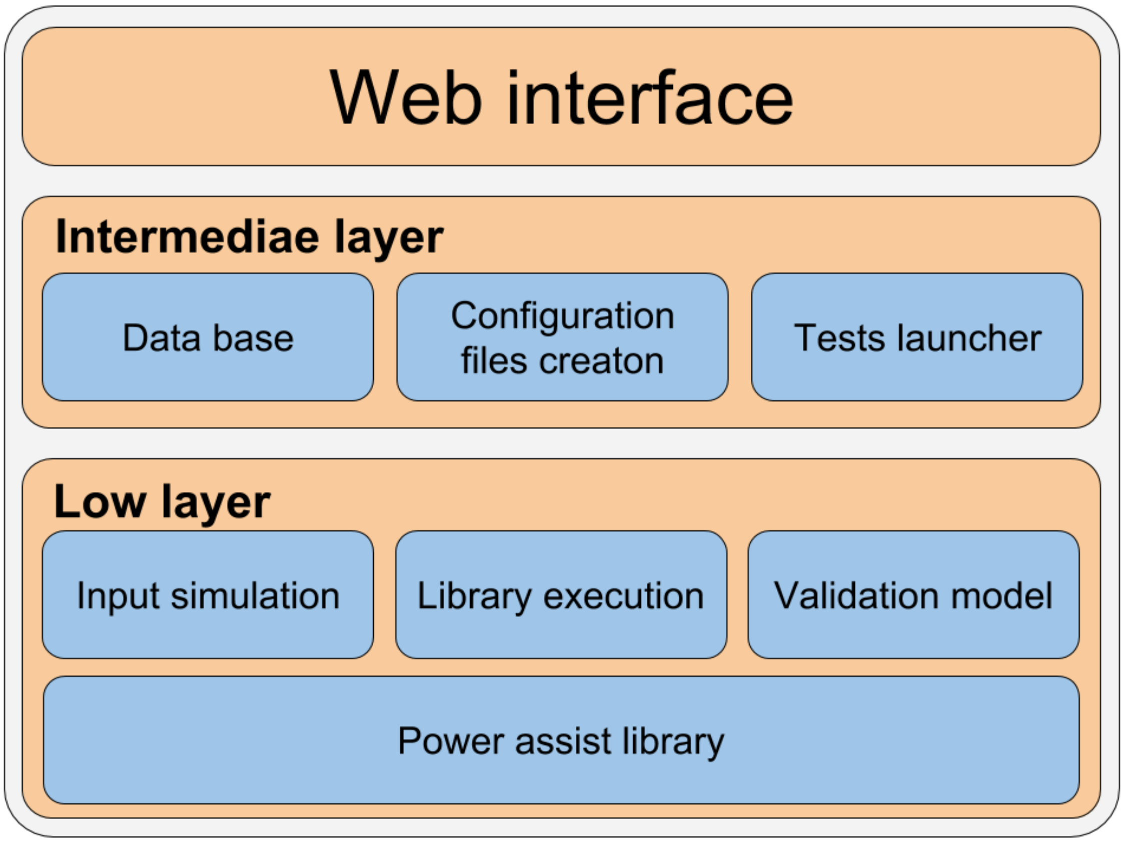
5.2 Test tool for the power assist library

5.2.1 Architecture

This test tool is developed as a web application and allow to simulate the behavior of the power assistance library and also to execute a list of non-regression tests.

Figure 5.1 represents test tool architecture.

Figure 5.1: Design of the test application



The test application is built in three different layers:

- A lower layer written in C which interacts directly with the power assistance library and is responsible to execute the simulation. It simulates some inputs too to get more realistic results and includes a validation model to check if some constraints are respected.
- An intermediary layer built with Python Django[1][8]. It is used to build the configuration files which are passed to the lower layer and to handle the storage of the motor, batteries and scenarios collections.
- A web interface allowing to use the test tool in an understandable way. It displays the result with curves and allows to create a test scenario easily. It is developed in HTML and JavaScript.

[1]Django is an open source framework used to code web application with Python

5.2.2 Environment simulation

To be as close as possible of a real system, most of the inputs are simulated. In fact, for a given test scenario, only the inputs which can be directly impacted by the user can be configured. Such inputs are:

- The pedaling frequency.
- The occurrence of a braking event.

The pedaling frequency should represent the input of a real sensor and needs to be filtered by the test application. In fact, if the data is given to the library as it is at each iteration, in a 100Hz loop, it means that a new event has been detected every 10ms. On a classic pedaling frequency sensor with 12 magnets, it corresponds to a pedaling frequency of 500RPM which is totally non realistic for a human.
So, at each iteration, the new value of the pedaling frequency is read from the scenario file and compared to an inner counter. This counter determines wether the value could be effectively captured by the sensor. If it can, the value is given to the library. Otherwise, the previous value is used to represent a real sensor with a buffer. The filtered result is very closed to a real sensor and allows to easily simulate the fact that the user stops to pedal.
As explained in Section 4.1.2, other inputs have to be simulated to get coherent results. It is the case of the bike speed, the battery voltage and the battery intensity. They are all simulated based on initial conditions and the power assist library output.
Thus, to get the intensity and the voltage of the battery we need to know the motor power because:

$$P_{motor} = P_{battery}$$

Where P is a power in Watts. And $P = UI$ for U a voltage in Volts and I an intensity in Amperes. As the battery level is one of the initial conditions, it is possible to get $U_{battery}$ from the battery profile. So, if we get P_{motor} we can determine $I_{battery}$.
For the motor, the voltage is computed with:

$$U_{motor} = K_{motor}V$$

Where V is the bike speed and K_{motor} is a constant specific to a motor and given by the constructor. The intensity is computed by:

$$I_{motor} = \frac{\alpha CMD - U_{motor}}{2R}$$

CMD corresponds to the output of the power assist library and varies betwen 0 and CMD_{MAX}. αCMD corresponds to the voltage provided by the battery. α is computed by:

$$\alpha = \frac{U_{batterie}}{CMD_{MAX}}$$

So, at this point, with a constant speed, the motor power increases with the command. As the battery voltage drops down very slowly, the motor power increase corresponds to an augmentation of the battery intensity, which corresponds to a behavior close to the reality. Now that the motor power is known it is possible to compute the bike acceleration at each iteration. A simple model is used:

$$a = (F_{motor} - F_{resitance})\frac{1}{m}$$

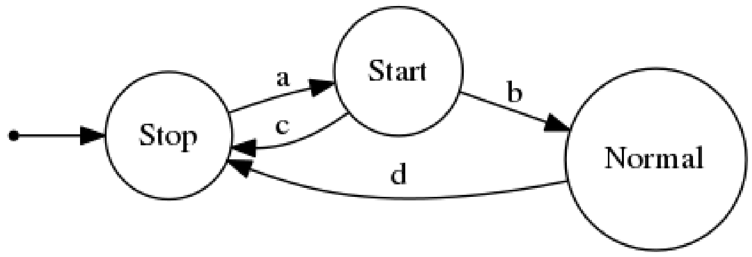
Where m is the mass of the system. F_{motor} is computed with $\frac{P_{motor}}{V}$. The resistance is essentially composed from frictions and is simplified with $\beta V + \delta$ where β are δ fixed values. This resistance can also be used to simulate a slope.

The acceleration a and the speed of the previous iteration allows to know the new speed.

5.2.3 Behavior validation

Now, output has to be validated as explained in section 4.1.3. First the power assist state machine has to be determined. For this case it is quite simple:

Figure 5.2: Power assist library state machine



The state transitions respect the following rules:

Table 5.1: Power assist lirbary transistion table

Transition name	Reasons
a	The user starts to pedal and the pedaling frequency gets higher than the start threshold (20 RPM).
b	- The user is in the start for 1 second - The pedaling frequency reaches the targeted one - The bike speed exceeds the maximum one
c	The user brakes for 50 ms or stops pedaling.
d	The user brakes for 50 ms or stops pedaling.

The list of constratins to check is longer (lower priority number corresponds to higher priority constraint):

Table 5.2: Power assist library constraints

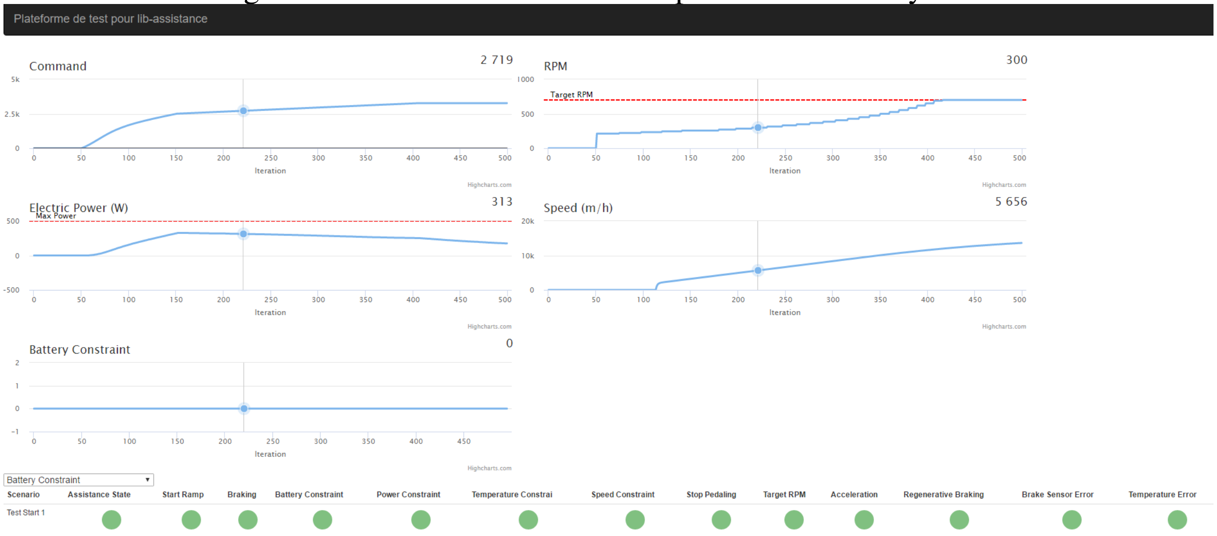
System state	Priority	Activator	Effect
Stop	-	-	Assistance command must be 0
Stop	-	User is braking	If the motor has a brake engine, brake command should grow linearly with the time of the braking action.
Start	-	-	Command should grow following as $O(n^2)$ function according to the time in the state.
Normal	-	-	Difference between two consecutive command values should not be higher than a fixed delta.
Normal	1	Power is lower than the minimal threshold	An inner command threshold is determined according to the current power value, the minimal one and the current command value. Command should increase to get higher than this threshold. Application can be checked on the next iteration.

Normal	1	Power is higher than maximal value	An inner command threshold is determined according to the current power value, the maximal one and the current command value. Command should drop down to get lower than this threshold. Application can be checked on the next iteration.
Normal	2	Speed is higher than the engine braking threshold	Assistance command should drop down to 0 immediately and braking command should increase linearly according to the bike speed. System state do not change. [2]
Normal	3	Speed is higher than a minimal threshold	As the power constraint, an inner threshold is computed based on the current speed, the maximal one and the command value. Command should decrease until it is lower than this threshold. Application can be checked on the next iteration.
Normal	4	The pedaling frequency is higher than the targeted one	Command should decrease linearly. Application can be checked on the next iteration.
Normal	4	The pedaling frequency is lower than the targeted one	Command should increase linearly. Application can be checked on the next iteration.

After the constraints and states list are established, the validation can be done. The input and the produced output of the library are passed to the validation model which will determine the current system state. Then, it takes the list of the constraints that should be applied for this state starting on with he highest priority. If the constraint should be triggered, its application is checked and the constraints with lower priorities are skipped.

[2]This constraint is only valid if the motor has a engine braking.

Figure 5.3: Simulation result of the power assist library



Scenario	Assistance State	Start Ramp	Braking	Battery Constraint	Power Constraint	Temperature Constrai	Speed Constraint	Stop Pedaling	Target RPM	Acceleration	Regenerative Braking	Brake Sensor Error	Temperature Error
Test Start 1	●	●	●	●	●	●	●	●	●	●	●	●	●

5.2.4 Presentation layers

As described in section 5.2.1, the intermediate layer and the higher layer are dedicated to make the tool easily usable through a web interface.

A server part was developed in python Django. To facilitate test execution, this part maintains a data base of motors, batteries and scenarios. The scenarios are saved a tuple (date, value). Obviously, most of the dates do not have an associated value. When the scenario is chosen and before the simulation, the known values are interpolated to get a value for each date.

Scenarios are split into two categories. The scenarios corresponding to the non-regression tests and a history of all executed simulations. Thus, if a problem occurs during a test, it is possible to do it again after some fixes.

A web interface displays the results collected during the simulation and displays it through curves to be easily understandable. These curves display the bike speed, the pedaling frequency, and the output of the library. Some flags of the library can also be displayed and can be used to know the internal state of the library.

In addition, the results of the validation model are displayed too and indicate for each constraint if it is respected or not.

Figure 5.3, shows a screenshot of the result page during a simulation of start phase.

5.3 eBikeCortex firmware

Once the power assistance library had been created and tested, the idea was to develop a new release of the eBikeCortex firmware following the new design.
The controller used for this development is based on a STM32F415 connected to a BLE model HM-10.
Firmware was developed step by step in order to reduct the test effort. Development steps were the following:

- Integration of the different kind of sensors.
- Integration of the new power assist library with a mock configuration.
- Integration of the BLE chip to allow configuration and data reporting.

5.3.0.1 Integration of the power assistance library

First phase was important as before being able to integrate the power assist library it was necessary to execute a basic operating system on our controller. In our case, we decided to use FreeRTOS[7].
FreeRTOS is a real-time operating system kernel. It is well-supported on STM32 due to the big community. Moreover, the development team already had some knowledge about it.
It is a multitask system: the developer create tasks with priorities, functions and some time intervals and the system schedules the task executions to match the system requirements.
To fulfill this phase, two FreeRTOS tasks were created.

- The first one is dedicated to the power assist library loop. It collects data from the different sensors, executes the library and then converts the output to a voltage for the motor controller. It is the higher priority task of the system and should be executed with a 100Hz frequency.
- A display task using LED to display error and battery level. It has a low priority and 5Hz frequency. This display can also be used for debug purposes during real bike debugging.

All the sensors were also managed during this development phase:

- The pedaling frequency sensor works thanks to STM32 timer and interruption mechanisms. Each time a new magnet is detected, an interruption is triggered and the timer value is read.

- The speed sensor is handled with the same mechanism. Another interruption channel is used but the timer is the same.
- A temperature sensor allows to monitor the motor temperature and works thank to an ADC[3] connected to a DMA controller[4]. The DMA controller directly updates the sensor value and allows to have the last one available when the power assistance library has to be executed.
- The battery intensity and voltage are handled too with ADC and DMA mechanism. Both values are provided by the motor controller and not by a battery management system.
- The brake sensor is handled like a button. The state of a GPIO pin is read and the logical level indicates if the user is braking or not.

The output commands of the power assistance library are then converted by a DAC [5] which sends it to the motor controller.
This phase was not so difficult but it took longer to validate. The first tests were run on a hardware test platform realized by the team. This platform simulates the brake and the pedaling frequency sensor to check if the controller correctly generates an assistance command and stops it during the braking action.
Once these basics were checked, the controller was setup on a bike on a home-trainer to test it on different behavior: different pedaling frequency, braking actions, etc. Finally, when the behavior was the same than the previous firmware release, it was possible to use this bike on real conditions. As it is used daily in the company, it allows to detect some bugs and as the power assistance library was already tested, these bugs were coming from the hardware integration layer.

5.3.0.2 BLE integration

The aim of this second phase was to be able to configure the eBikeCortex and so to set up the controller on other bikes.
The BLE communication is built over an existing BLE chip, the HM-10. The chip is connected through a serial connection to the STM32. A new FreeRTOS task is created

[3]Analog to Digital Converter: it is an electronic component able to convert an electrical value into a numeric one.

[4]Direct Memory Access: mechanism allowing to a subsystem to directly access the memory.

[5]Digital to Analog Converter: an electronic component able to convert a numerical value into an electric one

to handle this communication. It has a medium priority and 20Hz frequency. Sadly, no interruption lines were available to notify when the chip has some data to transmit and so, the BLE task has to run continuously to check if some data have been received.
Even with this defect, the communication between the STM32 and the HM-10 was improved. Previously, the STM32 sent the data to the HM-10 through busy waiting which should possibly block the system. This busy waiting was removed to use one more time the DMA controller but that time for output purpose.
A custom protocol based on GET and SET commands is built over BLE to read and write parameters into a Flash memory. Another command then updates the system to apply the new parameters.
Checking that previous eBikeCortex would be able to update correctly its firmware and without loosing its parameters was the most important part of this task. Indeed, all parameters are saved on specific flash addresses. After an update, the firmware retrieves its parameters from the same addres. If two data adresses were swapped on a new version, the firmware would not get a bad value which could lead to some unexpected behavior. The same error can occurred if the data units are not exactly the same.
The eBikeCortex still does nit have an efficient system to support parameters update and there was not time to develop one. So, I carefully used the same memory addresses with the same units for the same parameters.
The update was handled by a bootloader which did not change on the new firmware. So the only part to test was the parameter coherences after an update through the Android application.
To test the parameter states, test controller was configured with a previous firmware version and then updated. Afterwards, the parameter states were dumped. Such tests allow to detect a problem on the battery profile where the units diverge between the two releases.
Once this part has been developed and tested the new firmware was set up on all the company e-bikes. After one month of test, it was published through the Android application and started to be used by real users.

6 CONCLUSION

The eBike Cortex changed from a monolith to a modular architecture. This change required a lot of work on the existing code base but allowed to reach the project goals.
Some changes are still needed. The modular architecture was applied for the power assist library but other modules could be reviewed too. The test tool could be also improved to simulate real trips. This could improve greatly the parameter simulation with the slope integration.
A test tool was developed to validate this library and its simulation are closed to the behavior observed in real use. These simulations are very useful for the development. First, it allows to observe the behavior of the library on specific case without having to put it on a real bike. Secondly, it allows to launch non-regression tests on new version of the library.
The hardware integration was totally reviewed too to demonstrate how to use the new architecture. Product stays unchanged from the user point of view but the reliability increases as modules and hardware integration are now tested independently.

REFERENCES

[1] Bosch ebike systems. https://www.bosch-ebike.com/en/products/on-board-computer.

[2] ebikelabs website. https://ebikelabs.com/index.php/en/.

[3] E-bike import continues to show huge growth. lhttp://www.bike-eu.com/sales-trends/nieuws/2017/8/e-bike-import-continues-to-show-huge-growth-10131018.

[4] STM32F415 datasheet. http://www.st.com/resource/en/datasheet/stm32f415rg.pdf,.

[5] STM32F415 reference manual. www.st.com/resource/en/reference_manual/DM00031020.pdf,.

[6] Description of stm32f4xx hal drivers. www.st.com/resource/en/user_manual/dm00105879.pdf,.

[7] FreeRTOS documentation. http://www.freertos.org/Documentation/RTOS_book.html,.

[8] Django documentation. https://docs.djangoproject.com/en/1.10/, .

[9] M. Bacic. On hardware-in-the-loop simulation. In *Proceedings of the 44th IEEE Conference on Decision and Control*, pages 3194–3198, Dec 2005. doi: 10.1109/CDC.2005.1582653.

[10] A. G. Sutcliffe, N. A. M. Maiden, S. Minocha, and D. Manuel. Supporting scenario-based requirements engineering. *IEEE Transactions on Software Engineering*, 24 (12):1072–1088, Dec 1998. ISSN 0098-5589. doi: 10.1109/32.738340.