POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Politecnico
di Torino

Master Thesis in Embedded Systems

Development of a touch screen display with haptic functionality and a graphical user interface in a heterogeneous multi-core and multi-processor environment

Supervisors

Prof. Luciano LAVAGNO

Ing. Flavio CERRUTI

Candidate

Simone DI BLASI

October 2021

Narrami, o Musa, le avventure dell'uomo ricco d'astuzie, che a lungo vagò ...

Abstract

The aim of this thesis was to apply haptic functionality to a touch screen display and to develop a graphical user interface that shows the potentiality of this product. The haptic perception is the tactile sensation obtained as a result of interaction with surfaces and objects. This sense of touch can be artificially recreated by stimulating the human body through feedback, such as the vibration given by the smartphone after a long press.

The measure of the force and the stimuli generation, main actors in the haptic perception, were made using an actuator, the Niceclick, which thanks to an ad-hoc circuit transforms, when a force is applied, its magnetic reluctance variation into a pulse that can be easily read by a microcontroller. In addition, the actuator can also generate a sequence of pulses making up the haptic feedback. For these reasons, the generation of the haptic feedback can be done by checking if the force applied on the surface exceeds a settled threshold or generates it immediately once the presence of the finger, or a gesture, is confirmed. The firmware of the microcontroller works in a real time environment to make the haptic management and the operative tasks precise and repeatable. The operative task consists in the reception of commands, coming from the CAN Bus, their interpretation to perform specific operations and giving back the feedback status. This workflow is necessary because the display is managed by another processor, the iMX8M mini, which uses the external microcontroller as a slave sending commands or requests to manage the haptic part.

The iMX8 processors are a family of heterogeneous multi-core processors, where two different types of cores work in the same environment, sharing peripherals and memory portion. In this processor there is the Cortex A53 in charge to manage the graphic part of the application and the Cortex M4 that acts as microcontroller with a real time OS. The latter is used as interface between the graphic processor and the external microcontroller, passing the information back and forth managing the CAN communication.

Finally, the developed GUI display different interfaces that show the potentiality of this technology: an automotive dashboard, where the sense of press and release of a switch can be felt, images where a texture is perceived and a configurable panel where the force threshold and haptic effect could be changed.

Another goal of this thesis was to verify if the iMX8M mini could manage the entire system on its own, in other words, if the Cortex M4 could also perform the work done by the external microcontroller, while A53 handles the graphics. This study confirmed that M4 can handle the haptic part, giving improvement from the communication and computational point of view, however there is a lack of a peripheral which can be compensated re-adapting the control circuit.

To conclude, the use of heterogeneous processors allows to meet the growing demand for embedded devices with high computing power and low power consumption while the addition of haptic sensations to a device opens new scenarios to different technologies, meeting the most diverse needs.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The aim of this thesis is to apply haptic functionality to a touch screen display and develop a graphical user interface (GUI) that shows the potentiality of this product. Haptic is the science of touch and how people interact with things. Specifically, it defines the sensations perceived as a result of interacting with an object, such as material or response feedback. Nowadays, to narrow the gap between human and machine interface, this sense of touch has been introduced in several devices by stimulating the human body through the application of forces, vibrations or movements. Simple haptic devices are the game controllers and steering wheels, which transmit sensations to the user's hands by making him perceive a physical resistance.
However, several devices on the market have very limited haptic capabilities, which are used to notify an event rather than trying to reproduce reality. For this reason, this project demonstrates by addressing these shortcomings that a normal used device, such as a display, if provided with haptic capabilities can give sensations and reproduce objects in completely new ways.
TRAMA Engineering, the company where I carried out my thesis, has developed a linear actuator, which connected to a special control system is able to generate a sequence of pulsations that are perceived as a haptic sensation. Moreover, the same device is able to perform measurements on the force applied on the surface transforming its magnetic reluctance variation into pulses that can be easily read by a microcontroller, allowing to evaluate also possible orthogonal movements such as press.
The goal is to take this device and place it in the center of the system, realizing a firmware able to manage the haptic part in an effective way and at the same time receive external requests of execution and/or control from the CAN Bus.
This board, including actuators, must be thought as an add-on for a wider system that in this case is a touch screen display with its control processor.
The display is controlled by an i.MX8M mini processor, mounted on a dedicated development board. The work done on this processor concerns the realization of an interface with the external microcontroller that controls the haptic part and the

realization of a graphical application with which the user can interact.
To achieve these goals is exploited and highlighted the particularity of this processor to be a heterogeneous system, that is to integrate within it two cores of different architectures. The main core is mainly focused on graphics options and complex calculations while the second core manage real time tasks and system peripherals. The two cores can exchange information through a portion of shared memory following a protocol.
The decision to use this processor was made also with the purpose of testing if it could manage the whole system by itself, in other words, if it could also do the work done by the external microcontroller replacing it, while continuing to manage the graphics part.
Finally, the development of the graphical application should have highlighted the possibilities and functionalities of this technology allowing the user to interact with the screen understanding the difference between having and not having haptics on a device.

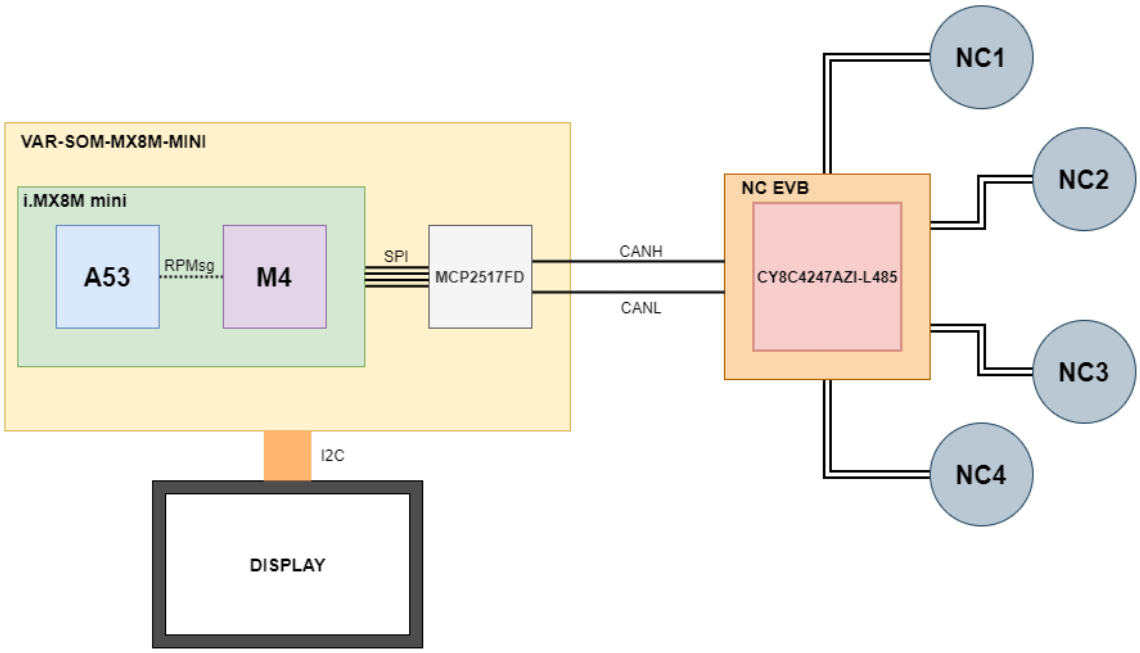

Figure 1.1: Full system.

1.1 Background

The starting point of the thesis was to study and analyze the elements already available and understand how to integrate, improve and implement what was missing.
The main element on which the entire project is based is the Niceclick, a linear actuator that together with its control circuit is able to generate haptic sensations and measure the applied force. This mechanism, consisting of both hardware and software, had already been developed and tested. In particular, the hardware part, in addition to the actuators (described in the 2.3 section), consists of a board integrating a microcontroller, capable of controlling the actuation and measurement circuits, using the internal peripherals to stimulate each external component. In the chapter 3 the structure of this circuit, its operation and how the microcontroller is expected to act on it are explained.
For what concern the software part, a support library, with two main functions, was already prepared. These functions are aimed at verifying that a set force threshold has been exceeded and at actuating a haptic effect. The main work, however, is done by a task, to be triggered periodically, with the purpose of updating all background data by evaluating the system state, measuring the force, and checking if there are any actuation requests (described in the section 4.2.1). This task must work in conjunction with an application task, which varies in base of the design, that uses the functions described above.
So, the basic haptic system had already been developed and the first approach was to study the physical and mathematical models on which it was based, understanding how the application of a force would bring changes to the value of the inductance, of which the actuator is made, and how this change could be measured to obtain force values. Then move on to study the software structure, the basic task, the library, the data structures, and how a haptic effect can be translated into code (4.1).
All this system had been developed with the purpose of controlling predefined haptic elements with a static position, in order to associate to a specific position a predefined effect. The touch detection could be done using capacitive membranes, since the board has a special connector, and all the system management, haptic and application tasks, would be handled by the microcontroller.
Since the goal of the thesis was to expand this structure to a touch screen display, this organization could not be applied but it was necessary to rework the software part in order to make dynamic the use of haptic elements, programmable by the user the haptic widget definition and the receipt of feedback on the state of actuation. This would have allowed to make this device a slave at the expense of a master that would have controlled it by making haptic requests.
This led to a modification of the library's APIs and various data structures in the already prepared code, adding all the firmware necessary to manage the requests and sending of feedback as well as identifying possible bugs and improvements (explained in the 4.3 section).
Once completed the development and testing of the firmware of the haptic board, the focus turned to the device that would control it via CAN, the heterogeneous

processor i.MX8M mini with its development board.
In this case, both the processor and the type of environment that should be implemented were new even for the company where I did my internship, in this case the study was based on knowing what is and how it works a heterogeneous system, what implications should be taken into account to manage it and what benefits it can bring (5). The work was then divided into two parts, as there were two cores to manage. For the development of the code of the secondary core was used an SDK of NXP, which provided libraries and drivers, in particular those for the management of RTOS and inter-process communication (sections 6.1.2 and 6.1.3). It was absent instead the driver to manage the CAN module on the board, of which I made a porting of the library from another processor and a wrapper of a driver to use it (6.1.1). Finally, the final firmware was implemented, as described in the 6.2 section.
The main core is managed by an Embedded Linux OS, whose build for this architecture was downloadable from the website of the company that produced the module. In the build there were also Kernel modules and drivers essential for the management of various peripherals and protocols, such as display control and the communication with the other core. Instead, the graphic application has been realized from scratch trying to define a back-end part that was independent from the adopted graphic so that it could be used in more contexts (see 7.3). To support the definition of the graphical elements the Qt framework and its basic tools were used, on which the whole application was structured (7.4).
In conclusion, the project started from a solid base for what concern the haptic part, even if the technology is under development, with supports that allowed to dwell mainly on the application and expansion part. While the research part has focused on the study of multi-core heterogeneous systems, with the aim of evaluating the possibility of a complete control by a single processor of the entire system.

1.2 Thesis outline

The thesis is divided into eight chapters. Initially the theory and technology on which this project is based on will be explained and then the development part and the choices that have been made will be analyzed.
The work will be discussed according to the following structure:

- **Overview on Touch and Haptic Technology**
  In Chapter 2, a brief overview of haptic technology is provided to the reader, illustrating both theoretical aspects and technologies on the market, in order to better understand the design choices made during the development process.

- **Niceclick technology**
  Chapter 3 explains the hardware system supporting the selected haptic actuator. The design choices are highlighted, as well as how the pulses are physically generated and how the force was intended to be measured.

- **CY8C4247AZI-L485 Firmware: Haptic Management**
  Chapter 4 explains how the firmware of the microcontroller that controls the haptic part was implemented. The main functions and operations performed, both of control and communication part, are reported.

- **Heterogeneous multi-core processor: i.MX8M mini**
  In Chapter 5, an overview of the microprocessor used to control the graphics and the display is provided. In addition, the peculiarities of this processor and the reasons why it was chosen are explained giving also details about its carrier board.

- **i.MX8M mini - Cortex M4: Firmware**
  Chapter 6 explains the code on the secondary core of the i.MX8M mini, the drivers and libraries used. Emphasis is placed on its role as an interface between the main core and the external microcontroller, but also investigating its future possibilities.

- **i.MX8M mini - Cortex A53: Graphical User Interface**
  Chapter 7 discusses the implementation of the graphical application and what principles it is intended to convey. In particular, the back-end part that manages the application, with the choice to adopt multithreading to improve performance, and the front-end part that creates the graphical components are explained.

- **Conclusion**
  The conclusive considerations and possible future improvements are discussed in chapter 8.

Chapter 2

Overview on Touch and Haptic Technology

This chapter explains what an haptic technology is and what elements are needed to make a device that is capable of generating haptic sensations.
In particular, an overview is provided on the main technologies that allow the identification of contact on a surface and the measurement of the applied force. The tracking and measurement of force are two of the three main players that allow the creation of a haptic interface. The remaining element concerns the generation of the feedback. The union of these three elements is what made it possible to create the final device on which this thesis is based, guaranteeing different haptic functions for both generated effects and gesture identification.

2.1 Haptic technology

Haptic perception is the tactile sensation obtained as a result of interaction with surfaces and objects, allowing a wide variety of exploration and manipulation tasks in the real world. Without haptics, we would have great difficulty grasping and manipulating objects, be unable to determine many materials or surface properties[1]. However, this sense of touch must be artificially recreated by stimulating the human body by applying forces, vibration, or motions to reproduces the characteristics of interaction needed to enhance realism.
Simple haptic devices are the game controllers or steering wheels, which transmit sensations to the user's hands, for example by making him perceive a physical resistance. Others haptic devices could be tablets and smartphones that have sensors to measure human inputs, but their outputs are limited primarily to the visual and auditory channels, and the quality of the haptic interactions (vibrations) leaves much room for improvement.
Hence, haptic can be used for a variety of purposes and in a variety of contexts.

Category	**Description**	**Possible application**
Spatial	Using haptic information to indicate the location of important objects	Awareness of surrounding
Warning	Using haptic information to warn the user in dangerous situations	Collision prevention
Communication	Using haptic information as a subtle communication channel	Navigation
Information	Using haptic information to display current status information regarding the device	Speed control
Interaction	Using haptic information in interaction with control units	Controlling the device's functions
Aesthetical	Using haptic information to evoke aesthetical appreciation	Perceived quality

Table 2.1: Possible applications of haptic technology.

A typical interaction with a haptic device consists mainly of two phases: haptic search and haptic confirmation. The haptic search refers to the functional use of haptic surface properties, such as joints, ridges, edges, surface geometries and textures as orientation and indication. This "search" becomes difficult when using flat surfaces such as displays or touch and gesture-sensitive surfaces. Instead, haptic confirmation refers to feedback obtained as reply to a change in the operating state of a control element. The simplest example is the feeling of a button click after pressing it[2].
However, every concept of interaction is based on a technological counterpart which leads to the division of haptic devices into two categories: passive and active.

- **Passive haptics**: haptic feedback generated by mechanical elements, or physically anchored and permanent stimuli. It involves no external electrical energy input. The energy is generated by pressure from the user, and the haptic feedback is generated by the reaction of the mechanical elements to this energy[2].
- **Active haptics**: haptic feedback that requires external electrical energy input. Typically, a sensor reacts to tangential (sliding, swiping) or orthogonal (press) movements of the user which then triggers an actuator. The actuator moves the interaction surface in a manner often characterized by high acceleration and short travel[2].

Among the two technologies, the one most associated with haptic concepts is the active one. Furthermore, the greatest benefits that can be derived from using an active haptic device are its programmability and flexibility. Haptic impulses can be changed depending on application and situation, while the same technology can be used in different use cases.

Figure 2.1: Overview on different terminologies of haptics.

Haptic feedback in consumer products is usually synonymous of vibration feedback. Vibrations provide an added dimension to gaming by simulating the sensation of collisions, the buzzing of a phone allows us to receive notifications and vibrations have been used to simulate the feel of a button click[1]. Common vibration actuators are:

- **Linear Resonant Actuator (LRA)**: is a vibration motor that produces an oscillating force across a single axis. LRA relies on an AC voltage to drive a voice coil pressed against a moving mass connected to a spring. When the voice coil is driven at the resonant frequency of the spring, the entire actuator vibrates with a perceptible force. Unlike a speaker, the frequency and amplitude may be adjusted by changing the AC input to create different effects.
- **Eccentric Rotating Mass (ERM)**: is like a conventional DC motor and uses the magnetic field from an electrical current to drive an object in a circle. The rotating mass is off-center from the point of rotation, for this reason when

the motor rotates produce a centripetal force which causes the motor to move backwards and forwards and vibrations.

- **Piezoelectric Actuators**: utilize the piezo effect to generate vibrations, i.e. the generation of electricity when a material is stressed mechanically. Piezo actuators are more precise than the others detailed above, because they can vibrate at a wider range of frequencies and amplitudes but also in more than one direction. These actuators require a higher voltage than their counterparts, but the current consumption is less than ERM and comparable to LRA.

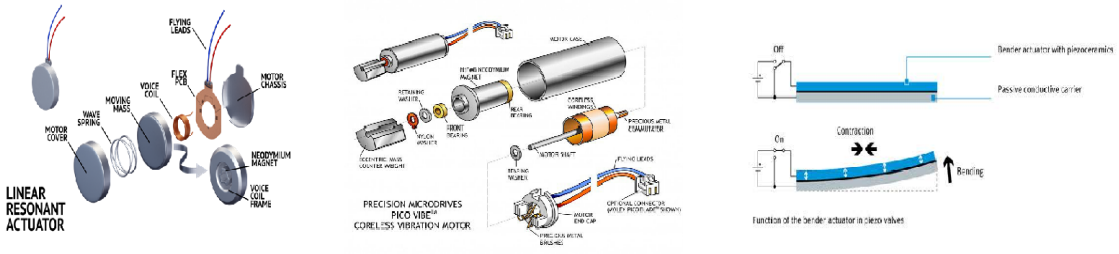

Figure 2.2: Left: Linear Resonant Actuator. Center: Eccentric Rotating Mass actuator. Right: Piezoelectric actuator.

These types of actuators have different fields of use but in the haptic suit ERMs are the most common. They are used a lot due to their low cost but due to their size and consumption it is being replaced by other technologies, while the Piezo are more precise and have a lower consumption but are fragile. However, the actuator alone is not enough to allow a device to provide haptic performance, but it is necessary to identify where the contact on the surface occurs, the concept of "haptic search", and whether in addition to tangential gestures there are orthogonal ones, thus evaluating the applied force.

2.2 Touch and Force detection

In the haptic context, the identification of the position of the touch is fundamental to ensure correct actuation and more realistic experience.
Furthermore, the use of touch interfaces allows the reduction of the mechanical parts enabling a cut in the manufacturing cost, reduction of false positive touch event risk and others benefits[3].
However, touch technology also has its downsides in base of the used one.
The detection of the touch could be done in different ways:

- **Resistive touch panel**: is composed of a glass screen and a film screen, separated by a small gap. Inside the gap, on each side, there is a metal foil that acts as an electrode. When the film screen is touched with a finger or any other type of object, it bends. During bending, the two electrodes connect, generating a current flow.
  Advantages of this technology are the low power consumption, low production costs and the water resistance. However, the external film is vulnerable to damages.

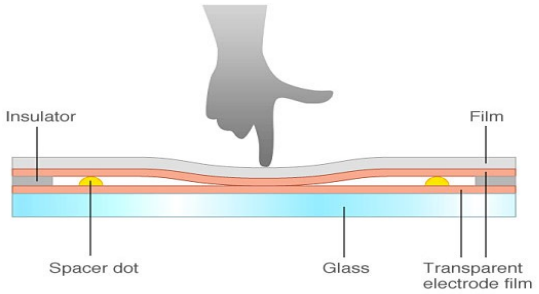

Figure 2.3: Resistive film touch pane[4].

- **Capacitive touch panels**: the point at which the touch occurs on these pannels is identified using sensors to sense minor changes in electrical current generated by contact or changes in electrostatic capacity.
  Unlike the resistive technology, the touch requires either a finger or a special stylus and feature strong resistance to dust, water drops and scratch resistance.
  Two types of touch panels use this method: surface capacitive touch panels and projective capacitive touch panels[4].

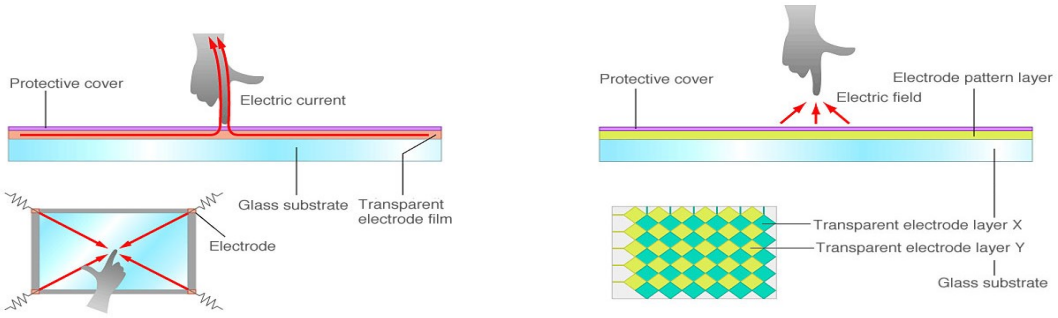

Figure 2.4: Left: Surface capacitance touch panel. Right: Projected capacitive touch panel[4].

- **Acoustic wave and Optical touch panels**: Surface Acoustic Wave (SAW) touch panels have been developed to achieve high levels of visibility. Internally, these panels consist of several piezoelectric transducers arranged in the corners of a glass that transmit ultrasounds on the surface in the form of elastic waves, which are received by transducers placed on the opposite side.
  When the screen is touched, ultrasound waves are attenuated by the object and the location is identified by detecting these changes[4].
  For what concern the Optical touch panels, their object detection exploits the use of infrared image sensors to sense position through triangulation generated by two infrared LED and a retro-reflective tape.

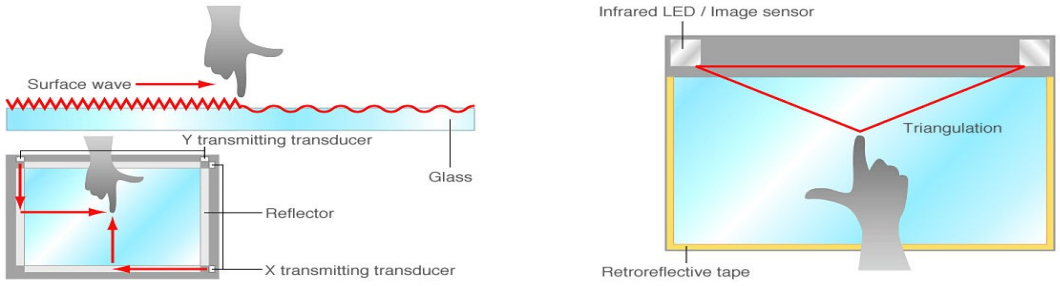

Figure 2.5: Left: Surface acoustic wave touch panel. Right: Optical touch panel[4].

All these types of technologies can detect the presence of a touch, who identifies only a finger presence and who any object. In addition, the various types of touch interfaces listed have their own application context defined by their advantages and disadvantages.
Nevertheless, focusing on the context of this project, the technology that best lends itself to this purpose is the capacitive one, since it allows the exclusive identification of the finger avoiding possible unexpected "touches" of other objects. In addition, capacitive surfaces are the most used not only in tablets and smartphones but also in

those devices that try to replace a physical button, as in the case of the automotive field.
However, a problem that persists is the "haptic confirmation". A flat surface with touch identification technology is only able to tell if the finger is present and where, by looking at the touch position along X-axis and Y-axis, but is not able to confirm if a pressure has occurred (orthogonal movement), if not by using some expedient. Furthermore, if this technology is used in environments where it is not always possible to look at the touch area, not having tactile feedback would cause distractions or wrong management.
Consequently, it is necessary to introduce a mechanism capable of detecting the force of pressure and one to generate a tactile feedback following this pressure.
For this reason, in addition to a touch detection mechanism there is a force detection device, leading the system to make three-dimensional evaluations with the introduction of the Z-axis.
Also for the detection of the force exist different types of technology:

- **Load cells**: Load cells converts a force such as tension, compression, pressure, or torque into an electrical signal that can be measured. If the force applied to the load cell increases, the electrical signal changes proportionally. Common type of load cell used is the strain gauges, that is highly accurate, versatile, and cost-effective. When a force is exerted on the strain gauges, they change shape by altering their resistance. This change in resistance can be measured as a voltage, which is proportional to the amount of force applied.

- **Force sensing resistors (FSR)**: FSR consists of a semi-conductive material or ink which is interposed between two substrates separated by a spacer. When a force is applied, a conductive film is deformed and presses against the ink. As more the contact is higher the resistance of the device decreases. Hence, the resistance drop is inversely proportional to the applied forces, so when zero force is applied the value of resistance is big (in the order of Mega ohms). The advantages of FSRs are they flexibility and thin dimension, the durability, have lower power consumption and are low in cost. However, they lack in their accuracy and repeatability, where repeated measurements can vary by 10% or more[5].

- **Force sensing capacitor**: This technology is made of a material whose capacitance changes when a force, pressure or mechanical stress is applied. Its structure includes two plates that simulate those of a capacitor, therefore reducing the distance between the two will also decrease the capacity, since they are directly proportional. Capacitive force sensors can provide improved sensitivity and repeatability compared to the FSR but it can easily be affected by noise.

The combination of position and force detection, with the consequential haptic actuation, allow the correct verification of the interaction with the device.

2.3 Niceclick

In the previous sections it was highlighted that for a correct haptic functionality, in addition to the identification of the position, it is also necessary to identify the applied force and actuate. As for the last two requests, unlike the devices listed in the previous sections, there is a device that is able to perform this double task, the Niceclick (NC-C1012-12V).
The Niceclick, developed by TRAMA Engineering, is a linear actuator with integrated force sensor designed for moving a touch surface with a predefined vibration. The user touch action on the surface is detected and measured by the integrated force sensor giving to a control system the needed information to trigger a specific haptic sensation. This product seeks to place itself in the automotive industry, as few such technologies exist. The few that do exist, require high voltages or are very expensive or age fast.
Regarding its composition, the Niceclick haptic actuator is an electromechanical device composed of three main mechanical parts: Stator, Elastic element, Cursor.

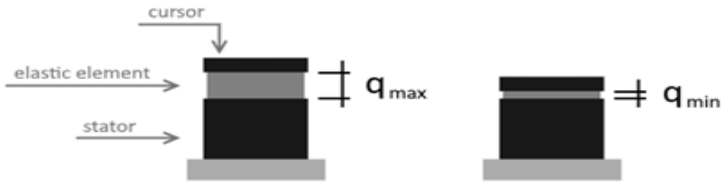

Figure 2.6: Niceclick composition.

The Niceclick actuator is a small cylinder which can change its vertical dimension when electrically powered. When an electrical current flows in it, the stator will attract the cursor compressing the elastic element, decreasing its vertical dimension. When the electrical current stops, the elastic element will pull the cursor up until the device will come back to the original dimension.
The standard installation of a Niceclick device is to attach one side (stator or cursor) to the haptic surface and the other side to the haptic device base or to a mass. When the device is powered, it will change its dimension so moving the haptic surface towards the base or the mass. The displacement of the surface, i.e. vibration, will be recognized by a human finger as an haptic effect. It is almost imperceptible to the naked eye, but very effective to the tact.
In order to identify the force and generate the feedback, Niceclick must have a special support circuit with a microcontroller that manages the application part by controlling the current flow and inductance variation. More details regarding the control circuit and the technology itself will be defined in the section 3.
In conclusion, this technology allows to avoid the control of multiple devices by integrating everything into this single component, facilitating management and the control.

Furthermore, one of the goals of this thesis, carried out in the company that holds the patent for this technology, was to verify the software library and expanding it with new features by placing Niceclick at the center of the project. In the following chapters will show how this device performs its work as a haptic actuator making an object interactivity and perceptible.

Chapter 3

Niceclick technology

As mentioned in the previous chapter the Niceclick thanks to the environment around it, hardware and software, is also able to measure the the force in addition to the normal actuation functions. This can be done because the actuator, which is fixed to the movable touch surface, is subjected to the displacement caused by the pressure of the user.
Based on this information and on the actuator model, the company has developed a special circuit able to generate and obtain all the information necessary to achieve this purpose. In the same way, software bases were defined for which this information could be correctly processed and monitored.
The hardware was developed basing on the idea that varying the value of the inductance also the energy stored in the solenoid changes. The circuit must be able to convert this energy variation into a varying signal that can be easily analysed by a microcontroller.
Therefore, the fundamental circuits that allow the generation of the impulses and the force measurement are reported below, pouring the main peculiarities of a haptic system into a single element.

3.1 Actuation and Charging circuit

This part of the circuit is the same used for the haptic feedback generation and charging mechanism for force measurement task, that works at different moments. This double role is obtained by driving the circuit's current with a PWM, generated by a microcontroller, changing its duty-cycle (DC) value. In this way is possible to manage the force exercised by the actuator, the inductor in the circuit, or stimulate a specific second order response for measuring purpose.
The first step to observe inductance variation is the excitation of the solenoid present in the Niceclick. By letting a current flowing into the solenoid, a part of energy is stored using a capacitor in parallel (is present in the discharging/measuring circuit 3.3). This operation cannot be too strong, because the current must be considerably small to avoid that the magnetic force created is not able to move the armature of the actuator causing a continuous feedback on the touch surface.
The circuit is composed by a low side n-channel MOSFET connected to ground and driven by the MCU via PWM, though an RC network.

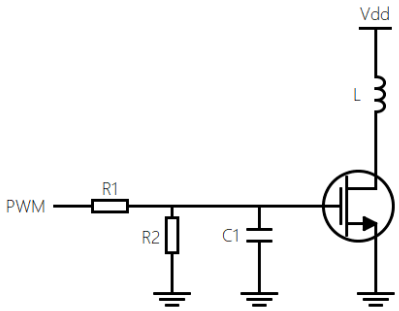

Figure 3.1: Actuation/Charging circuit.

In the figure R1 is used to avoid switching loss since the MOSFET does not turn off quickly due to parasitic oscillations caused by gate capacitance and the inductance. R2 instead is used to reduce the gate-source voltage to 0V. The Capacitor C1 is connected between the gate and ground to improve the turning on of the n-MOS.
In support of the previous statements, from the Faraday's law of induction is possible to see that a current flowing into an inductor generates a voltage across it:

$$V_L = -\frac{d\Phi_B}{dt}$$

where the magnetic flux is equal to $\Phi_B = Li$ and according to this, the equation can be rewritten as:

$$V_L = -L\frac{di_L}{dt}$$

This means that the voltage generated on the inductor is directly proportional to the change rate of the current and to the inductance value.

By stimulating a circuit consisting mainly of inductance, its parasitic resistance and the external capacitor, what is obtained is a second order transient like the one shown below.

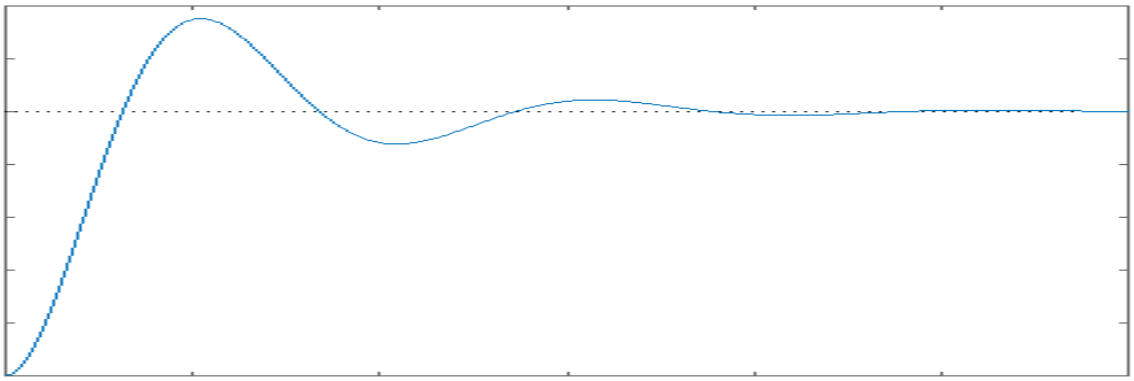

Figure 3.2: Second order transitory.

When the value of the inductance varies, the charging and discharging time can increase or decrease as well as the voltage peak.
By keeping the current value constant, therefore continuing to generate a PWM with the same duty-cycle, the variation obtained when a force is applied is the variation of the time constant, i.e. the discharge time is lower and the voltage peak higher.
This happens because the time constant depends on the value of the inductance itself.

$$\Delta L \rightarrow \Delta \tau \rightarrow \Delta i_L$$

The force measurement and identification procedure is carried out by another circuit that makes evaluations related to the falling time of the transient, in a situation with and without the force.

3.2 Measurement acquisition circuit

The main part of measuring mechanism is managed by the following circuit.

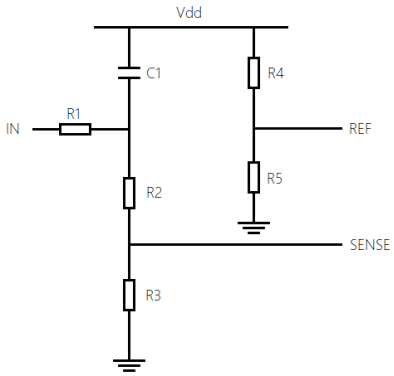

Figure 3.3: Measurement acquisition circuit.

It is based on the use of a comparator, with inputs the transient of the RLC circuit and a reference voltage, which generates a signal every time the set threshold is exceeded.
In the circuit presented in the image, the comparator is not present since it is included within the selected microcontroller.
The circuit is composed by a resistive network and a capacitor, in parallel with the inductor present in the actuator and connected to the supply voltage. The purpose of the capacitor is to be charged and discharged to obtain the desired signal for the correct comparison. The reference voltage is obtained through a voltage divider on the same supply voltage of the actuator. In the same way also the input signal, coming from the discharge of the capacitor, is rescaled via a voltage divider bringing it within a range of values that can be supported by the comparator and by the microcontroller.
The input signal is connected to the circuit in figure 3.1 to the drain of the nMOS and when the discharging phase starts the nMOS is opened and the inductor due to its physical properties tries to maintain the current constant. According to its constitutive equation the voltage rises impulsively in the same direction of the current and for this reason the inductor can be represented as a voltage generator in series with the supply voltage, obtaining a voltage higher than the supply one.
When the inductance changes, what is changing in the circuit is the time constant. Varying the time constant also varies the discharging current as show red curve in the graph.

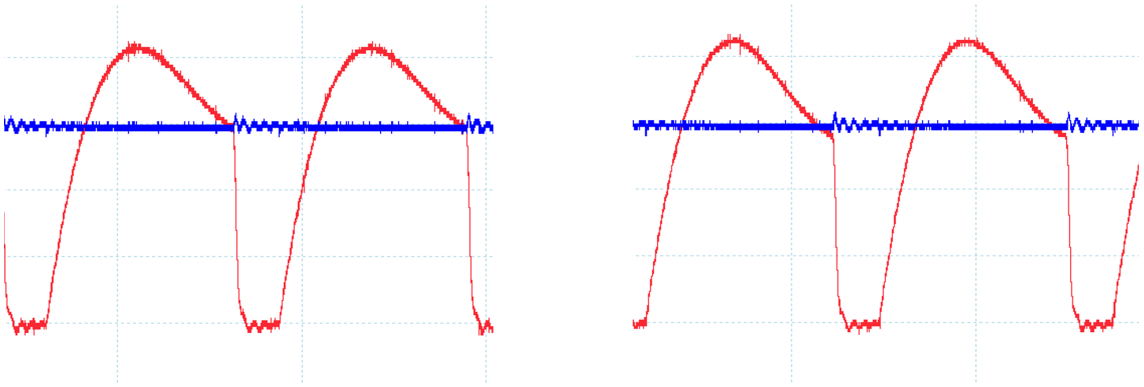

Figure 3.4: Variation of the curve at the press on the surface. Left: Not pressed. Right: Pressed.

The internal comparator is set as non-inverting, where on the negative input it has the constant reference threshold, blue line in the 3.4 image, while on the positive one the "sense" signal is connected which varies according to the pressure applied to the surface, red line in the 3.4 image.
When the signal exceeds the threshold the comparator output will become high while when it is under the threshold the output becomes low. By applying this logic to the curve shown above, we will have that in the charging phase as soon as the threshold is exceeded, the comparator will pass from low to high and will remain high until the discharge period brings the voltage back to a value lower than the threshold.
If the discharge time varies, the identification of the falling edge changes and therefore the square wave output from the comparator may have a longer or shorter period based on the applied force. At the firmware level, appropriate interrupts are set to sample the comparator output and measure the signal period in order to capture the difference between the case of non-pressure and the one under pressure. Then difference between two signals is translated from the time domain to a domain established by the ticks of a software counter.

3.3 Flyback diode circuit

In this section is exploited the use of the flyback diode and its circuitry.
Normally this kind of diode is used in parallel to an inductive load to avoid the return of EFM voltage generated by the inductor when the current rapidly decrease/increase. When this happens the diode became forward-biased and so start conducting and allow the current to recirculate avoiding that the voltage on the nMOS, rise too much causing damages on the device.
However, if this diode is present all the current will flow in it and so the measuring circuit will not be able to receive any information. For this reason a more complex circuit is used instead a single diode. This circuit is used only when it is not necessary to measure the applied force and when the actuator must provide an haptic feedback.

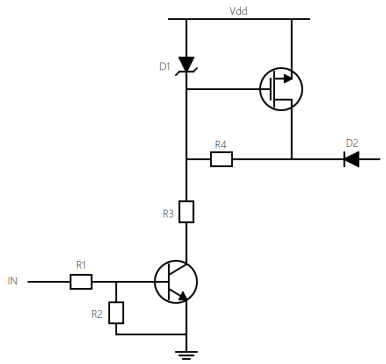

Figure 3.5: Flyback diode circuit.

The npn transistor is driven by a microcontroller acting on its base voltage. If the driving signal set a low logic level, the npn is off and no current will flow in the collector resistance. When the inductor voltage V_L rise above a threshold defined by the voltages on the two diodes and those on the resistance R4, the MOSFET is "on" and a connection between V_{dd} and the diode is established such as in the normal flyback circuit schematic, forcing the inductance current to flow in this part of the circuit avoiding that the driving nMOS and the measuring circuit can be damaged. On the other side, when the input signal is settled to a high logic level the transistor is "on". The Schottky diode became forward biased fixing a negative V_{GS} that put the nMOS in the interdiction zone.

3.4 Microcontroller

The microcontroller selected to control the board is Cypress's CY8C4247AZI-L485, consisting of a single core, a Cortex M0.
This processor is constitute by the PSoC 4 architecture that is a combination of a microcontroller with digital programmable logic, programmable analog, programmable interconnect, secure expansion of memory off-chip, high-performance ADC, opamps with comparator mode, and standard communication and timing peripherals. The programmable analog and digital subsystems allow flexibility and in-field tuning of the design[6].
Below are listed some of the main characteristic of this device:

- 32-bit MCU subsystem with 48 MHz ARM Cortex-M0 CPU, Up to 256 kB of flash, Up to 32 kB of SRAM and DMA engine with 32 channels;
- Programmable analog peripherals such as four opamps, four DAC and two comparator;
- Programmable digital blocks;
- Capacitive Sensing;
- Serial Communication including four reconfigurable serial communication blocks (I2C, SPI, or UART), USB, two independent CAN blocks;
- Timing and PWM with Eight 16-bit timer/counter pulse-width modulator blocks;
- Up to 98 programmable GPIOs;
- Low power 1.71 V to 5.5 V operation.

When this board was developed, the goal was to create a driver for the Niceclick, but also to be a device in line with automotive standards. For these reasons the choice fell on this microcontroller, it is designed for the management of capacitive elements, which can be used as devices able to identify the position of the finger and consequently actuate the haptic effects when necessary (the PCB has a connector for this purpose). It has an appreciable calculation speed and has all the peripherals necessary for the operation of the various control tasks, especially the presence of internal comparators allows the simplification of the physical circuit.
In addition, Cypress's processors feature all the suitable elements for automotive applications and are tested against those standards.

3.5 Final circuit

The final circuit on which I worked is the combination of those illustrated but replicated four times, in order to control and measure four actuators.
Each signal is properly connected to the microcontroller, the CY8C4247AZI-L485. Therefore, a typical management in the measurement phase consists in generating a PWM with a certain DC, while the generated square wave is in the “high” state the received signal is null, as it is cut-off by the flyback system, and as soon as it becomes "low" the response from the actuator is obtained (as one in the 3.2 figure). The received curve will be monitored by the internal comparators of the microcontroller which will identify the two instants in which the threshold is reached, rising and falling, allowing to identify possible differences in case of applied force.
On the other hand, the haptic effect is generated, the sampling system will not be active but a PWM with different DCs will be generated based on the desired effect, i.e. the actuators will be stimulated with currents of different intensity making them generate pulses with different intensities.
Finally, a fundamental part is the connection with the outside world, established by the presence of a CAN Bus network, receiving and transmitting the necessary information for the software application.

Figure 3.6: Niceclick Evaluation Board.

Chapter 4

CY8C4247AZI-L485 Firmware: Haptic Management

This chapter explains the functions and operations performed by the CY8C4247AZI-L485 microcontroller inserted in the board illustrated in the chapter 3.
The work done by this microcontroller is to manage the haptic part by carrying out force measurements and managing the activation of the actuators. Furthermore, it allows the reception and transmission of information through the CAN bus. This last point is fundamental because this board must be considered as a "slave" receiving instructions and commands from an external device and notifying any changes in the state, as for an ECU inside a car. This allows an easy integration in a wider and complex system, independently from the "master" that controls it, it is only necessary to respect the defined protocol for data exchange. In the case of this project, the master that sends the instructions is the processor that manages the graphic part and the control of the touch screen display.
A fundamental aspect required of an embedded device, which performs accurate measurements and operations, is time management. A real time system (RTS) ensures that all operations are performed at the right time and in the right order, also simplifying the debugging phase. In this case, this management is possible thanks to the use of Timer Events that are evoked when a certain time has elapsed. In order to perform these operations an internal timer has been used, with the working frequency of 48 MHz, and an interrupt is triggered when the count of the timer becomes equal to a certain compare value.
In this application two events are created:

- 1 ms Event: this event is the trigger for the haptic tasks that takes care about the management of the different output feedbacks and the force recognition.
- 3 ms Event: this event is the trigger for the application task that permit the

decoding and management of the information received through the CAN Bus.

This timing management allows to synchronize each task and be sure that the order of execution is always the same.
Below is possible to see the code organization already mentioned.

Listing 4.1: Main loop.

```
1   for(;;)
2   {
3       Board_DisableInterrupts();
4       event1ms  = TMR_GetEventState(timer1ms);
5       event3ms  = TMR_GetEventState(timer3ms);
6       Board_EnableInterrupts();
7
8       if(event1ms)
9       {
10          NC_Task();
11          NC_Status();
12      }
13      if(event3ms)
14      {
15          ApplicationTask();
16      }
17  }
```

Since some of the APIs of the “NC_Task” and the “ApplicationTask” itself exploit the use of a Fine State Machine as structure, each time that an event is evoked the states of the FSMs are updated. In this way in every moment, knowing the inputs and the timer output, it is possible to predict the state of each task.
In the next pages the most important tasks and functions would be analyzed.

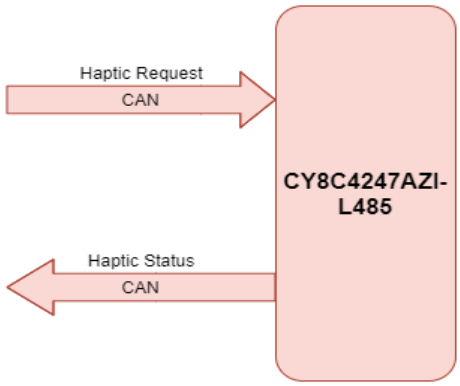

Figure 4.1: CY8C4247AZI-L485 Global interface.

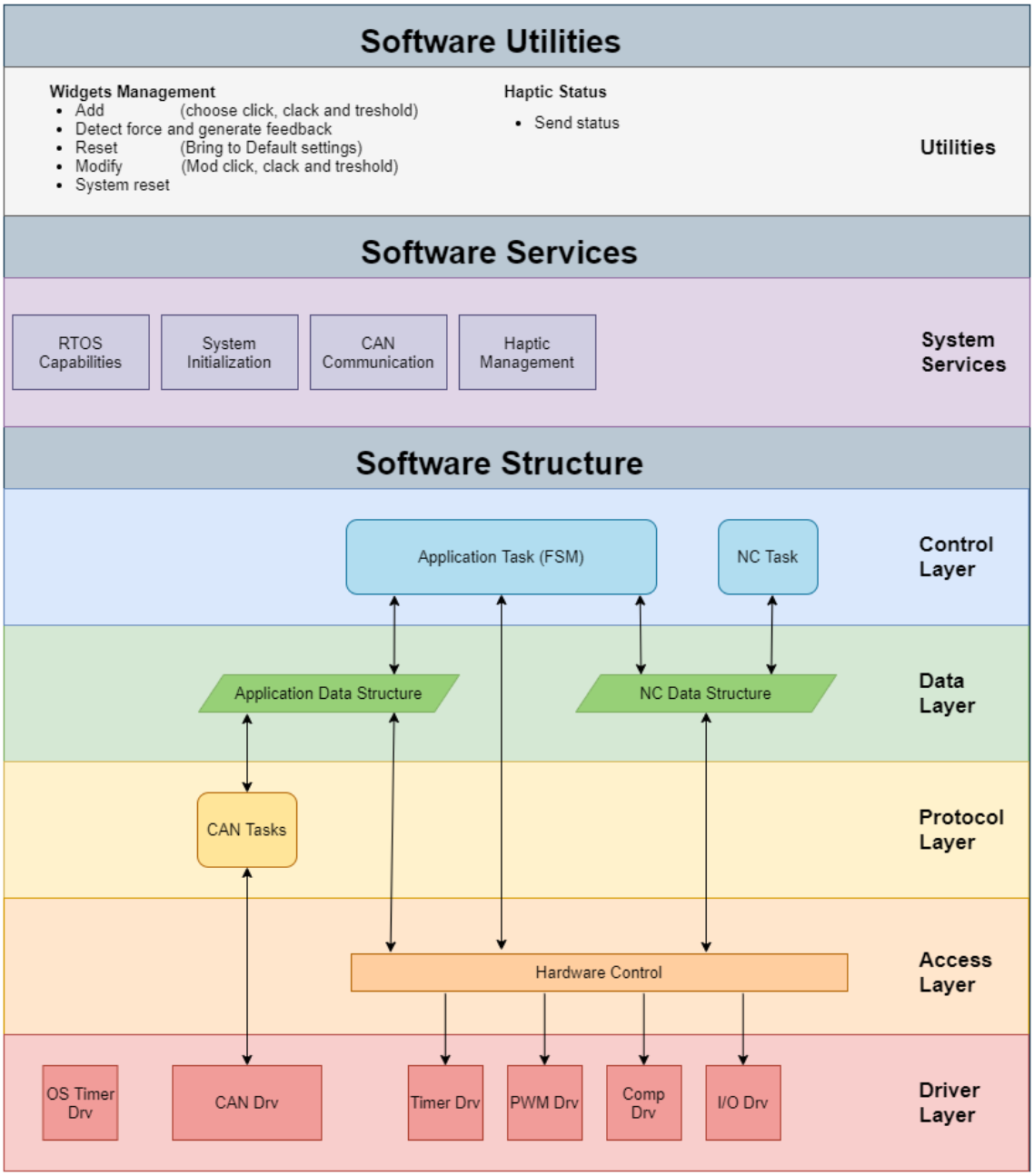

Figure 4.2: Software system structure for CY8C4247AZI-L485.

4.1 Widgets and haptic effects definition

The entire application is based on the concept of widget. In previous projects the widget was always associated to a specific position on a surface, identified for example by a capacitive membrane. This position was associated with a haptic effect, activated when the finger was detected or if a force threshold was exceeded. In particular, the typical application target of the technology was to emulate a button, thus associating to it an effect on pressure and eventually on release.
Therefore, the composition of haptic widgets consists of three basic components:

- Force threshold for actuation
- Effect associated with press
- Effect associated with release

The various widgets, made up of their ternary structure, are saved inside an array where the position index is also the ID of the widget itself which is then used by the various functions.
The threshold if set to 0 will mean that the activation of the haptic effect will be immediate, if the applied force is checked. While the associated effects are identified by means of a numerical value that refers to the library of effects present in memory. The release effect may also not be present.
As will be explained later, in this project the concept of widget is extended over multiple fields and not only for button emulation, as well as the non-static association of a position. Nevertheless, its definition and structure is preserved thanks to the introduction of the 0 threshold and the non-compulsoriness of the release effect, mentioned above.

Regarding the generation of a haptic effect, it is associated with the action of moving the haptic surface in a predefined way. The Niceclick actuator is able to deliver to the haptic surface a level of force depending on the current flowing in it and on the distance between stator and cursor. Different haptic effects can be generated by modulating the current, therefore modulating the force applied to the haptic surface.
The haptic effect is composed of a succession of pulsations, generated by the actuators, at defined time intervals and with intensities that can vary between one and the other. The intensity is varied by acting on the duty cycle of the PWMs that actuate the Niceclicks, for this reason the effect, at firmware level, is represented by an array of values identifying the various DC of the pulsations to be generated (values between 0 and 200, i.e. 0% and 100%).
The maximum number of pulses is 100 but the array in question has a size of 105 cells (105 Bytes). The five additional parameters are additional values that allow to manage some aspect of the effect.
In particular they are:

- Total number of used samples

- Time between two samples
- Number of repetition of the haptic effect
- Time between two repetitions
- Special identifier

The first was inserted to avoid loading 100 samples each time but to load only the useful values, while the second and fourth parameters are numerical values that are multiplied to a time constant to make them comparable with the application.
The last parameter is a special value that is used only for some effects, allowing to not generate a last pulsation used to reset the magnetic poles of the Niceclick.
Usually to allow the reading of the release gesture and do it in cooperation with the baseline calculation, the magnetic poles of the actuator must be reset by generating a pulse at maximum power, but this sometimes brings an undesired perception of the effect. If this parameter is active it will ensure that this last pulsation is not generated and it will do only if the effect on the release is not present, in order to preserve the stability of the system.
Below there is an array representing a haptic effect.

Listing 4.2: Haptic effect array.

```
const uint8_t defaultPulse1[] =
{
    0x64, 0x64, 0x64, 0x64, 0x64, 0x64, 0x64, 0x64, 0x64, 0x64, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x64, 0x64, 0x64, 0x64,
    0x64, 0x64, 0x64, 0x64, 0x64, 0x64, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
    0x00, 0x00, 0x00, 0x00, 0x1E, 0x01, 0x00, 0x00, 0x00
};
```

Looking at the acceleration profile generated by this haptic effect, placing an accelerometer on the surface of the display, is possible to see similarities to the standard acceleration curve of a keyboard, except for variations due to the different type of material.

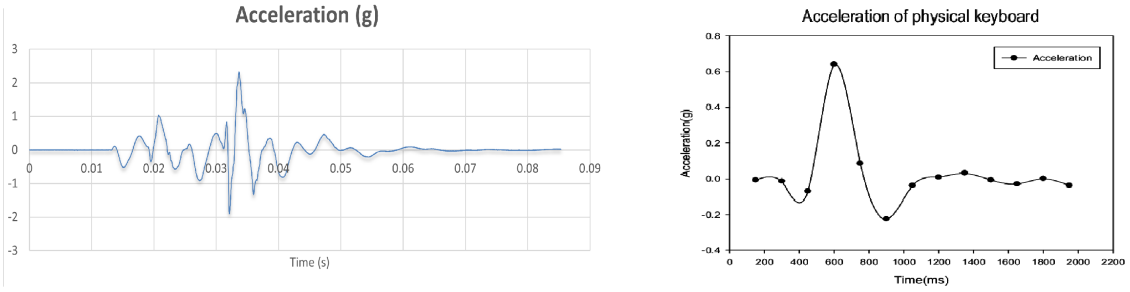

Figure 4.3: Acceleration profiles. Left: Niceclick. Right: Keyboard.

In this way, all the desired effects could be generated and analyzed.

4.2 Haptic Tasks

Below are illustrated the three tasks that allow the control and management of the haptic part of the system.

- NC_Task: management task
- NC_Status: status task
- NC_Init: initialization API

The first two task are performed every ms, which is the optimal time for which information can be managed correctly. In particular, the “NC_Task” allows the generation of haptic feedback and at the same time carries out the measurements on possible changes in force, controlling the peripherals and the external circuitry in the appropriate way. While the status task allows the identification of any changes in the status of a widget, pressed or released.
The initialization task is performed at the beginning of the program to load all configuration parameters and initialize the required devices.
These tasks must collaborate with an application task that gives the appropriate commands for their operation.

4.2.1 Niceclick Management Task

This task, called "NC_Task", is responsible for the correct functioning of the entire haptic system, as at each period T (set to 1 ms) it carries out a sequence of fundamental operations by updating the internal data structure of the system by actuating, sampling force and update the baseline.
The task is divided into three parts, which correspond to three APIs called in sequence:

- Measurement of the Niceclick status
- Calculation of the applied force
- Pulse generation

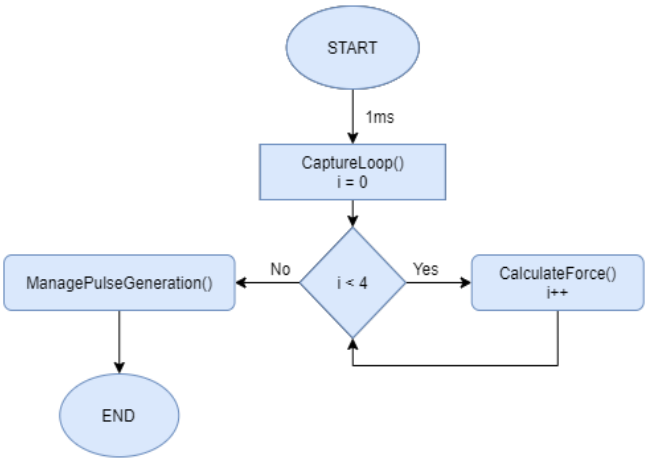

Figure 4.4: Haptic Task.

4.2.1.1 Measuring API

This API oversees to the managing of the measuring system in order to obtain information about the status of the Niceclick. Managing the measuring systems do not means only read the information but also provide the correct signals to the circuit.
In order to work properly this function need that flyback diode is "off", in this way the current, and so the information, contained in the inductor is no more pushed into the diode but can flow in the measuring circuit. This operation is performed when we move from pulse generation phase to measuring phase.
The second step is to set properly the signals that drives the four transistors that control the current flowing in the inductor (see 3.1) in order to stimulate the Niceclick to reply with the expected curve, like the one in 3.2 image.
The signals are created with the use of PWM, in the configuration shown in the figure 4.5.
In the measuring phase every PWM module is settled at the frequency of 16 kHz (which is the minimum frequency value for which the vibration cannot be perceived by the human ear) and the duty cycle to 20% (correspondent to 12.5 µs) with the bus frequency of 32221kHz.
When the stimulus is generated, the response must be measured and this is made possible through an input capture timer on the internal comparator, which compare the two values coming from the circuit 3.3, i.e. the reference voltage and the sense voltage. The peripherals are set as in image 4.6.
The capture has been settled to trigger on rising edge of the input signal and when this happens an interrupt is triggered and ISR called.
The measure is performed by the ISR that each 62.5 µs compute a new data. Since the trigger is set to sense the rising edge, it means that the response to the stimulus is growing and this value is used as a reference and the capture is set with falling

edge as trigger. In this way when the signal goes down the routine is called again and compute the difference between the new value of the counter and the old one, getting the total amount of ticks between the two instants. The result is saved inside a queue.
Since the PWM frequency is set to 16 kHz means that every 1 ms the microcontroller receives 16 data from each stage and so each queue must have at least this number of storing locations.
Once all the samples are stored the program enters in a “for” statement that read data from queue and sum all of them and at the end divide the result obtaining a single value called “raw value”.
This procedure is done for all Niceclicks present in the system one a time and cyclically.

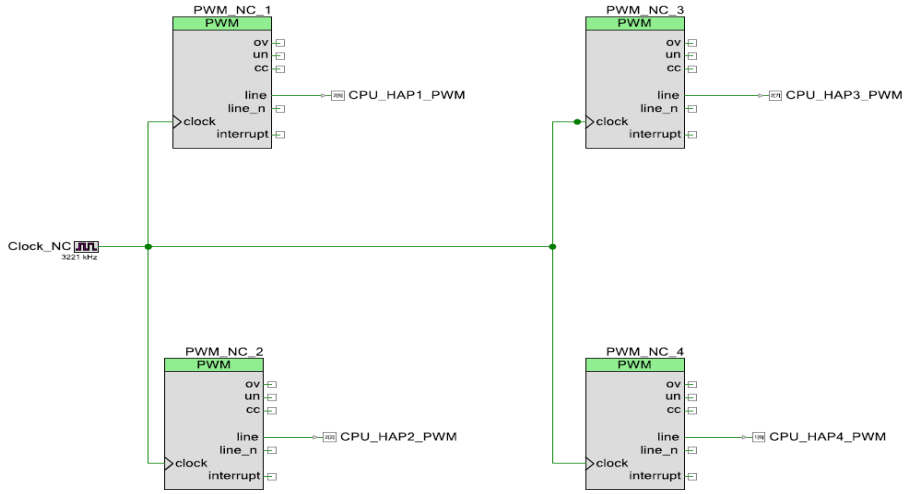

Figure 4.5: PWM peripherals diagram.

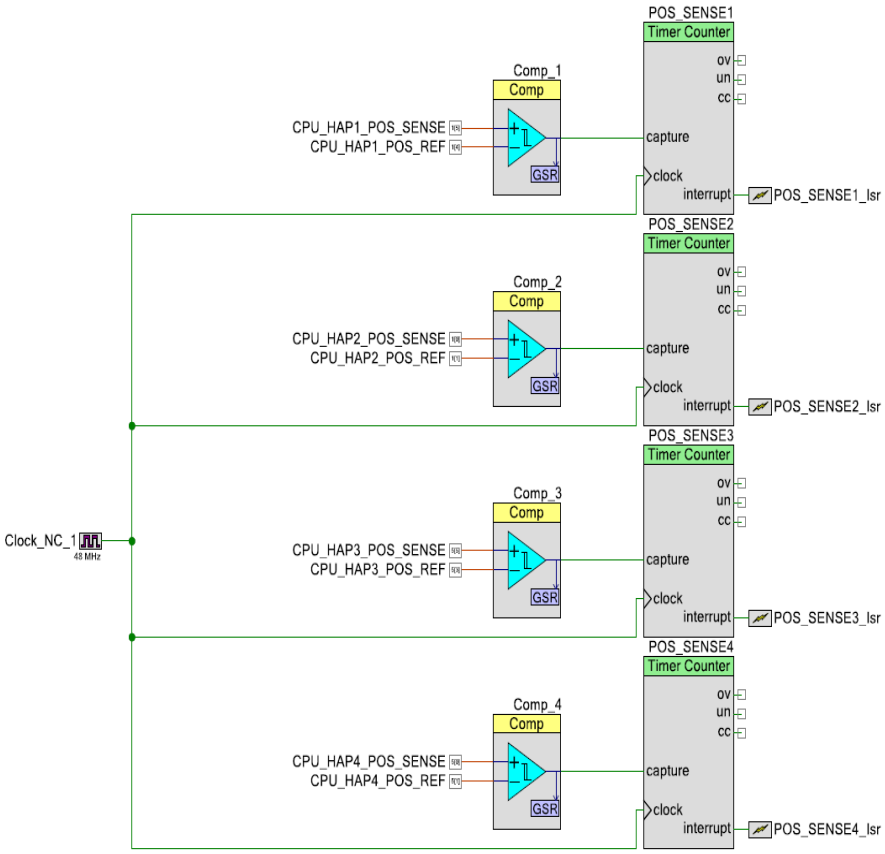

Figure 4.6: Capture timers diagram.

4.2.1.2 Force calculation API

While the measurement API provides all the information and data from the sensing circuit without doing any manipulation of them, this function is the core of the force recognition algorithm doing the right conversion and manipulation.
It receives as input, from previous function, the raw value and produce as output the force and the baseline.
The baseline is a signal that help the microcontroller in the computation of the applied force. This need is since the raw signal can have a drift due to the temperature and hysteresis dependency and so for this reason it is not possible to simply see the raw behavior. In fact, the signal may rise slowly due to a temperature drift without any force applied and in this case the system may interpret it as a slow applied force comparing it to a fixed value, but this is not correct. So, for this reason a moving threshold that follow the signals is needed.
The baseline is also used for the force computation. Once the baseline has been calculated, if its value is lower than the raw value a value called delta is computed with the difference between the raw value and the baseline. Otherwise, if it is greater

the delta will be set equal to 0.

$$\delta = raw\_value - baseline$$

The baseline and delta are calculated individually for each Niceclick used. The force applied is instead calculated considering all the Niceclick, using the following formula:

$$F = F + \frac{\delta * G[i]}{100}$$

Where G is the gain and is established, individually for each Niceclick, in the initialization phase of a widget. This formula is applied 4 times (number of Niceclick) by increasing i by 1 every time.
Subsequently this value is used to check if there is an exceeding of the force threshold leading to the actuation of a haptic feedback.

4.2.1.3 Pulse generation API

A haptic effect, in this application, consists of a first effect typically generated upon recognition of a pressure and followed by another effect generated upon release. The latter may also not be present since not all actions involve a press and release or the sequence of one effect is sufficient, like a simple vibration.
This function bases its work on an FSM that manages the control logic and an ISR that allows the impulses to be actuated.
The generation of haptic feedback, as the code has been structured, needs an explicit request. The request can be made without checking the exceeding of a force threshold or by controlling it.
This procedure is carried out through two APIs, which will be illustrated in the 4.3.2 section, where a variable is set allowing the FSM to proceed with its work.
The decision to wait for an explicit request was made because it is necessary to say which widget we are referring to and also because the analyzes carried out every ms on the force require that the measurement circuit be active while for actuation it must be "closed".
In the first state of the FSM, once the execution request is confirmed, the Niceclick response measurement task will be disabled by enabling the flyback diodes, for the reasons defined in the 3.3 section, and by enabling a timer with the aim of generating at a predetermined interval each pulsation component the haptic effect.
The timer is configured as follows:

Clock	10kHz
Period (T)	500µs
Dimension	16bit
Interrupt	on Terminal Count (TC)

Table 4.1: Pulse timer setup.

The period may change depending on the desired effect.
Finally, the configuration for the haptic effect of press gesture for that widget is loaded.
Following the set-up phase, is waited that all pulses have been generated or that the time-out time has been reached before moving on to the next state.
As mentioned at the beginning of the section, the generation of the pulsations that make up the haptic effect is managed by an ISR activated every period T (500µs standard), for a number of times equal to the number of samples present, which is one of the extra parameters of the haptic effect array (see 4.1), and modifies the DC of PWMs, according to the sample, creating the haptic effect. Once the total number of samples has been reached, the ISR checks whether the feedback to be generated is composed of several repetitions of these samples (for example in the case of a vibration), if yes it resets the parameters and starts again until it ends otherwise notifies the system the conclusion of the operating cycle by disabling the flyback diodes and setting back the DC of PWMs to 20% as required for the force calculations.
At this point, the management of the pulsations may vary in base of the presence of release effect. If the effect on the release is not present, the actuation procedure ends here otherwise it proceeds with the new state of FSM.
To verify the presence of the release effect, the same procedure carried out by the API which verifies whether the force threshold of pressure is overpassed, but in the opposite way, is used and then actuate.
Since the measurement system has been re-enabled, a new force values is had and being under pressure the force value obtained will be higher than the expected threshold. The release is identified by the return of the force value below the threshold.
The graph below shows how the force grows with the pressure, initially exceeding the threshold, which the system will perceives as a "touch" and then begins to decrease with the release, arriving again below the threshold, thus activating the second feedback, "release".

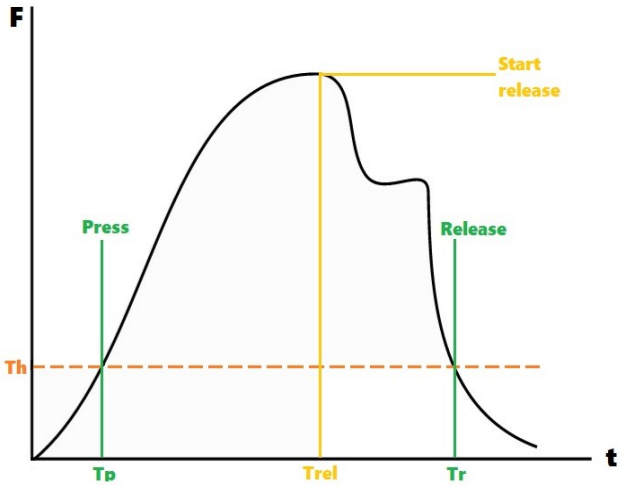

Figure 4.7: Force graph and relative threshold.

If the release action starts but with not enough impact, this mechanism allows to identify the system as still under pressure and avoid unwanted activation.
Furthermore, this graph is also useful for showing that if a pressure is not high enough to exceed the threshold, the effect will not be generated. The value of this threshold can be configured making the system flexible and configurable for the most varied cases.

Once the presence of the release has been identified, the measurement procedure will be suspended again and the haptic effect will be loaded, which may differ from that of pressure, and then the management will be passed to the ISR and proceed as already described.

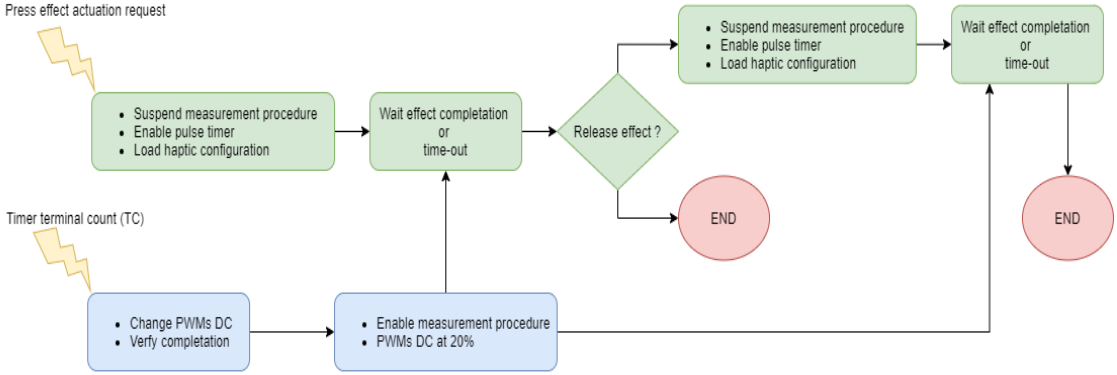

Figure 4.8: Pulse generation procedure.

4.2.2 Niceclick Status Task

The following task was introduced to store the status changes of the activated widgets. The execution of this task must follow the NC_Task so that it can immediately identify the changes and store them.
There are two possible states of change, pressed or released.
The first is notified as soon as the force threshold is exceeded and the execution of the haptic feedback is at the begin, while the second is notified when the force return above the threshold and before the haptic feedback is generated.
The second case may not even present, it depends on how the widget has been set up.
The storage mechanism consists of a FIFO where at each change a data structure is saved, which has the widget identifier and its status as parameters.
The use of a FIFO was decided because the sending out of the state is performed by the last state of application FSM (4.3). In the worst case the sending will take 9ms from when the change of state is detected, the detection takes place every 1ms, so that if there is a change in the meantime the new state is saved without overwriting the previous one.
In this application is talked about the state pressed and released but in reality what is communicated is the beginning of the generation of an effect. If the widget provides the sequence of two effects, which can be associated to the actions of press and release, what is notified is the activation of the press effect followed by the activation of the release effect. In the case of a single effect, not necessarily associated with a pressure, only its activation will be notified.

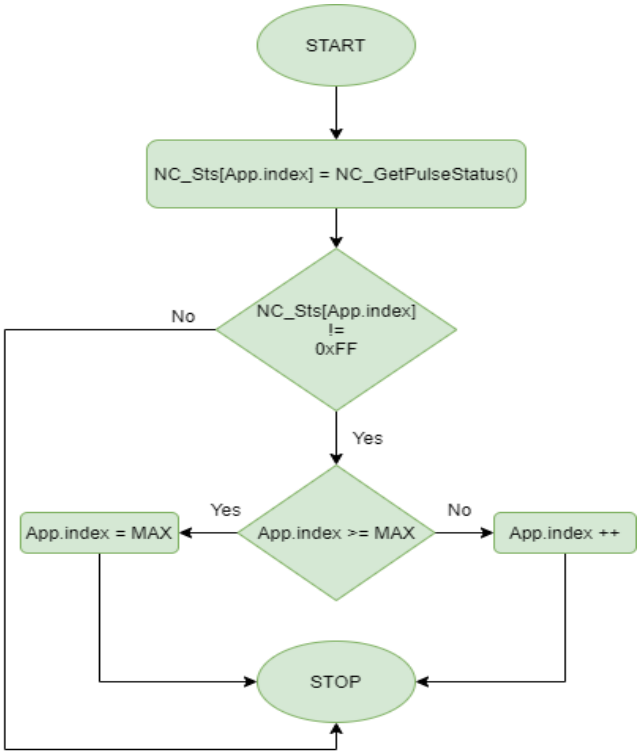

Figure 4.9: Niceclick Status task.

4.2.3 Niceclick Init API

This function, called on boot, initializes the entire library.
It initializes the peripherals, loads the default parameters, loads the haptic effects and starts the measurement procedure.
In particular, the function starts by setting all the elements making up the system data structure to 0 and initializing the PWMs with the duty cycle at 20%.
Subsequently, the flyback diodes are disabled, allowing the measurement circuit to work, the input capture in the rising edge form are also set to capture the first rising edge of the internal comparators.
Finally, another function performed by this API is the loading of the supported haptic effects library, defined as array of values as explained in the section 4.1, into the data structure. The haptic effects are defined in advance and saved in the memory of the microcontroller, not allowing the final user to create his own effect but only select among one of the supported ones.
At this point the system that manages the haptic part is completely initialized.

4.3 Application Task

The following task represents the application part of the system. It allows to receive the necessary information, to decode them and manage the haptic part, notifying the status of the system externally.
The logic of this task is managed by a four-state FSM:

- BUSY: initial wait.
- RECEPTION: reception and interpretation of external commands.
- PROCESSING: execution of the received commands.
- TRANSMISSION: external notification on the status of haptic feedback.

The transition from one state to another occurs sequentially every 3 ms, except for the BUSY state which is entered only once at the beginning. So, a full turn of the FSM takes at least 9 ms.
The work carried out by this task does not allow to act directly on the haptic part but updates some variables or parameters that will be implemented only when the "NC_Task" (see 4.2.1) has been executed.
The haptic tasks, executed every ms, are always applied three times between two states of the application task FSM allowing a correct interpretation of the new parameters and scrolling between the states of their internal FSMs.
The latter are the reasons why it was decided to use this time between two states, as 1 or 2 ms is not enough to guaranteed sufficient accuracy in the interpretation of the data, as they are too fast.
The various states that make up the task are explained in more detail below.

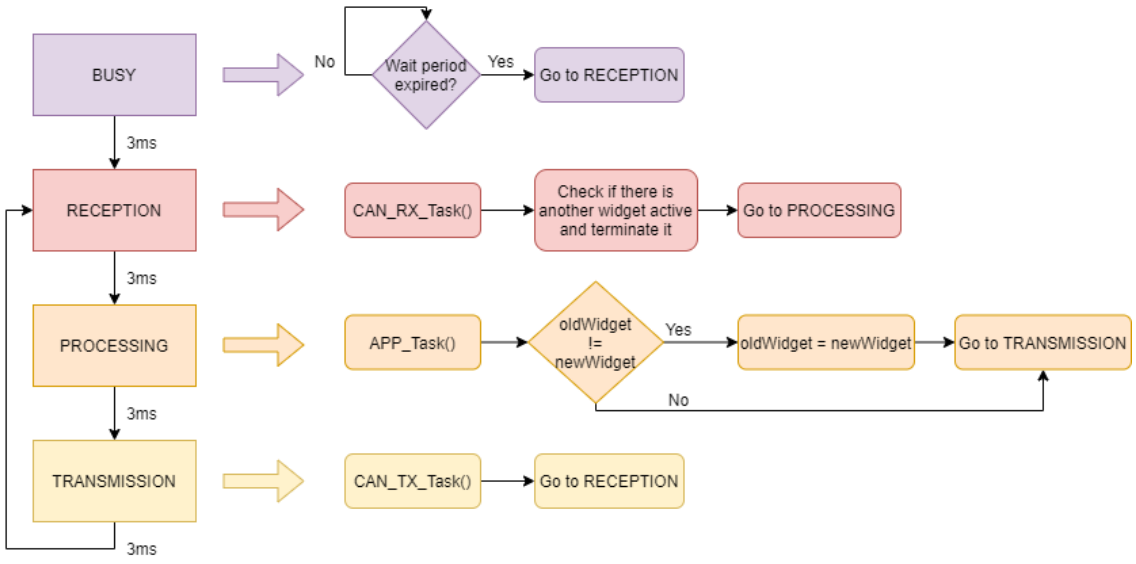

Figure 4.10: Application Task FSM structure.

4.3.1 Reception State

In this state of the FSM the reception of the CAN messages received is managed.
Using a CAN driver, created by the company specifically for Cypress processors, it is possible to check if there is a message inside a software queue.
The actual reception is handled by an interrupt which reads the message from the bus, if it has the expected ID, and copies it from the mailbox to the software queue.
Once the presence of the message is established, it is decoded and the various data bytes are assigned to the correct variables used by the application.
The decoding is based on recognizing the type of request and based on it a different DLC could be had since the bytes needed vary according to the message received.
The request identifier is always the first data byte of the CAN message followed by the data necessary to execute this request.
Below there is a table containing all the necessary information.

Request	DLC	Code	Data
WDT_CONFIG	4	0x00	Threshold, Click type, Clack type
WDT_TOUCH	3	0x01	Widget ID, Force detection status
WDT_START	2	0x02	Widget ID
WDT_RESET	2	0x03	Widget ID
WDT_MOD	5	0x04	Widget ID, Threshold, Click type, Clack type
SYS_RESET	3	0xFF	0xFF, 0xFF

Table 4.2: CAN valid message configuration.

The requests shown correspond to specific operations of which I report a brief description, following the order of the table 4.2:

- Adding a new widget.
- Enable or Disable for a specific widget the haptic procedure (force detection and feedback generation).
- Actuate immediately the haptic feedback of a specific widget.
- Reset the widget to a default configuration.
- Modify the haptic parameters (threshold, press and release effect) of a widget.
- Reset the entire system (reboot).

The last request listed has been introduced to reset the entire system when is necessary, simply by sending a CAN message, without directly acting on the power supply of the board by forcing its reset.
Furthermore, this operation is the only one that is performed in this state, the others update only the variables but are performed in the next state of the FSM.

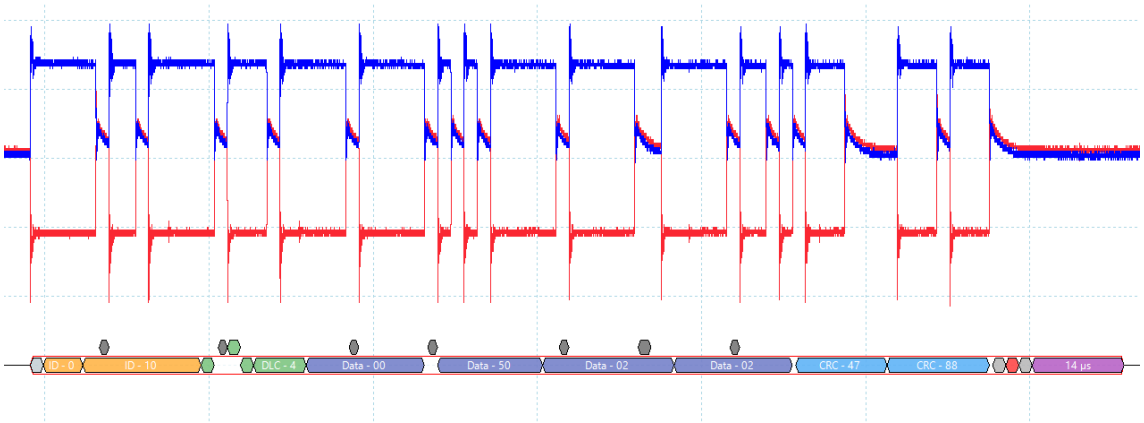

Figure 4.11: Oscilloscope image of received CAN message with WDT_CONFIG request.

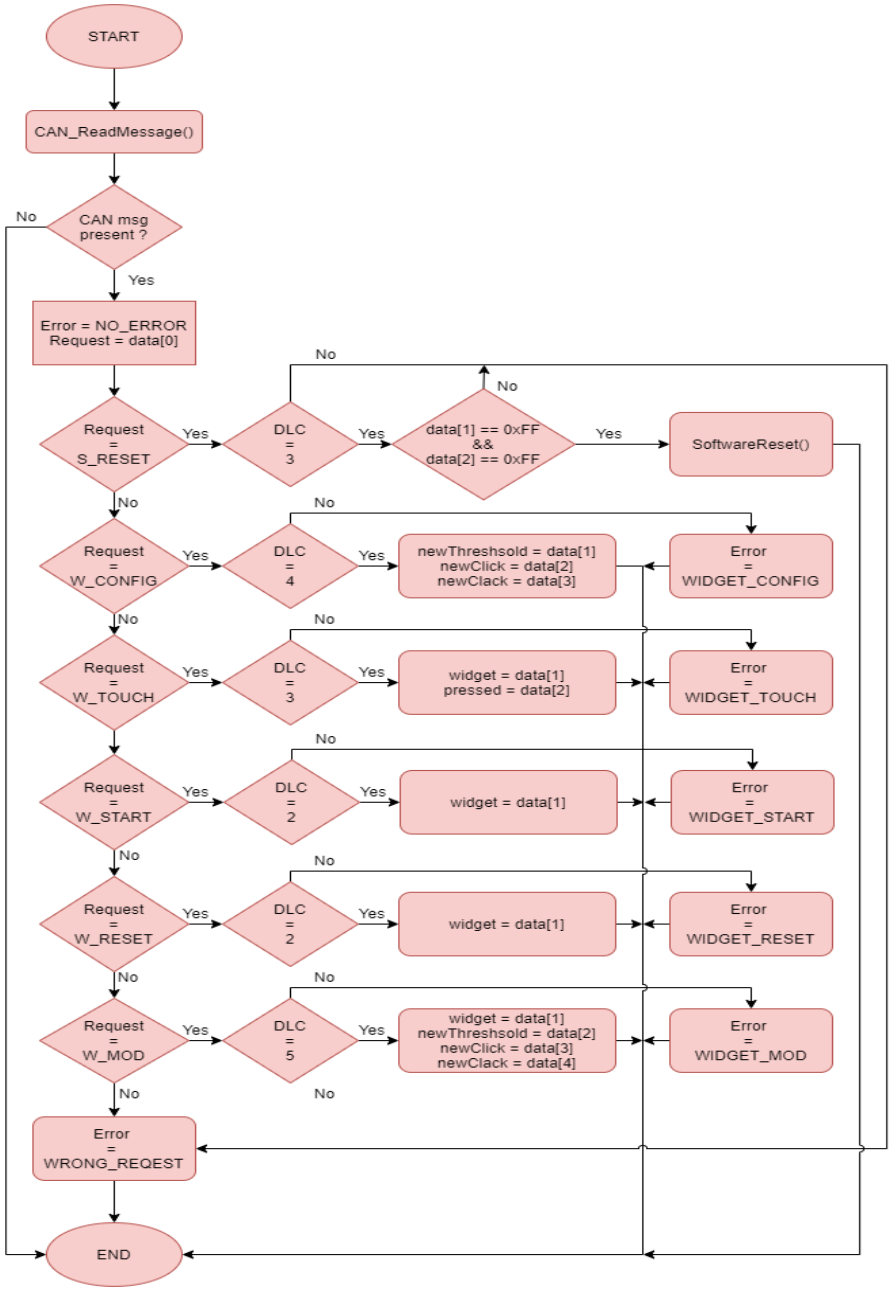

Figure 4.12: CAN reception task.

In this state, following the previous task, an additional check is made which consists in aborting the execution of the current haptic effect upon the occurrence of particular conditions.
The conditions for which execution is interrupted are:

- The new request refers to a widget different from the current one and the current one is active (executing or waits for to carry out the release effect).

- The request is that of immediate execution (WDT_START) but the current widget is active.

If one of these conditions is satisfied a signal is set that will communicate the will to terminate the current execution. This signal is processed before reaching the next stage of the FSM, as the NC_Task responsible for the execution and detection, is performed every milli second while the next state must wait for 3ms before being considered allowing correct and safe management.
This check is useful in the case of subsequent and extremely rapid execution requests, by suppressing the previous one and executing the new one.

4.3.2 Processing State

The third stage of the FSM is the operational one. Based on the information obtained during the reading and encoding of the CAN message (see 4.3.1) the corresponding API is called.
The table 4.2 lists the possible requests, in particular the configuration, reset and modification requests are managed by the same function, varying only the parameters passed in input.
The function in question is an API called "NC_Widget_Init", it is one of the functions I introduced in the Niceclick library. It allows to make the definition or modification of a widget dynamically.
Initially, the desired haptic effects and the related perception thresholds were associated a priori to a widget, not allowing the user to customize it or make a decision.
Since the goal was to expand the library also for use in graphic software or that could be used in different contexts, I thought it appropriate to introduce the possibility that a programmer could independently define what type of effect to associate to a widget or at what threshold the actuation is generated, starting from a library of haptic effects already defined in the memory.
This function accepts four input parameters, the widget, the threshold and the effects associated to the pressure and to the release actions (indicated by a numerical value).
This function works in two ways: adding a new widget or modifying an existing one.
In the first case, the identifier of the widget is not passed, but a predefined value is used that recognized by the algorithm allows to satisfy this request and increase the total number of registered widgets.
The configuration is carried out by associating the desired feedback for both pressure and release to the data structure of the library, the latter may also not be present, checking that the received value is present in the effects library stored in memory otherwise it will be associated with a default one. Similar procedure for setting the threshold.
This type of management allows to overwrite the values, threshold or haptic effects, if the identifier of an existing widget is passed or add a new widget.
In this way, if the request is one of the above-mentioned, this function is called with the values to be set.

The other two cases, on the other hand, WDT_TOUCH and WDT_START, have a different management.
The first works with two parameters, the identifier of the widget and a flag for actuation procedure, that enable the check of the exceeding of the force threshold and actuate accordingly.
This type of management was also introduced to facilitate integration into an external device, allowing the force reading and actuation loop to be externally activated and deactivated. In this way it is the external device that takes care of the identification of a possible finger position and decides if it is the case to evaluate the applied force or not.
This management is possible through the combined use of two APIs.
The first called "NC_GetPushsts" allows to check whether the last force value read, for that widget, has exceeded the set threshold value by communicating to the system the identification of a press. Before carrying out this check, however, the function check if the system is not currently under actuation, so as not to create overlaps in the generation of effects.

Listing 4.3: NC_GetPushsts API.

```
uint8_t NC_GetPushSts(uint8_t position)
{
    uint8_t ret = NC_BUSY;
    if(NcLibData.isGeneratingPulse)
    {
        ret = NC_BUSY;
    }
    else if(NcLibData.pos[position].force > NcLibData.config.posConfig[position].threshold)
    {
        ret = NC_PRESSED;
    }
    else
    {
        ret = NC_NOT_PRESSED;
    }
    return ret;
}
```

The second, called "NC_StartHaptics", verified that the selected widget is supported and there are no other widgets active at the moment, sets a variable that will allow to activate the pulse generation mechanism described in previous section (see 4.2.1.3).

Listing 4.4: NC_StartHaptics API.

```
uint8_t NC_StartHaptics(uint8_t position)
{
    uint8_t ret = NC_GENERIC_ERROR;
    if(position < supported_widgets)
    {
        if(NcLibData.isGeneratingPulse)
        {
            ret = NC_PULSE_GEN_BUSY;
        }
        else
        {
            NcLibData.pulseReqPosId = position;
            NcLibData.pulseGenReq = 1;
            ret = NC_NO_ERROR;
        }
    }
    else
    {
        ret = NC_GENERIC_ERROR;
    }
    return ret;
}
```

These two functions are related because having verified the threshold crossing, "NC_GetPushsts", the pressure detection command is given and the actuation is possible, "NC_StartHaptics".

Listing 4.5: WDT_TOUCH case management.

```
1     if(ApplicationData.pressed)
2     {
3         if(NC_GetPushSts(ApplicationData.widget) == NC_PRESSED)
4         {
5             (void)NC_StartHaptics(ApplicationData.widget);
6         }
7     }
8     else
9     {
10        NC_StopPulse();
11    }
```

In a typical management, the external device sends a CAN message, asking to verify if a possible press can occurs, in this way the loop constitute by the force reading and actuation is opened (ApplicationData.pressed = true, according to the previous code). If the force applied on haptic surface exceeds the set threshold then the actuators will operate, if the force returns below the threshold, then the release effect is also actuated, if present (like in 4.7 graph). As long as the external device not change this status, the execution loop continues to run in order to handle cases of subsequent presses. When a new CAN message with the status update is received, the loop is closed (ApplicationData.pressed = false) and tapping elsewhere on the surface there will be no type of effect.
The loop is automatically closed if there is a widget change or another type of request is made.
The WDT_START request unlike the one just explained does not require to verify if the threshold is exceeded but immediately actuate the associated effect, using the same API.

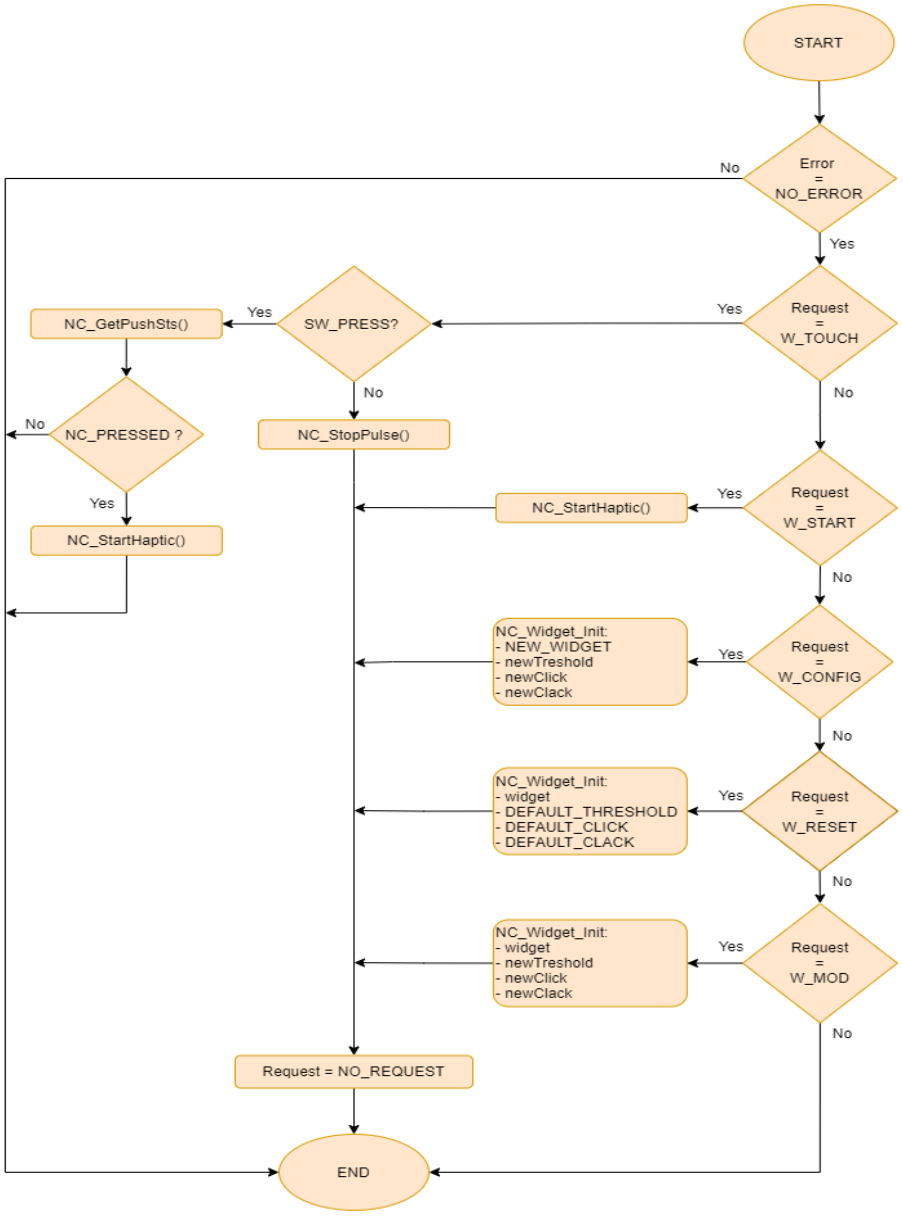

Figure 4.13: Application task.

Once the execution of this task is finished, before changing state of the FSM, the active widget is updated if it differs from the previous one used.

4.3.3 Transmission State

The transmission state is the latest of the FSM, in which the current status of the widget and its ID is communicated externally.
The operation that is performed consists in preparing and sending a CAN message containing this information.

The DLC is always equal to 2, as the first byte corresponds to the widget identifier while the second represents its status, pressed or released.
As explained in the 4.2.2 section, there is a task that identified a change in the state of the widget in action and saves this change within a FIFO.
Once in this state of FSM, the first element present in the FIFO will be transmitted and all the other elements, if present, are shifted by one position and the global FIFO index, representing the position of the last element present, is reduced by one.

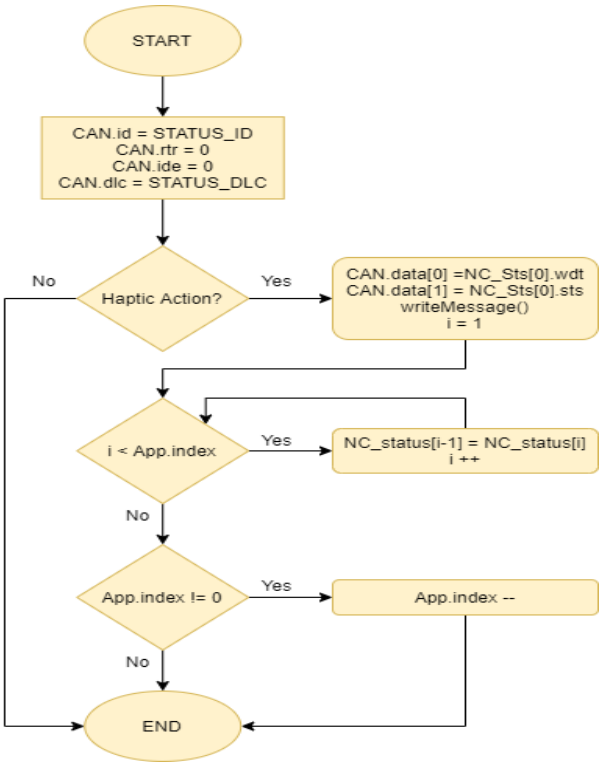

Figure 4.14: CAN transmission task.

Chapter 5

Heterogeneous multi-core processor: i.MX8M mini

This chapter provides an overview of the microprocessor used to control the graphic part of the project and the touch screen display, the i.MX8M mini. In addition, the peculiarities of this processor and the reasons why it was selected are explained. Being a heterogeneous processor is explained how this technology was born, how it works, what benefits it can bring and how the internal cores communicate with each other.
Finally, details are also provided about the carrier board in which the processor is mounted.

5.1 Overview on Heterogeneous Systems

Heterogeneous multi-core systems have two or more cores that differ in architecture or micro-architecture. Example of heterogeneous multi-core systems is the combination of a microprocessor core with a microcontroller class core in the same chip[7].
The choice to combine different types of cores lies in exploiting, depending on the situation, the benefits brought by one technology over the other. For example, analyzing different types of ARM cores, Cortex-A are powerful but consume a lot and introduce latency, so they are used in contexts where computing power is needed and consumption is not an issue. Whereas, Cortex-M have typical microcontroller features, adding determinism to an application and have low power consumption, so they are typical in embedded systems. Combining these two architectures is possible to span among multiple application domains, reducing physical space and gaining benefits from both speed and power consumption point of view.
Another reason for the introduction of heterogeneous systems, in addition to the growing demand for increasingly powerful and high-performance embedded devices, is due to the momentary stall in processor scaling. In scaling down, it was more difficult to decrease voltage and increase operating frequency while trying to maintain or reduce power consumption. The only remaining advantage was to exploit the advantages of integration, in agreement with Moore's law. Based on this consideration new technologies have been introduced that combine cores of different types and that achieve heterogeneous parallel processing on a single chip, always taking into account the power consumption[8].
When discussing multi-core systems, a distinction must be made between symmetric multiprocessing (SMP) systems and asymmetric multiprocessing (AMP) systems:

- **Symmetric multiprocessing systems**: These systems usually contain identical cores, or cores with the same instruction set, and run a single operating system with shared memory. Thanks to scheduler administration, the workload is distributed among the different cores equally[7].
- **Asymmetric multiprocessing systems**: These systems contain multiple cores, of the same or different types, and memory that can be shared or separate. Typically, different types of OS run on the various system architectures. A usual case is the coexistence of an embedded Linux operating system with a real time one, allowing a precise control of peripherals in parallel with an administrative part. The AMP system also is used in many use cases which can take advantage of an optimized core for specific types of computing[7].

The homogeneous multi-core configuration running in SMP mode can be considered the most popular way to scale processing, however the benefits of a heterogeneous multi-core configuration running in AMP mode may be the right fit for designs looking for efficiency in processing and power consumption.

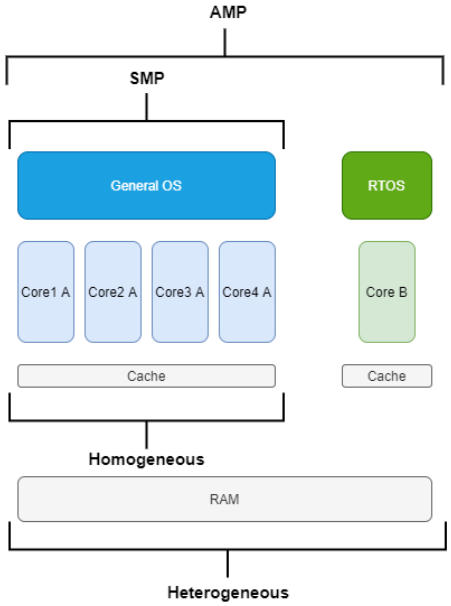

Figure 5.1: Mix Processing and Architecture.

5.2 i.MX8M mini

The i.MX8 is a family of products focused on delivering an excellent video and audio experience, combining media-specific features with high-performance processing optimized for low-power consumption.
The i.MX8M Mini media applications processor is built to achieve both high performance and low power consumption and relies on a powerful fully coherent core based on a quad Cortex-A53 cluster with video and graphics accelerators, and a general purpose Cortex-M4 core for low-power processing. Thanks to the coexistence of two cores with different architectures and functionalities, this processor can be defined a heterogeneous multi-core processor.
The target applications of this processor are for embedded consumer and industrial applications. Possible uses are for general purpose human-machine interface (HMI) solutions that enable touch, video and graphics, or for video and audio streaming devices. They can be used in IoT environments, high resolution 3D graphics, and image sensors.

The Cortex A53 cluster is a 1.8 GHz mid-range, low-power processor that implements the ARMv8-A architecture. The cluster has four cores, each with two 32 KB L1 memory system, instruction and data caches, and a single 512 KB shared L2 cache. Each core also features an FPU and a Media Processing Engine (MPE) with NEON technology. In addition, the ARM Cortex A53 processor consists of an integrated Snoop Control Unit (SCU), connecting the four cores within the cluster providing cluster memory coherency and implements the ARM Generic Interrupt Controller (GIC) architecture, that supports and manages interrupts[9].
All these features, especially the MPE, allow the simultaneous execution of different instructions with an improvement in the multimedia user experience and in 2D/3D graphics. For these reasons the Cortex A53 is used for high computing purpose and graphic.

The Cortex M4 processor is a 400 MHz low-power processor that offers low gate count, low interrupt latency, and low-cost debug. In addition, the M4 includes floating point arithmetic capabilities, two 16 KB L1 caches memory for data and instructions, and a 256 KB tightly coupled memory (TCM). Finally, it implements the ARMv7 instruction set architecture and adds significant functionality with the DSP and SIMD extensions[9].
This processor is intended for deeply integrated applications that require fast interrupt response capabilities, low power management, real time tasks, but also for control and signal processing targets.

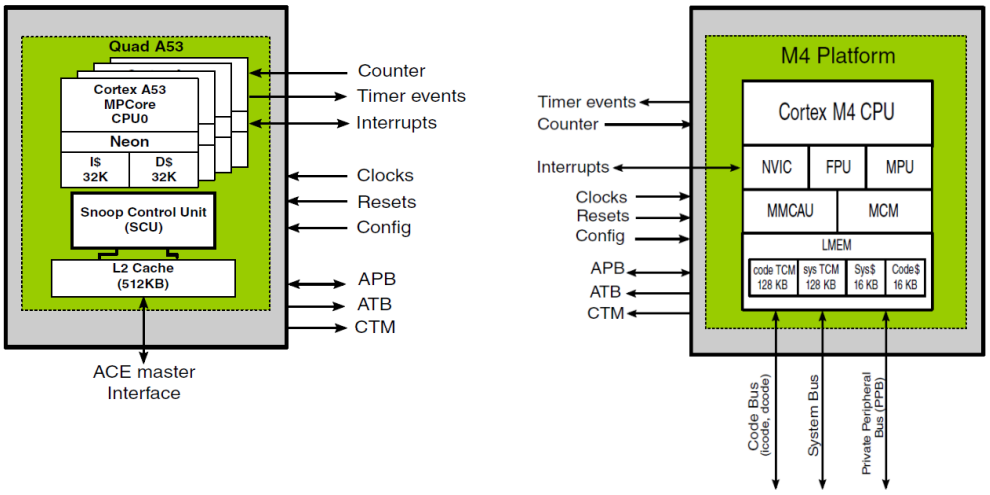

Figure 5.2: Left: Cortex A53 Block Diagram. Right: Cortex M4 Block Diagram. (Credits to NXP)

Around the two cores there is a series of peripherals and functional blocks that guarantee the correct functioning of the system, the application variety and the increase in number of the features supported by the whole processor.
Some of this feature are listed in the table 5.1. All these features allow supporting user interactions over an advanced multimedia service, such as a touch screen display, while combining them with traditional real-time embedded design requirements and maintaining low power consumption.
While in the following image is shown a block diagram of the structure, highlighting the bus communication, the internal connectivity and possible external peripherals that can be connected.

Connectivity	1 PCIe
	2 USB
	1 Gb Ethernet
	UART, I2C, SPI
	GPIO modules with interrupt capability
On-chip Memory	256 KB Boot ROM
	256 KB + 32 KB RAM
External memory interface	DRAM
	eMMC Flash, SPI NOR Flash
Multimedia	VPU, GPU
	Display Controller
	Audio Interface
Other peripherals	6 general purpose timers, 4 PWM, 3 WDOG
	Debug interfaces
	Multi-core supports

Table 5.1: Additional Features of i.MX8M mini.

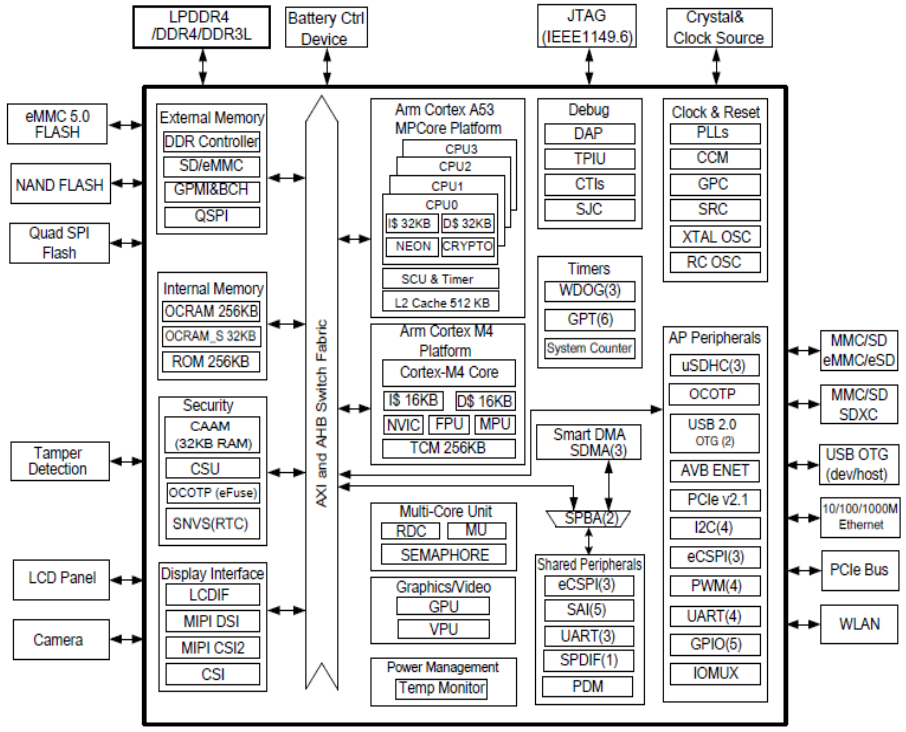

Figure 5.3: i.MX8M mini Block Diagram with interconnections (Credits to NXP).

5.2.1 Multi-core Management

The presence of multiple cores allow increasing in performance and flexibility. In some cases, the cores serve different functions and the software for each core may be developed by different providers. For efficiency reasons the code on the cores may share chip resources such as peripherals and memory. The sharing of chip resources between the somewhat independent processing domains allows possible data collisions where information stored in peripherals or memory by a process on one core is overwritten by software running on another core. Without careful collaboration between the two operating systems inadvertent malfunction or degradation in performance may result, and this happen because software does not take into account the hardware mechanisms already provided by the system[9].
To guarantee the best experience with the heterogeneous multi-core processing architecture, the software running on it must consider multi-core support to ensure safe access and allow access restrictions for peripherals and memory. The multi-core support includes the Resource Domain Controller (RDC), Messaging Unit (MU) and hardware semaphores. These three components are in place to guarantee a successful communication between the different cores.

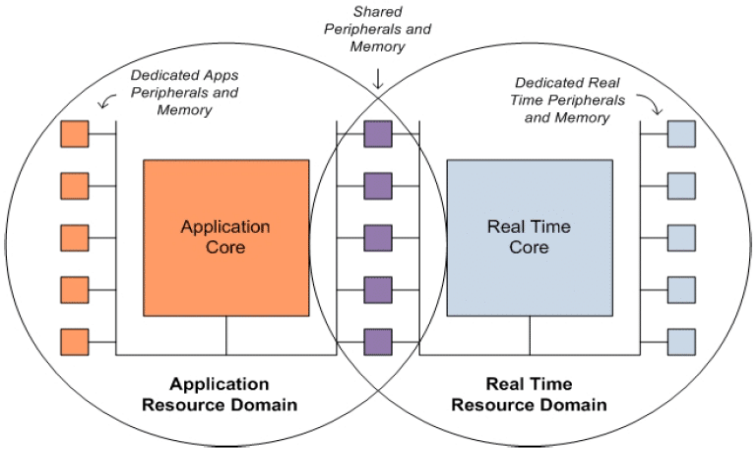

Figure 5.4: Dedicated and Shared Peripherals.

The RDC provides a mechanism to allow boot time configuration code to establish resource domains by assigning cores, bus masters, peripherals and memory regions to domain identifiers. This will allow, once configured, to monitor based on the domain identifiers and restricted access.
For shared peripherals, RDC permits more than one domain access to a single peripheral and provide three ways to control synchronized use of shared peripherals. These methods includes hardware-enforced synchronization, software-based semaphores, or no synchronization. The semaphore-based locking mechanism provide temporary exclusivity while the domain software uses the peripheral. Once the software of one domain has finished the task and finished with the peripheral then it may release

the semaphore making the peripheral available to the other domain, guaranteeing a "safe sharing"[9].

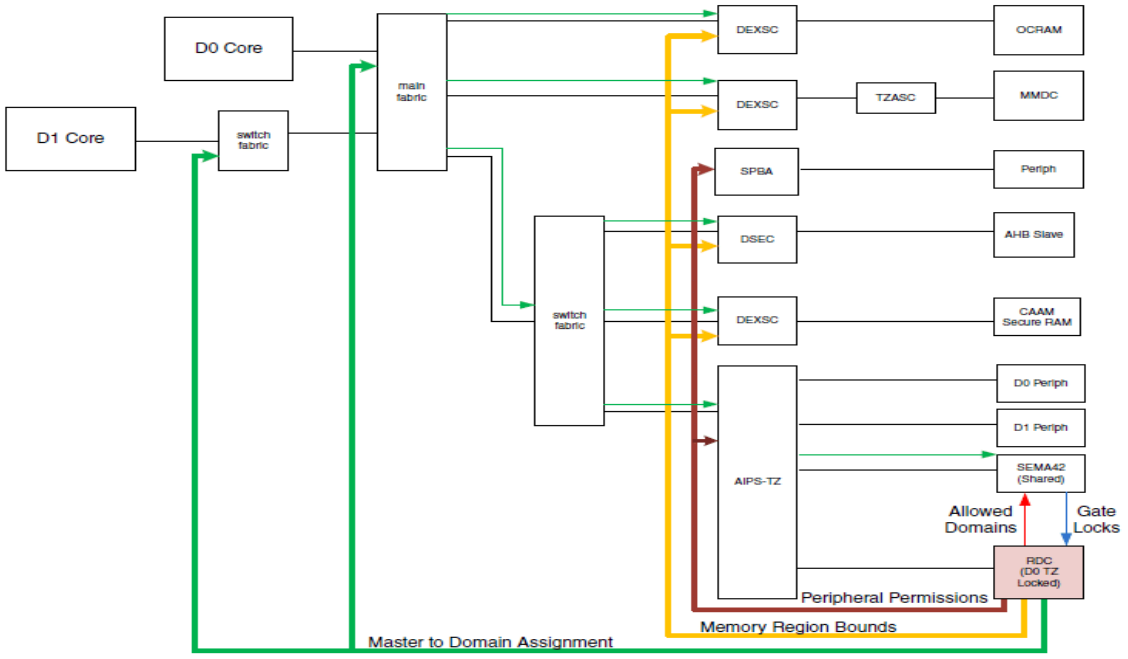

Figure 5.5: Example of RDC Connections.

In conclusion, the purpose of the RDC configuration is to enable isolation between core processing environments to prevent collisions and increase reliability.
The Messaging Unit module allows the two internal cores to communicate and coordinate with each other by passing messages through the MU interface. It also provides the ability for one processor to signal an event to the other processor using interrupts. The exchange of information takes place with the use of different clocks, due to the different peripheral buses, thus requiring synchronization on both sides and the MU accomplishes this job by using two sets of corresponding registers[9].
The main ability of this unit is the possibility for one core to wake-up or enable the other core by asserting an interrupt. Another important capability is the direct access to the shared memory resources of the SoC by one of the two cores. However, to prevent simultaneous access to shared memory by both cores, the MU provides a method (to prevent simultaneous access) by using interrupts and transmit-receive registers for both processors. Also, for messaging, the MU has four 32-bit write-only transmit registers and four 32-bit read-only receive registers on both processors, used to send messages from one to the other.

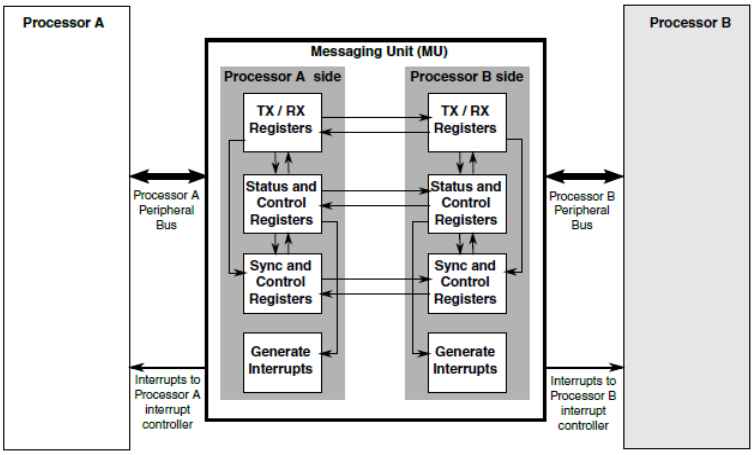

Figure 5.6: MU Block Diagram (Credits to NXP).

All these components will require some support in the software solution running in different cores. Operating systems such as Linux for the graphics core and FreeRTOS for the general purpose core, provide this type of support.

5.2.2 Multi-core Interaction and Communication

This section describes how the communication between these two cores is set and what types of interactions occur.
In a heterogeneous system and more generally in AMP (Asymmetric Multi-Processing) systems, given the presence of multiple processors in a single chip, must be able to run different operating environments side by side. OpenAMP (Open Asymmetric Multi-Processing) fits into this context, which is a framework that provides the necessary software components to enable the development of software applications for AMP systems. It allows operating systems to interact within a wide range of homogeneous and heterogeneous complex architectures and enables asymmetric multi-processing applications to take advantage of the parallelism offered by the multi-core configuration.
OpenAMP adheres to master-slave topology wherein master software controls the life cycle of the remote software and establishes the communication channel. The core building blocks of OpenAMP are Remoteproc and RPMsg. They provide resource management and communication features respectively.

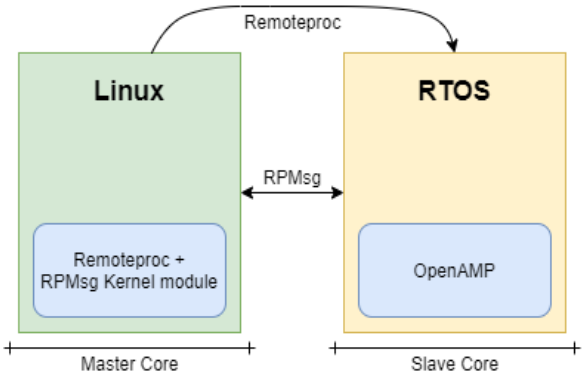

Figure 5.7: AMP system configuration with OpenAMP framework.

The life cycle management capability of OpenAMP is provided by Remoteproc.
The remoteproc framework allows different platforms/architectures to control (turn on, load firmware, turn off) remote processors. Remoteproc allows to load the software image, in ELF format, for the remote processor. Within the image is a data structure called resource table, which lists the resources required to activate the core. These resources include memory for the remote image, shared memory for data I/O, and trace buffers for the remote software. The resource table is initially populated by the remote processor with requests for the master. The master's software parses the ELF file, obtaining from it the information to be searched for. Once the information are identified and inserted into the table, the master loads the program segment into memory and starts the remote core from the entry point. In the same way once the remote core is turned on it can be turned off by the master.This procedure highlights why during boot the system always starts from the master core while the secondary core is activated only at the end or only on explicit request.
The communication between the software contexts is provided by the RPMsg component. RPMsg protocol enables two heterogeneous processor cores to communicate using a shared memory. The technique uses single-writer single-reader circular buffers to pass message to the other core. Thanks to this approach, this protocol does not require any multi-core synchronization element such as multi-core mutex[10]. The hierarchy of the protocol and its matching to ISO/OSI model is shown in figure below.

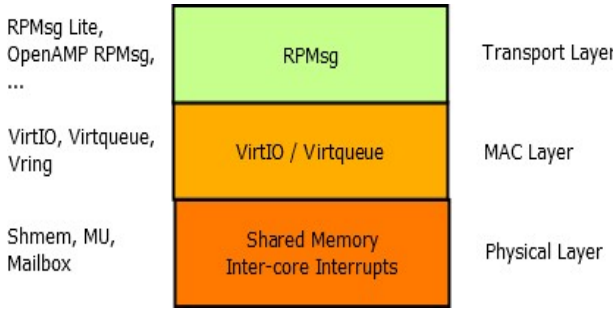

Figure 5.8: Layered model of RPMsg protocol.

The RPMsg protocol is based on VirtIO/virtqueue component as a shared memory-based transport abstraction. VirtIO defines a representation of virtual devices in the shared memory and defines APIs to read/write to device memory. VirtIO also defines a communication abstraction known as virtqueues. This is used to transfer data between guest and host. RPMsg uses the same abstraction to exchange data between the master and the remote. Internally, virtqueues consist in a ring buffer, known as vring. The vring resides in the shared memory and contains ring of buffer descriptors, that contain the pointers to buffers exchange between the master and the remote and read/write permissions[11].
Every remote core in RPMsg component is represented by RPMsg device that provides a communication channel between master and remote, hence RPMsg devices are also known as channel. RPMsg channel is identified by textual name, local (source) and destination address. The RPMsg framework keeps track of channels using their names[12]. The RPMsg channel is usually dynamically created and the remote processor advertises its presence to the master by sending the Name Service (NS) announcement containing the name of the channel. It is also possible to have a static channel where both sides know the name of the channel in advance.
Another important component in the RPMsg protocol is the endpoint. RPMsg endpoints provide logical connections on top of RPMsg channel. It allows the user to bind multiple rx callbacks on the same channel. Every RPMsg endpoint has a unique source address and associated call back function. When an application creates an endpoint with the local address, all the further inbound messages with the destination address equal to local address of endpoint are routed to that callback function. Every channel has a default endpoint which enables applications to communicate without even creating new endpoints[12]. So, if a channel has more than one endpoint, it is possible to transmit messages at different points using the same communication line, but activating different callbacks.

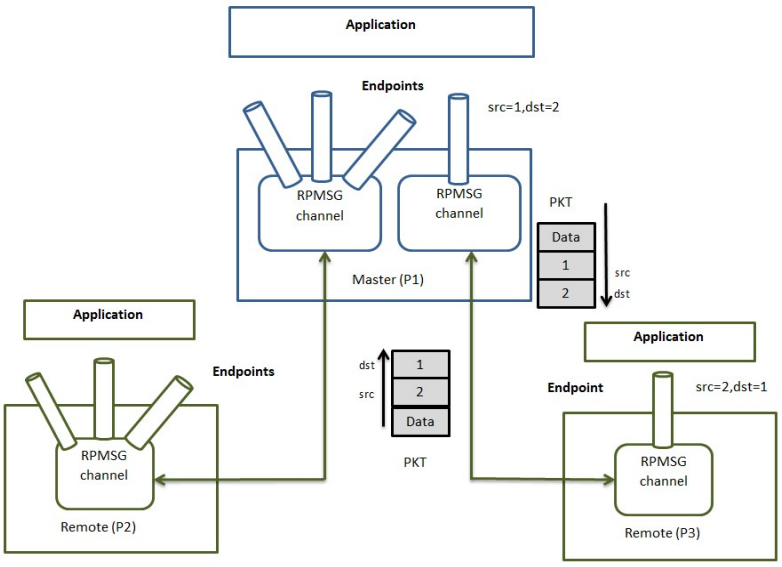

Figure 5.9: Concept of channels and endpoints in the RPMsg framework.

Having defined the concepts and elements on which the protocol is based a possible example of application is given in the [10] paper, which is very similar to the logic adopted in the project of this thesis.
The document says that on the primary core can run Linux OS and a graphical user interface (GUI), meanwhile the secondary core can be used for motor control. The core considered to be critical validates itself all data received from the primary core via RPMsg. It can also provide some data to the primary noncritical core and these can be visualized via a GUI. The two algorithms are therefore, completely separated and as RPMsg does not require any multi-core mutex, in case of a failure of the non-critical system, the critical part can continue to execute.
This protocol is becoming a standard in the communication of heterogeneous systems thanks to its integration in Linux Kernels and to large silicon companies such as NXP, Qualcomm or Xilinx that are adopting it for their platforms.

5.2.3 Considerations

The use of a heterogeneous processor allows high processing requirements to be met while satisfying low power requirements. They allow to extend the application areas of an embedded system to graphics processing, audio and IoT devices, following their growing demand. These types of devices have high demands both from a computing point of view but also from a control and security perspective. The ability to combine the capabilities of a processor and a microcontroller on a single chip allows these goals to be achieved and also reduces PCB space, therefore the number of used components. Power consumption can be further reduced by the ability to put the main processor on standby while the secondary processor is operating. Instead, the inter-process communication makes the system safer, avoiding to export the

communication externally, but managing it all from inside with specific protection and control mechanisms.
The ability to connect all these features brings great benefits to a system with great needs, which generally could present either one or the other particularity by making compromises. In the case of this project, it was essential that the processor was able to manage the graphics quickly and control those classic peripherals of a microcontroller putting them in a real time environment.
As will be explained in the section 6.2.4, a secondary objective of the thesis was to study this technology with the aim of assessing whether this processor was able, given the presence of the Cortex M4, to control entirely within it both the graphics and the control of the force and the actuation.

5.3 i.MX8M mini SOM and Carrier Board

In order to take advantage of all the potential of the i.MX8M mini is necessary that this processor is placed on a board that exposes to the outside all its connectivity and a mechanism to program and access to it.
On the market there are several development boards, the one selected for this project is made by the company Variscite. In particular, the processor is mounted on a module, the VAR-SOM-MX8M-MINI, integrating other ICs that expand the potential of the processor. In addition, this SOM can be connected to a carrier board that provides connectors, buttons, leds, etc., so as to facilitate management and direct access to the peripherals. This board is called Symphony-Board.
The most relevant additional functionality and features introduced by the SOM are listed below.

- 4 GB of DDR4 RAM memory.
- Non-volatile storage memory (for Flash Disk purposes, OS image, Boot-loader and application/user data storage). Up to 64GB of eMMC and NAND with density of 512MB.
- High performance ultra-low power stereo CODEC optimized for portable audio applications.
- Wi-Fi and Bluetooth platform, for both Dual band and Single band.
- Ethernet transceiver (AR8033).
- CAN-FD controller (MCP2517FD).
- MIPI-DSI to Dual Channel LVDS Bridge (SN65DSI84), for display control.
- Touch Panel controller (TSC2046I).
- JTAG connection.

While the connector that will connect to the carrier board allows to export some pins of the processor and those of the components listed above.
The Symphony-Board is a complete development board, utilizing the VAR-SOM-MX8M-MINI System-on-Module's features. It is assembled with large variety of user and debug interfaces enabling it to serve as both a complete development kit or as a stand-alone end-product.
The board provides the following interfaces and connectors:

- Display support: 2x18-bit LVDS Interface supporting Variscite's 7" TFT capacitive touch LCD.
- Touch panel interface: Capacitive (I2C based), Resistive (SPI).
- Connectors: Ethernet, PCIe, SATA, USB, Headphones jack, µSD-Card slot.

- CAN Bus
- Debug USB
- RTC.
- Additional: UART, PWM, SAI, SPI, I2C, GPIOs, LEDs, Buttons.
- Power: 12VDC input, 5V and 3.3V header.

Thanks to the availability of these elements it was possible to make the connections with the display, also provided in the kit, with the CAN network and to program the board and analyze its behavior.

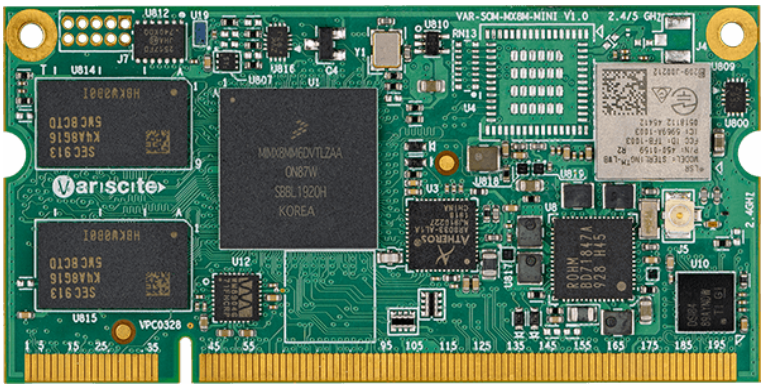

Figure 5.10: VAR-SOM-MX8M-MINI.

Figure 5.11: Symphony-Board.

5.4 Operative Systems

To support the hardware component there is the part related to the operating system that will run on the main core.
The selected operating system is Yocto with the Dunfell kernel version.
The Yocto project is a framework to create a Linux distribution for embedded devices. Its layering mechanism makes it easy to add Linux to new target devices highly customized for a particular platform; it can include custom boot scripts, software packages built with a high degree of optimization for a particular architecture, and several user interfaces from the full Gnome desktop to a simple serial console.
The release that was installed is based on NXP's BSP layer for the Yocto framework.Variscite extends this layer to support its System On Module products.
Thus, the operating system is optimized to fit the processor in question, minimizing functions typical of an OS, which in the case of an embedded system would not be useful, taking also advantage from the available technology. An example is the management of the heterogeneous part thanks to the use of specific kernel modules, like the one to control RPMsg protocol.
Regarding the operating system that can run on the secondary core, the Cortex M4, FreeRTOS has been selected as it is available in the SDK provided by NXP for this specific architecture.
FreeRTOS is a real-time operating system optimized for embedded devices, which provides several features for secure management of system tasks.
The SDK, as will be explained in the next section, also provides drivers for peripherals and support for the RPMsg protocol.

Chapter 6

i.MX8M mini - Cortex M4: Firmware

This section explains the functions and operations performed by one of the two cores present in the i.MX8M mini processor, the Cortex M4.
As explained in the chapter dedicated to this processor (see 5.2), the Cortex M4 can perform real-time functions or all those typical applications of a microcontroller leaving all the management of more complex tasks, such as graphics, to the main core, Cortex A53.
Keeping this division of roles, it was decided to exploit the Cortex M4 for the management of the haptic part communication, sending commands and status feedback back and forth acting as an interface between A53 and external board.
Furthermore, in this chapter the second objective of the thesis is explained from which comes the decision to use a heterogeneous system, and especially the Cortex M4 even if its role could be done by the Cortex A53.
The following sections shows how these requirements have been met and implemented, how the code running on this processor works and what libraries it needs for its operations.

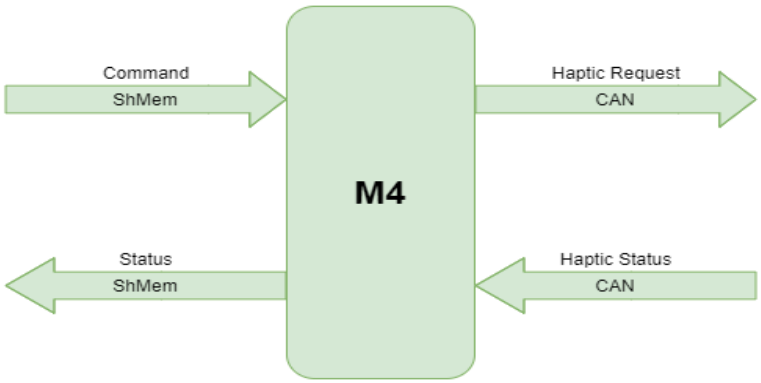

Figure 6.1: Cortex M4 Global interface.

6.1 Libraries and drivers

In order to allow the correct functioning of the code it is necessary to include appropriate libraries and drivers. To support programming, NXP Semiconductors provides a specific software development kit (SDK) with the presence of several drivers and libraries that allow the processor to be programmed.
Below are listed the libraries and drivers not present in the SDK or which play a crucial role in the implementation of the application. In particular, a porting of a library for the control of an external CAN module, the MCP2517FD, was carried out and libraries that allow the maintenance of the real time operating system are explained.
Finally, it is explained how the communication between the two core is made possible, i.e. using a lite version of the RPMsg protocol.

6.1.1 CAN-FD Driver

A fundamental attribute required for the development of this project is the functioning of the CAN communication. This type of communication is also necessary for the connection with the external microcontroller, the CY8C4247AZIL485. Moreover, being a product that could be placed also in the automotive market, the control of this type of communication is essential.
Unfortunately, in the mini version of the i.MX8M processor there are no internal peripherals that allow CAN management. However, a specific module is integrated in the SOM that allows the processor to connect it to the CAN bus. The module in question is the MCP2517FD, an external stand-alone CAN FD controller with SPI interface, which is able to support both CAN2.0B and CAN FD formats.
Its internal structure contains the following main parts:

- CAN FD Controller module: implements the CAN FD protocol and contains the FIFOs and Filters.
- SPI interface: used to control the device by accessing SFRs and RAM.
- RAM controller: used to arbitrates the RAM accesses between the SPI and CAN FD Controller module.
- Message RAM: used to store the data of the Message Objects.
- Oscillator: used to generates the CAN clock.
- Internal LDO and POR circuit.
- I/O control.

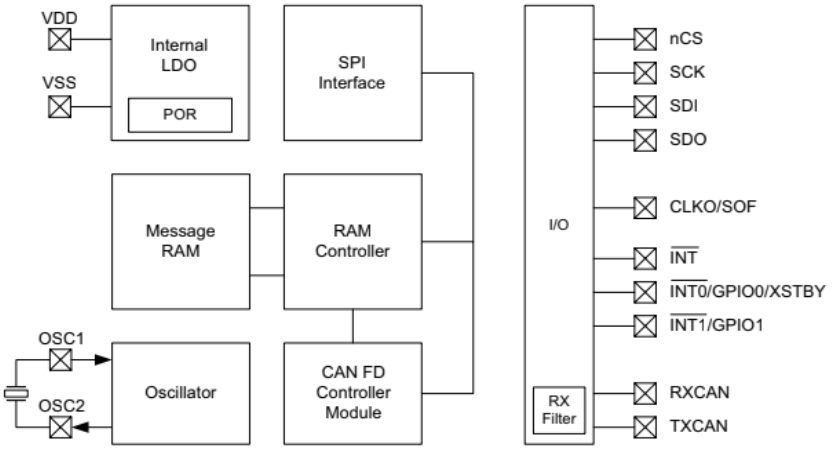

Figure 6.2: MCP2517FD block diagram.

In particular, also the CAN FD Controller module can be divided into sub-blocks each with its own functionality. The most important one is the CAN FD Bit Stream Processor (BSP) that implements the medium access control to the CAN FD protocol described in ISO 11898-1:2015. It serializes and deserializes the bit stream, encodes and decodes CAN FD frames, handles acknowledgement frames, detects and reports errors.
The module also has blocks for managing transmission and reception. The TX Handler prioritizes the messages that are requested for transmission by the Transmit FIFOs, while the RX Handler uses acceptance filters to filter out messages that shall be stored into the Receive FIFOs. Each FIFO can be configured either as a Transmit or Receive FIFO.
Finally, other blocks to consider are: the Transmit Event FIFO (TEF) that stores the message IDs of the transmitted messages, the Interrupt Controller that generates interrupts when new messages are received or when messages were transmitted successfully and the Special Function Registers (SFRs) used to control and read the status of the CAN FD Controller module[13].
Based on these elements the module is able to self-manage the communication by supervising the CAN Bus and the various information can be read or written by interacting with it, via SPI, when necessary. Through its interface it is possible to configure the module, transmit messages on the bus or read the FIFOs that have stored the data with the expected requirements.
Microchip Technology Inc., manufacturer of the module, has made available an example of a driver, written in C, for another type of microcontroller. In order to adapt this library to the Cortex M4 a porting has been done by modifying some components of it.
The driver is composed of the following 4 files:

- Device register addresses and structures
- Device specific defines

- API function for the CAN FD SPI controller
- MCU specific SPI definitions and declarations

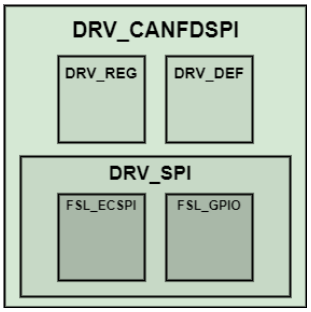

Figure 6.3: MCP2517FD driver structure.

The first two files contain all the information necessary for accessing and managing the internal registers of the CAN module. In particular, as regards the first file, the various addresses of the registers and the structure of each of them are defined, i.e. the attribution of the correct meaning to each bit that makes up the register is declared. The description of the contents of a register and the consequent representation in C is showed below.

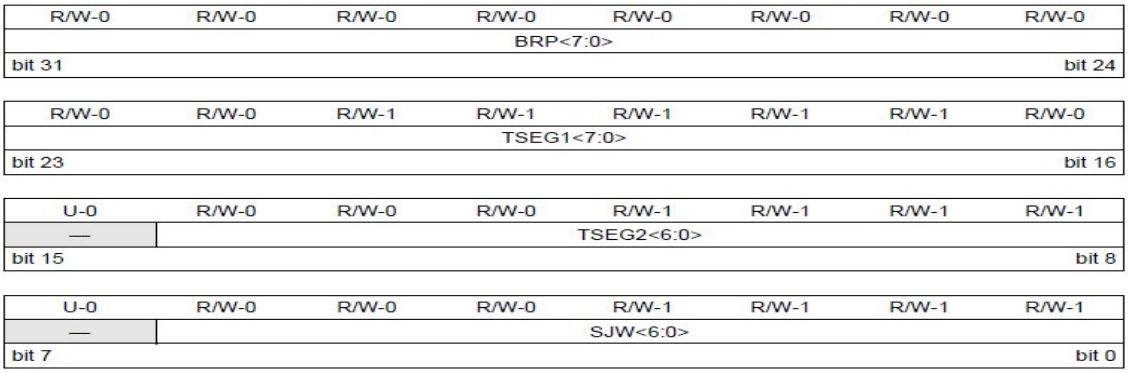

Figure 6.4: MCP2517FD data bit time configuration register structure[13].

Listing 6.1: C rappresentation of CiNBTCFG register .

```
1   typedef union _REG_CiNBTCFG {
2       struct {
3           uint32_t SJW : 7;
4           uint32_t unimplemented1 : 1;
5           uint32_t TSEG2 : 7;
6           uint32_t unimplemented2 : 1;
7           uint32_t TSEG1 : 8;
8           uint32_t BRP : 8;
9       } bF;
10      uint32_t word;
11      uint8_t byte[4];
12  } REG_CiNBTCFG;
```

The C implementation uses a construct called "union" which allows to manage the element in different ways: as a structure where each bit is specified, as a word or as a 4-byte array. In the case of the structure, each bit, in its position and size, corresponds to the one present in the table above. All the registers constituting the internal memory of the module have been defined in the same way.
The third file, on the other hand, contains all the APIs that are used to communicate with the module, either for setting issues or for the transmission/reception of CAN messages. In addition, they also allow to check of events, errors and get time stamps. The API categories present in the file are the follows:

- SPI access functions (read/write for different data dimension);
- Configuration;
- CAN transmission (transmission, FIFO and Queue management);
- CAN reception (reception, filter and channel management);
- Events and Error handling;
- GPIO, ECC and CRC.

In these three files the changes made have been minimal, for example in the file containing the APIs debug functions have been added, parameters removed from the function construct and new functions have been introduced that simplify some management.
The changes made that permit to carry out the porting were those concerning the SPI communication. Instead of creating a specific driver for this type of communication, as it is already present in the SDK provided by NXP, I have created a wrapper in which the data is transferred on the appropriate channel, toggling the GPIO port that manages the chip select (CS) of the slave, the MCP2517FD.
All the CAN write and read functions will contain the call to this function.
The function and how it is called in the CAN library is shown below.

Listing 6.2: SPI transfer data function.

```
1   int8_t DRV_SPI_TransferData(uint32_t *SpiTxData, uint32_t *SpiRxData, size_t spiTransferSize
    )
2   {
3       ecspi_transfer_t masterXfer;
4       masterXfer.txData   = SpiTxData;
5       masterXfer.rxData   = SpiRxData;
6       masterXfer.dataSize = spiTransferSize;
7       masterXfer.channel  = CHANNEL;
8       GPIO_PinWrite(CS_GPIO, CS_PIN, 0U); //enable the module transmission
9       status_t status = ECSPI_MasterTransferBlocking(ECSPI_MASTER_BASEADDR, &masterXfer);
10      GPIO_PinWrite(CS_GPIO, CS_PIN, 1U); //disable the module transmission
11      return (int8_t) status;
12  }
```

Listing 6.3: CAN single byte write function.

```
int8_t DRV_CANFDSPI_WriteByte(uint16_t address, uint8_t txd)
{
    uint16_t spiTransferSize = 3;
    int8_t spiTransferError = 0;
    // Compose command
    spiTransmitBuffer[0] = (uint8_t) ((cINSTRUCTION_WRITE << 4) + ((address >> 8) & 0xF));
    spiTransmitBuffer[1] = (uint8_t) (address & 0xFF);
    spiTransmitBuffer[2] = txd;
    spiTransferError = DRV_SPI_TransferData(spiTransmitBuffer, NULL, spiTransferSize);
    return spiTransferError;
}
```

In the SPI wrapper there is also an initialization function of the SPI communication, which allows to set the clock, the baud-rate, the polarity, the phase and the GPIO used as CS port. The values and data inserted were obtained following a series of tests and seeing the results on the oscilloscope.
The values can be seen in the table 6.4 in next sub-chapter.

6.1.2 FreeRTOS

FreeRTOS is a real-time operating system kernel for embedded devices.
It provides:

- A multitasking scheduler
- Multiple memory allocation options
- Inter-task coordination primitives, including task notifications, message queues, multiple types of semaphore, and stream and message buffers

An embedded application that uses an RTOS can be structured as a set of independent tasks. Each task executes within its own context, with no dependency on other tasks. Only one task in the application is running at any point in time. The real-time scheduler determines when each task should run. When a task is swapped out so another task can run, the task's execution context is saved to its own stack so it can be restored when the same task resume its execution[14].
To provide deterministic real-time behavior, the FreeRTOS tasks scheduler allows tasks to be assigned strict priorities. RTOS ensures the highest priority task that is able to execute is given processing time. This requires sharing processing time between tasks of equal priority if they are ready to run simultaneously. FreeRTOS also creates an idle task that executes only when no other tasks are ready to run[14]. This would ensure that hard real-time tasks are always executed ahead of soft real-time tasks.
For what concern the memory allocation and management, a RTOS kernel requires RAM each time a task, queue, or other objects are created. The RAM can be allocated statically at compiling time or dynamically from the HEAP. In the latter case the use of the standard C functions like "malloc" and "free" is not always appropriate because they are not thread-safe and they are not deterministic. For these reasons, FreeRTOS keeps the memory allocation API in its portable layer that

is outside of the source files that implement the core RTOS functionality. In this way is possible to implement application-specific for a real-time system.
It is possible configure FreeRTOS kernel by setting different parameters in it configuration file. In this specific application the FreeRTOS kernel is configured as follows:

Configuration option	Value	Usage
USE_PREEMPTION	1	Preemptive RTOS scheduler enabled.
CPU_CLOCK_HZ	SysClock	System clock frequency.
TICK_RATE_HZ	1000	The frequency of the RTOS tick interrupt.
MAX_PRIORITIES	5	The number of priorities available.
MIN_STACK_SIZE	90	The size of the stack used by the idle task.
IDLE_YIELD	1	The preemptive scheduler is being used.
TIME_SLICING	0	No switch between tasks of equal priority.
TOT_HEAP_SIZE	40960	The total amount of RAM in RTOS heap.

Table 6.1: FreeRTOS most significant configuration parameters.

6.1.3 RPMsg-Lite

RPMsg-Lite library is an open-source component developed by NXP Semiconductors used to implement inter-core communication between multiple cores in a heterogeneous multicore system. It works with Linux RPMsg master peer, that have its own driver, to transfer string content back and forth.
The RPMsg-Lite component is a lightweight implementation of the Remote Processor Messaging (RPMsg) protocol. Compared to the RPMsg implementation of the OpenAMP framework, the RPMsg-Lite offers a code size reduction, API simplification, and improved modularity.
Small MCU-based systems often do not implement dynamic memory allocation. For this reason in this lite version there is also the presence of static API, enabling the reduction of resource usage and increase the velocity of the communication[15].

Component/Configuration	Flash [B]	RAM [B]
OpenAMP RPMsg	5547	456 + dynamic
RPMsg-Lite / Dynamic API	3462	56 + dynamic
Relative Difference [%]	∼62.4%	∼12.3%
RPMsg-Lite / Static API (no malloc)	2926	352
Relative Difference [%]	∼52.7%	∼77.2%

Table 6.2: Comparison between OpenAMP RPMsg implementation and RPMsg-Lite implementation (credits to NXP Semiconductors).

The implementation of RPMsg-Lite can be divided into three sub-components. The core component (in rpmsg_lite.c), the one used to implement a blocking receive API

(in rpmsg_queue.c) and the third one used for dynamic "named" endpoint creation and deletion announcement service (in rpmsg_ns.c).
The "media access" layer, which is one of the few files shared with the OpenAMP implementation, defines the shared memory model, and internally defines used components such as vring or virtqueue.
The porting layer is split into two sub-layers: the environment layer and the platform layer. The first sublayer is to be implemented separately for each environment. The second sublayer is implemented in rpmsg_platform.c and defines low-level functions for interrupt enabling, disabling, and triggering[15].

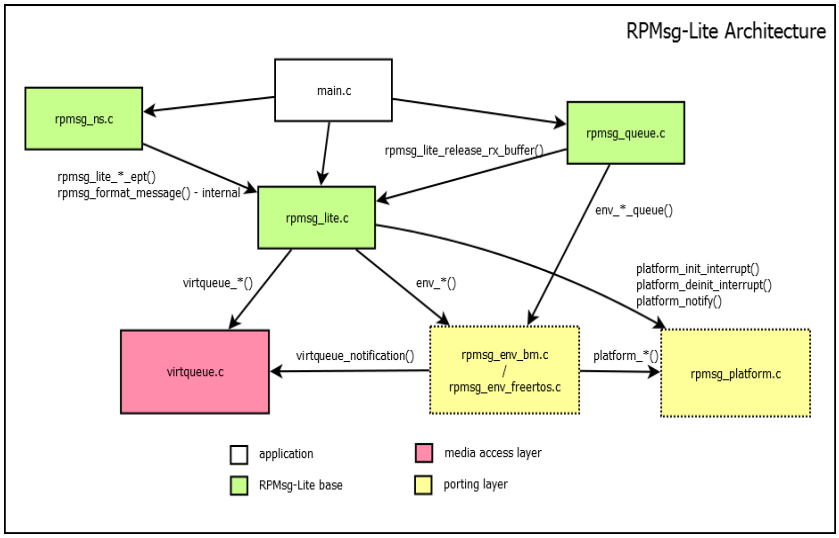

Figure 6.5: RPMsg-Lite Architecture (credits to NXP).

This sub-component implements a blocking send API and callback-based receive API. The RPMsg protocol is part of the transport layer, realized by using endpoints. Each endpoint can be assigned a different receive callback function.
The “Name Service” sub-component is a minimum implementation of the name service which is present in the Linux Kernel implementation of RPMsg. It allows the communicating node both to send announcements about "named" endpoint creation or deletion and to receive this announcement taking any user-defined action in an application callback. The typical usage of this library starts with the following procedures:

- Stack initialization
- Queue creation (in case of RTOS environment)
- Communication endpoint creation

Once all the environment is settled the communication can start by exchanging data using copy or no-copy send/receive mechanisms. The user is responsible for destroying and deinitialize any RPMsg-Lite objects whene they are no longer needed.

6.2 Software structure

The application written for this microcontroller has the role to manage the communication with the graphics processor through shared memory, using a lite version of the RPMsg protocol, while the interaction with the external processor is done via CAN bus.
It will periodically analyze the data present in the mailbox of the CAN transceiver managing the notification received at any change of the haptic status. Once, this information are received, they are forwarded to the A53 Core for graphic purpose. The reverse procedure is done when the graphic processor made an haptic request.
The entire control of the code and the correct scheduling of tasks is administered by a real-time operating system, FreeRTOS, in order to have the control of the execution times. The latter is a fundamental concept for all those applications that require adaptability and precision.

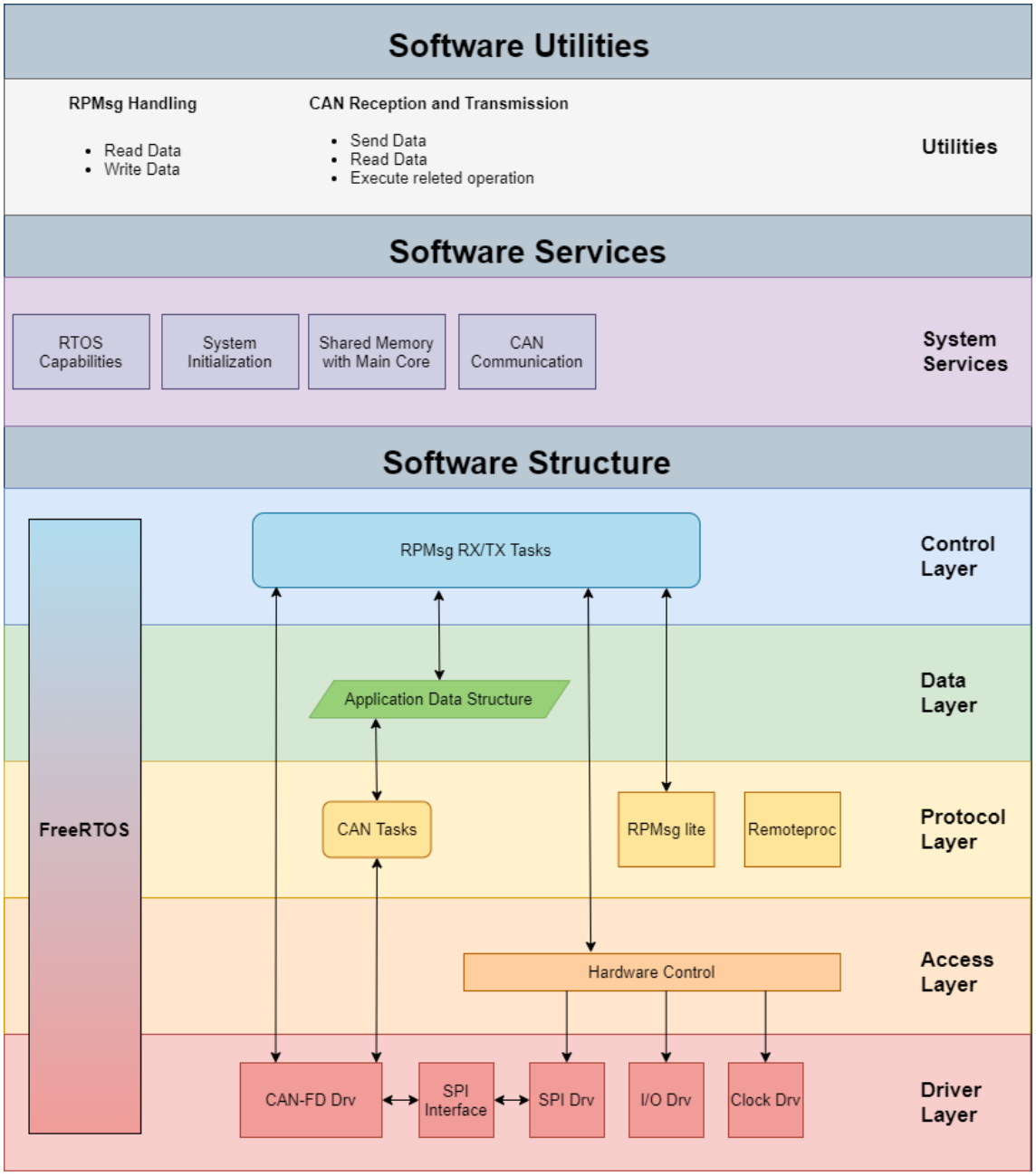

Figure 6.6: Software system structure for Cortex M4 core of i.MX 8M mini.

6.2.1 Initialization

To function properly, the system requires the initialization of the peripherals, memory and clocks used by the system.
The first initialized element is the Resource Domain Controller (RDC) which provides a mechanism to allow boot time configuration code to establish resource domains by assigning cores, bus masters, peripherals and memory regions to domain identifiers.

For shared peripherals, the RDC provides a semaphore-based locking mechanism to provide for temporary exclusivity while the domain software uses the peripheral. [9] This will avoid conflicts in case the Cortex A53 goes to use memory portions shared with the Cortex M4. Then the pins, clocks and memory are configured.

ECSPI1 MISO	Low
ECSPI1 MOSI	Low
ECSPI1 SCLK	Low
ECSPI1 SS0	High
UART3 RX	Low
UART3 TX	Low

Table 6.3: Cortex M4 pins configuration.

Another element that needs to be set up before system start-up is the CAN-FD module, MCP2517FD. To communicate with the module, SPI communication is required, which is why those pins have been initialized in the 6.3 table.
The initialization parameters of both SPI and CAN are listed below.

SPI Parameters	
Clock	80MHz
Baud-rate	10MHz
Burst lenght	8bit
Polarity	Active High
Clock phase	First Edge

Table 6.4: Cortex M4 SPI configuration.

CAN Nominal Bit Time Parameters		CAN Data Bit Time Parameters	
NBR	500kbps	DRB	2Mbps
SYSCLK	20MHz	SYSCLK	20MHz
NTQ/bit	40	DTQ/bit	10
NPRSEG	23TQ	DPRSEG	0TQ
NPHSEG1	8TQ	DPHSEG1	7TQ
NPHSEG2	8TQ	DPHSEG2	2TQ
NSJW	8TQ	DSJW	2TQ
N Sample Point	80%	D Sample Point	80%

Table 6.5: Cortex M4 CAN-FD configuration.

In addition to the initialization shown in the previous table, the receiving and

transmitting FIFOs are also initialized by setting the relative masks for filtering the correct IDs (in this case are accepted all IDs between 0x300 and 0x3FF).
The activation of communication between the M4 and A53 cores is carried out within the first task that will be described below, but fundamental parameters must be set in the appropriate configuration file before.

Configuration option	Value	Usage
RL_MS_PER_INTERVAL	1	Delay in ms used for polling.
RL_BUFFER_PAYLOAD_SIZE	496	Size, in byte, of the buffer payload.
RL_BUFFER_COUNT	2	Number of the buffers.
RL_API_HAS_ZEROCOPY	1	Zero-copy API functions enabled.

Table 6.6: RPMsg-Lite configuration.

Following the initialization phase, there is the creation of the tasks and the start of the RT scheduler.
In the program there are two tasks with equal priority, in addition to the system one with priority 0, and both with a stack space equal to 256 bytes.

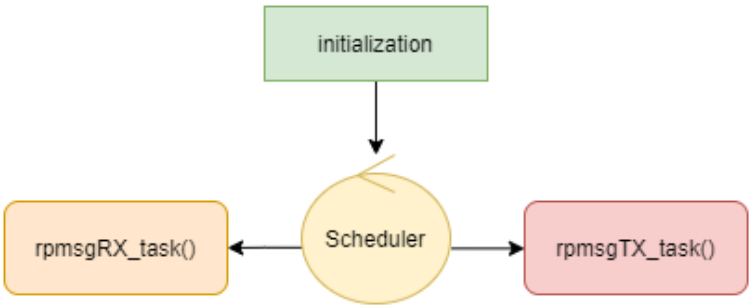

Figure 6.7: Cortex M4 scheduler.

6.2.2 RPMsg Receiving Task

The typical configuration of a task, which runs in an environment controlled by FreeRTOS, includes an initialization part performed at the time of creation and an operational part that operates periodically when the task is called.
In this task, initialization consists in establishing the connection with the Cortex A53. As soon as the graphics processor will initialize the RPMsg communication, via shared memory, this task will receive all the information necessary to create the components necessary for communication, storing them in a structure. It will then create the queue used for blocking reception and the endpoint, used as "address" in the communication process.
Finally, it is announced to the processor with which we want to communicate that the entire system is ready for data exchange and therefore communication can start (the name service handshake is performed to create the communication channels).

At this point the Cortex A53, when it wants to transmit data, it will refer to the endpoint so that the data will flow, through the channel created, to the Cortex M4 directly.

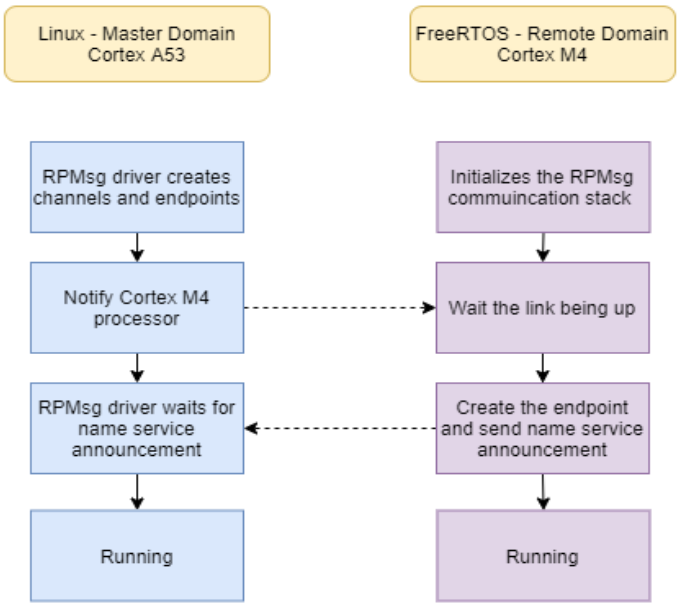

Figure 6.8: RPMsg communication setup.

The task loop has the role of taking the shared memory buffer, using a blocking function that allows to get only the pointer to the memory without copying its data, and reading its contents. During this period the buffer is controlled by this core and consequently the other core cannot transmit, therefore overwrite the data, but only receive until the buffer is released.
Once the buffer contents have been read, it will be converted from ASCII to binary, since the driver used by Linux expects to write data in this format, and stores it in the array that represents the data portion of the CAN messages, this is made only if the received data has more then 2 bytes (1 byte for the widget and the others for the data or type of request).
Finally, the CAN message is set and sent, using the driver described in 6.1.1 section, via the MCP2517FD module and release the RPMsg buffer.
This task checks the availability of the shared buffer every 1ms and transmits on CAN bus only if something is written to it otherwise the scheduler will pass control to the other task present.

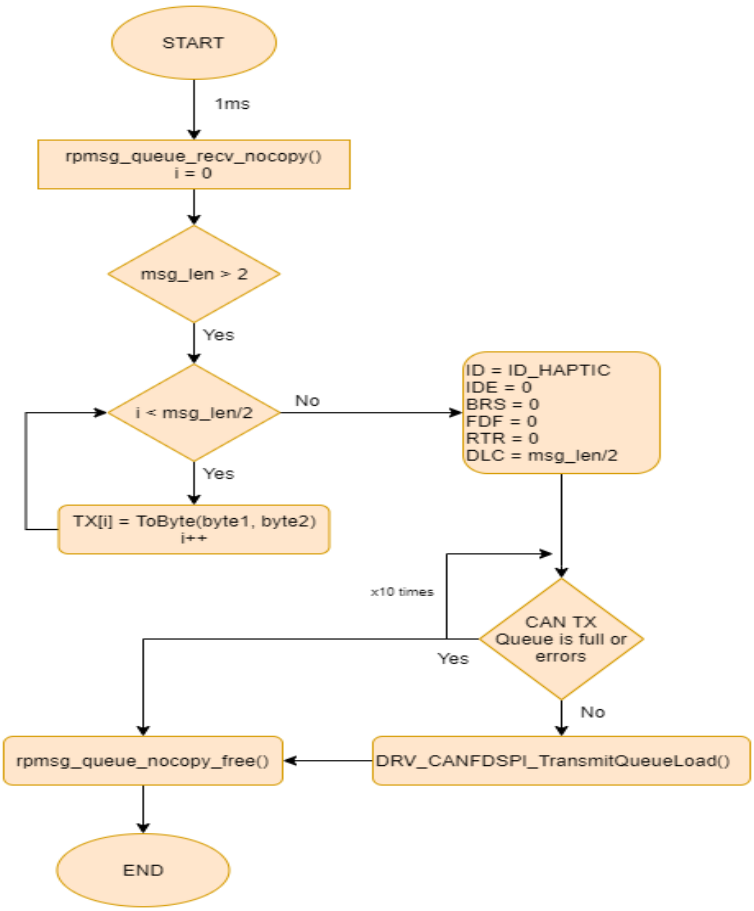

Figure 6.9: Cortex M4 Task for RPMsg reception and CAN transmission.

6.2.3 RPMsg Transmitting Task

The following task has the opposite role to the previous one, i.e. it receives a CAN message, containing the status of the haptic feedback, and if it is in the correct format, transmits it directly to the graphics processor, writing the data in a buffer allocated in the shared memory area.
In particular, the CAN communication is managed autonomously by the external module so if a message arrives on the CAN bus, with an ID in the range of IDs allowed by the filter set, it is stored in a FIFO present in the module itself.
The task will check, periodically, if there is a message in the FIFO and in base of the ID it performs a particular operation. In this case, with an ID equal to 0x300 only two bytes of data are expected, containing the identifier of the widget that has actuated and its status (0x01 if pressed and 0x00 if released).
Once the presence of the data is confirmed, they are converted into ASCII format, as required by the RPMsg Linux driver present in the other processor. Subsequently, a buffer for the message payload is allocated in the shared memory area following the RPMsg protocol, in this way by simply filling the buffer and making a write request,

the data travels directly to the wanted processor.

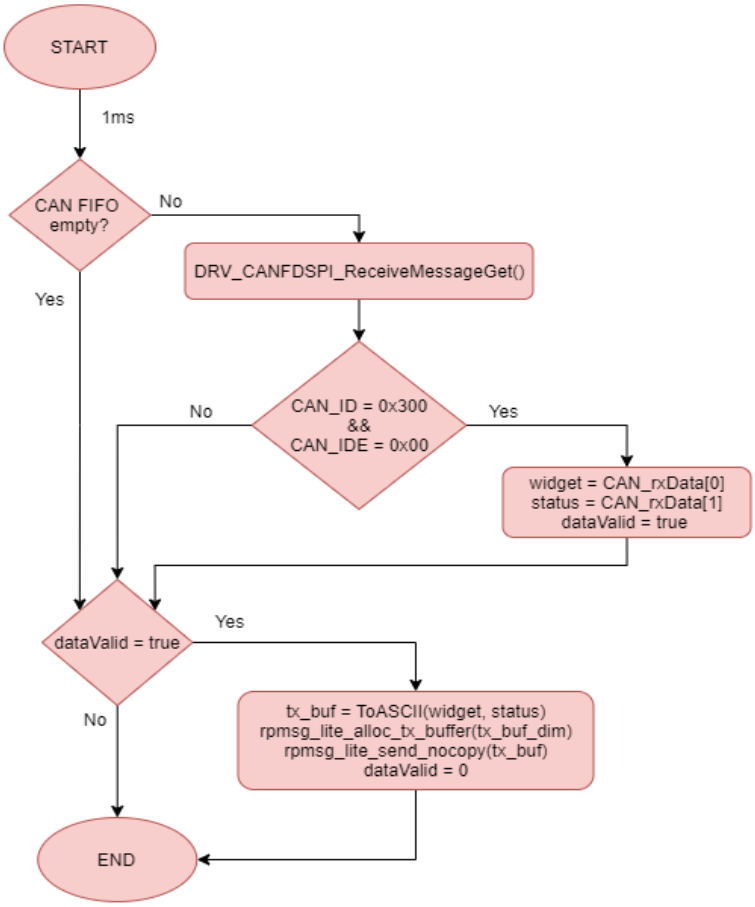

Figure 6.10: Cortex M4 Task for CAN reception and RPMsg transmission.

6.2.4 Future Haptic Management integration and others possibility

The use of this core for the purposes of the project does not bring great benefits other than separate the graphic context from the application context, the management of CAN communication.
The decision to use it in any way was taken to be able to understand its potential and if in the future only this processor would be able to manage the haptic and the graphics part at the same time without resorting to an external processor (in this case also due to HW necessity), obtaining the result shown in the figure.

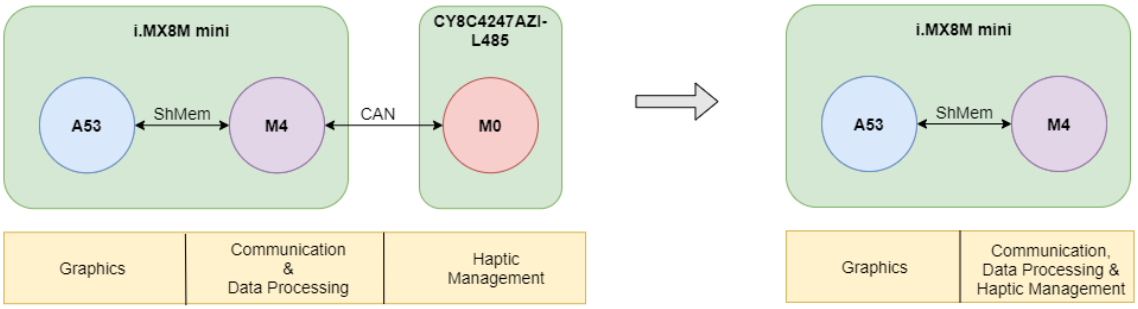

Figure 6.11: Actual system organization and future system organization

Following various experiments and tests, it has been established that this processor has all the requirements to perform both tasks since it has all the peripherals necessary for the correct functioning of the haptic system (force management and actuation). The necessary peripherals are:

- 2 16bit timers
- 4 PWMs
- 4 16bit counter timers
- I/O controllable ports
- Communication peripheral

Some of those are showed in the 4.2.3 section.
The single M4 core can control all of this peripherals autonomously while the A53 quad-core performs its functions in parallel.
Furthermore, a possible use of this core would bring greater computational benefits as the maximum working frequency is 400MHz compared to the 48MHz of the CY8C4247AZI-L485.
The only lack is the absence of internal comparators that can play the role of detecting the threshold exceeding in the calculation of the applied force.
One solution would be to use an external IC that performs this job and then bring the output of the comparator to a pin of the microcontroller or emulate the comparator at the IT level, but it would reduce the benefits of using interrupts.
The Niceclick library has been set up to port to any processor, changing only the content of basic functions that allow to manage at high-level the used peripherals, called internally by the API of the library.
These functions are the start and stop of the timers, the activation of the PWMs and the variation of their DC, the mode changes for the capture and the I/O management of the flyback diodes. By simply changing these functions it is possible to allow the library to work on any microcontroller.
A further fundamental concept is the exchange of information between M4 and A53, having a shared memory avoids possible external communications, going to write

and read portions of internal memory.
In this application, following what is explained on the RPMsg protocol, a single channel is activated in which information is exchanged between the two cores. In the case of more complicated applications or an extension of this project, multiple endpoints could be defined so that different information can travel in the predetermined direction and therefore not refer to the same memory location.
One test I did set up two endpoints, one transmits the commands from the graphic part and receives feedback on the status of the actuation while the other allows to periodically transmit the temperature status of the system. The usefulness of using two endpoints is that it is known with certainty that one endpoint refers to the read of temperature and to the other to the haptic state transmission, avoiding defining protocols or procedures but simply reading at the correct location.

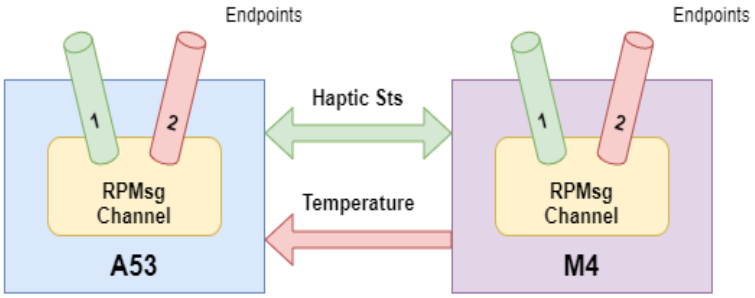

Figure 6.12: Example of 2 endpoint use

In conclusion, the use of a heterogeneous processor such as the i.MX8M mini would allow haptic and graphic management simultaneously, bringing benefits from the point of view of speed and inter-core communication, guaranteeing quality and ease of management.

Chapter 7

i.MX8M mini - Cortex A53: Graphical User Interface

This chapter illustrates the realization of the graphic part of the device and which principles it wants to convey.
The realization of an application with which the user can interact was designed to demonstrate the potential of haptic actuators, the Niceclick, and how with the addition of haptic sensations can be achieved a level of realism, in the interaction, totally different from those to which we are accustomed with normal devices.
The graphics has been developed, using Qt framework, to run on an embedded version of Linux on the quad-core Cortex A53.
The objectives and preliminary analysis, up to the implementation of the application itself and how it interacts with the heterogeneous system are discussed below.

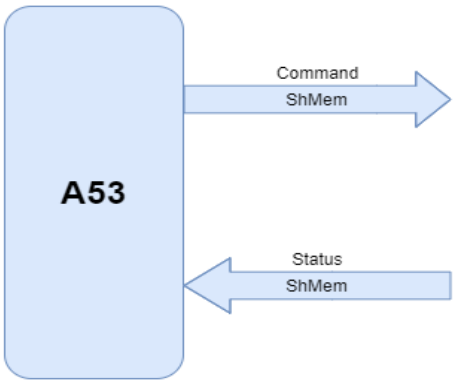

Figure 7.1: Cortex A53 Global interface.

7.1 Objectives

The purpose of this project was to demonstrate the potential of Niceclick, the haptic actuators, i.e. create situations that highlight the efficiency and the possibilities that a product supported by this technology can have.
The Niceclicks allow for example the simulation of the press and the release of a button, applying force and obtaining in response a feeling that emulates a mechanical component. However, their application cannot be only limited to the purpose to replace a button but also to attribute to an element a haptic sensation. The sensation can be, for example, the perception of a texture, the vibration following an event or the perception of shapes.
The integration of Niceclicks with a device can be done with any surface, making it a haptic device. In the following image is represented a platform used to test the various haptic effects, thought it is possible to demonstrate how on any surface it is possible to perceive haptic sensations if equipped with Niceclicks. Several experiments have allowed the surface to be perceived as "soft" or as something "mechanical".

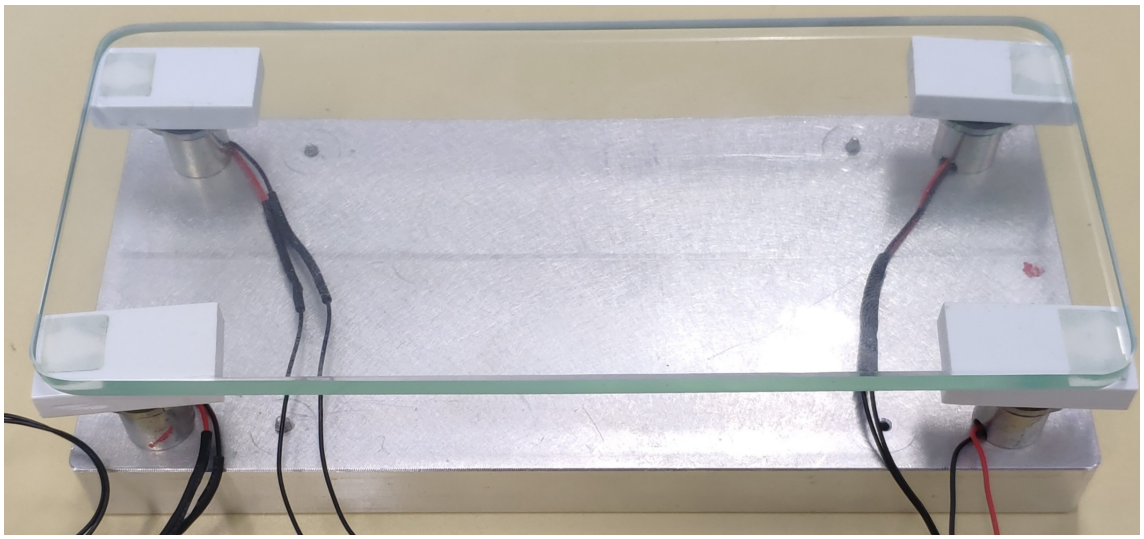

Figure 7.2: Niceclick test platform.

However, the limits of this application are linked to the impossibility of localizing the position of the finger, solved by surfaces integrating capacitive membranes. Thanks to which more complicated and different management are possible. This type of solution is congenial for all those devices that require a fixed position for the haptic perceptions or visual part that not require changes as result of press or release.
These assumptions define the purpose of this thesis, which is to integrate Niceclicks and their capabilities on a display. The aid of a display allows not only the localization of the finger but also the definition of a supporting graphical application. The latter had to demonstrate that the change of state of an element could happen only when the force exceeds the set one and had to show different elements with which the user

could interact, understanding what haptics means and what capabilities it can give to a device. So, the graphic support allows to add visual effect to the tactile one, increasing the realistic effect of the experience.

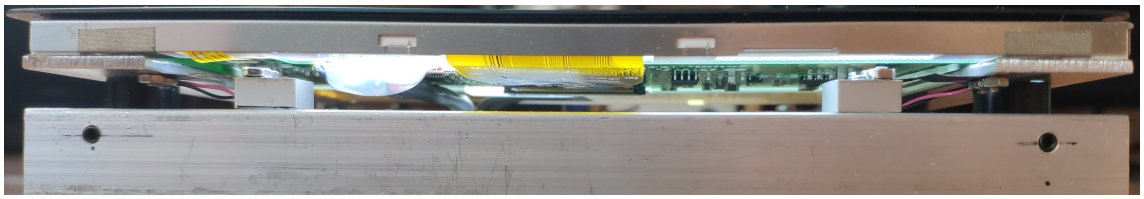

Figure 7.3: Display Support.

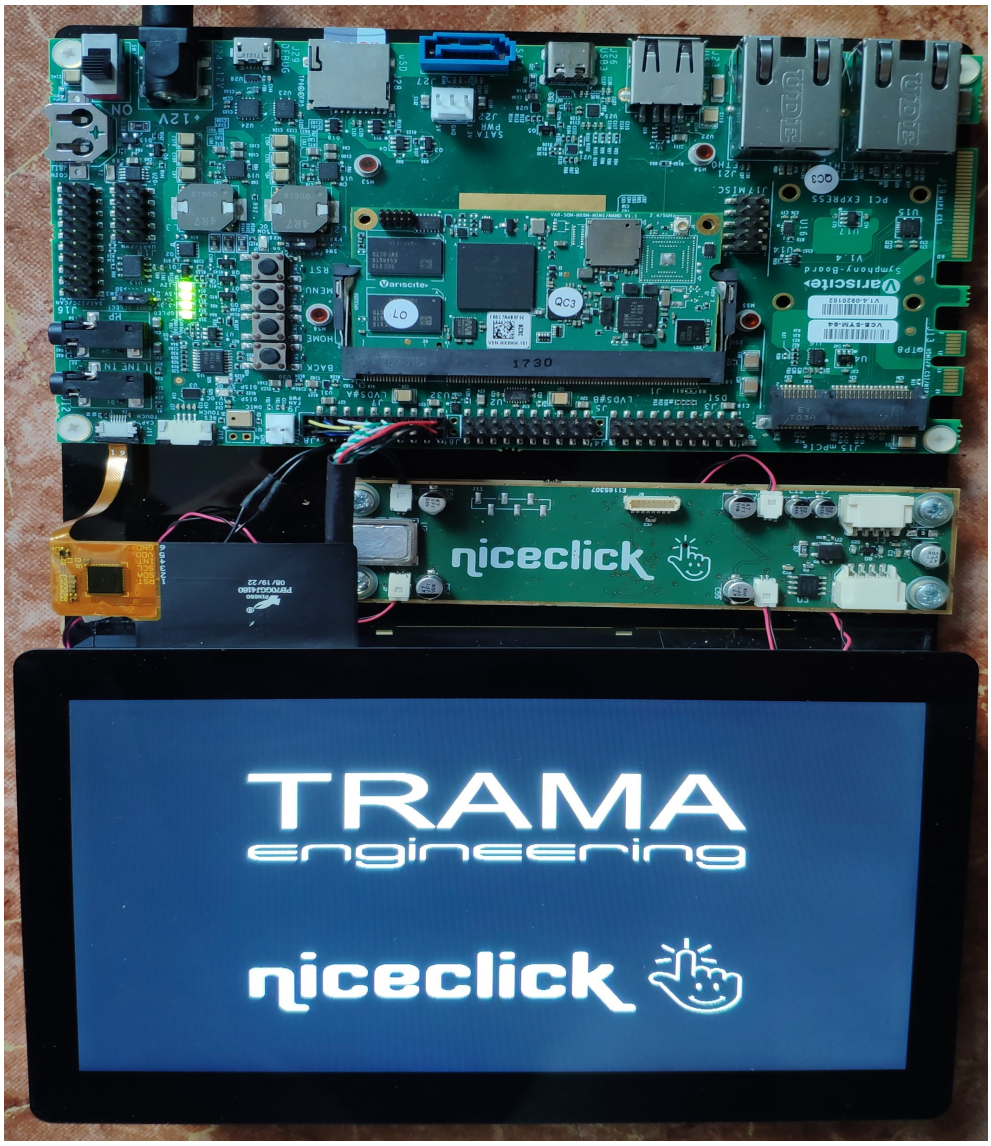

Figure 7.4: Final Device.

7.2 Preliminary analyzes and Solutions

To ensure what was said in the section 7.1 a graphical user interface was created to allow the user to interact with the display.
The realization of the GUI had several objectives, in particular, it had to serve as a touch identification system, so present a mechanism that would allow to track the X and Y coordinates and send accordingly the haptic request, on the CAN Bus, if the finger was in the area of interest or if a gesture was performed. Another objective was to create graphics that would highlight the ductility of this product, showing what tactile sensations could be perceived and how much a digital application could be closer to a real element.
Therefore, it was necessary to find a way to create an application that would satisfy these requirements and at the same time integrate well with a Linux operating system and its classic management mechanisms.
The choice fell on the use of the Qt/QML framework.
Qt is a widget toolkit for creating graphical user interfaces as well as cross-platform applications that run on various software and hardware platforms such as Linux. Qt supports various compilers, including the GCC C++ compiler, and also provides QtQuick, which includes a declarative scripting language called QML that allows to use JavaScript for logic. With QtQuick, rapid development of applications for mobile and embedded devices has become possible, making graphical elements with a tool that models their shape, position, color, etc., while the logic can still be written even with native code for the best possible performance.
In this way it is possible to separate the back-end part from the front-end part, the first one programmed in C++ that manages the control logic and the more complicated functions while the second one, written in QML and developed with QtQuick, will have the task of creating the graphical components and their basic functions, communicating to the back-end part possible changes of state or calling functions following an event.
In particular Qt is based on key concepts such as signals and slots.
Signals and slots are used for communication between object and its mechanism is a central feature of Qt and probably the part that differs most from the features provided by other frameworks. When a signal is emitted, i.e. when its internal state has changed in some way, the slots connected to it are executed immediately, just like a normal function call. Slots do not know if they have signals connected, which ensures that signals and slots are completely independent mechanisms[16]. It is also possible to associate several signals to one slot and several slots to the same signal or a signal that activates another signal. This mechanism makes easier to interact with graphics and exchange internal information. In that, the signal could be the on-screen touch and the slot contains a function that creates an animation or call a back-end function that executes a routine.
Moreover, writing Linux system calls as parts of slot functions, they can respond to specific signals by realizing the combination of a Qt application with Linux mechanisms. Of course, to achieve reading and writing of a specific device file,

there must be device drivers which provide reading and writing operation interface functions.
This aspect is fundamental for managing communication with the secondary core. As mentioned in the 5.2.2 section, data exchange takes place in a shared memory area and through the RPMsg protocol, which must be supported and enabled by both cores.
The management of the protocol on the Linux side is made possible thanks to a Kernel module that implements a virtual serial communication, TTY, with the other core, on which the data exchange is managed according to the rules of the protocol. In particular, the module being initialized will create the channel and notify the secondary core that it is ready for communication. Receiving this notification, the secondary core proceeds with its initialization and prepares for communication as shown in the image 6.8 in the 6.2.2 section. At this point, by performing reads and writes on this virtual serial, specific for endpoints, it is possible to transmit and receive messages with the remote core. The writing function performed by the driver is shown below.

Listing 7.1: Linux RPMsg driver write function.

```
1  static int rpmsgtty_write(struct tty_struct *tty, const unsigned char *buf, int total)
2  {
3      int count, ret = 0;
4      const unsigned char *tbuf;
5      struct rpmsgtty_port *rptty_port = container_of(tty->port, struct rpmsgtty_port, port);
6      struct rpmsg_device *rpdev = rptty_port->rpdev;
7
8      if (NULL == buf) {
9          pr_err("buf shouldn't be null.\n");
10         return -ENOMEM;
11     }
12
13     count = total;
14     tbuf = buf;
15     do {
16         /* send a message to our remote processor */
17         ret = rpmsg_send(rpdev->ept, (void *)tbuf,
18             count > RPMSG_MAX_SIZE ? RPMSG_MAX_SIZE : count);
19         if (ret) {
20             dev_err(&rpdev->dev, "rpmsg_send failed: %d\n", ret);
21             return ret;
22         }
23         if (count > RPMSG_MAX_SIZE) {
24             count -= RPMSG_MAX_SIZE;
25             tbuf += RPMSG_MAX_SIZE;
26         } else {
27             count = 0;
28         }
29     } while (count > 0);
30
31     return total;
32 }
```

Unfortunately, this module, already present in the operating system, does not have a specific reading function, but it can be done by reading the virtual serial line, then accessing a file and reading its contents, whenever there is a message. Checking for the presence of the message is another limitation of the driver as it will not be notified; therefore, the data will accumulate in the reading "file" until it is read. To have a quick system, messages should be read immediately, checking their presence constantly.
The solution to this is addressed in the following subsection.

7.2.1 Multithreading

When a Qt application starts, only the initial thread is running. This is the thread allowed to create the QApplication object and call exec() on it. For this reason, this thread is called the GUI thread. After the call to exec(), this thread is either waiting for an event or processing an event.
Since, this system must always read the virtual serial line as said previously, if there is only one thread, the GUI will be frozen by continuously reading the serial. So it was essential to create a new thread, in order to communicate with Cortex M4.
The decision to adopt this methodology was obtained by consulting the paper [16] in which a similar problem is faced. The assumptions made in this document are not limited only to the parallel processing of the two threads but also on how they should interact, since once the data has been collected they must be communicated to the GUI. In this case, the use of shared variables with the presence of mutexes and semaphores would still block the graphics. Instead, by taking advantage of Qt's signal and event mechanisms it is possible to communicate from a non-GUI thread to the GUI thread without affecting the normal operation of the system, as it is a thread-safe mechanism.
At that point the GUI thread will receive the data and can process it normally.
A secondary thread, in C++ with the help of the Qt libraries, is created using the QThread class. The first task performed by this thread, once created, is to activate its "run ()" function. In this case I did an over ride of it by making it first open the Linux file, in which the received messages will be present, and then emit a signal that will notify the main thread that the communication is now ready. Finally, it enters a loop where it constantly calls the "readRPMsg ()" function.
This function reads the file for an amount of bytes equal to what is predetermined and emits a signal that allows the transmission of this information to the GUI. In case no data is present, then to the OS scheduler is told to move to the next thread.

Listing 7.2: Secondary thread read function.

```
1   void RPMsg_Thread::readRPMsg()
2   {
3       int len;
4       len = read(_fd, _buff, RX_DLC*2);
5
6       if (len == RX_DLC*2){
7           _widget.id = ToByte(_buff[0], _buff[1]);
8           _widget.status = ToByte(_buff[2], _buff[3]);
9           emit messageReady(_widget.id, _widget.status);
10      }else{
11          sched_yield();
12      }
13  }
```

Instead the write operation, i.e. sending data to the Cortex M4 that will forward them to the CAN bus, consists in writing data on the virtual serial and then the Kernel module will perform the actual sending. The commands to be transmitted come from the main thread through a signal connected to a slot that performs the operation just mentioned.
Finally, there are two additional signals that this thread can handle, one is the

"close" signal and the other is the "error" signal. The first one is received by the main thread allowing the safe closing of all open Linux files and the sending of the RESET message to the external microcontroller, with the formatting present in the table 4.2. While the second signal goes in the direction of the main thread to warn it about the presence of an error in the functions, the consequence is the closure of the thread.

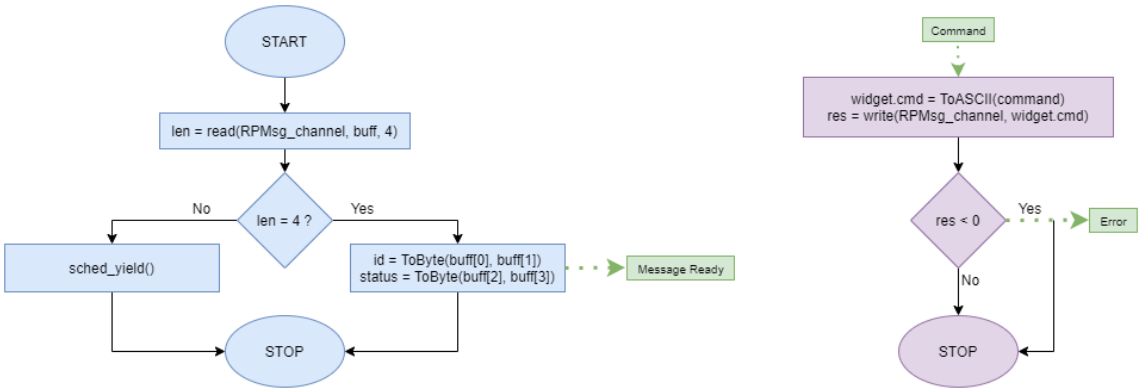

Figure 7.5: Functions of reading and writing to the virtual serial.

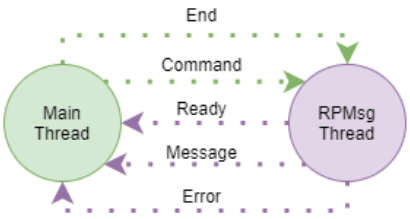

Figure 7.6: Signals exchange between main and secondary thread.

7.3 Back-end

The terms front-end and back-end denote, respectively, the user-visible part of a program with which the user can interact and the part that enables the operation of these functions or processes the input data and complex operations.
In the Qt environment the back-end part can be written in different programming languages, in the case of this project C++ has been adopted. For its definition was used the QObject library that allows to define an object with the functional parts of the Qt environment as the definition of signals and slots.
Once the object is created, the data structures and the secondary thread will be created. In the constructor of the class two activities are performed: the connection of signals and slots and the thread activation.
The activation of the thread is done by calling the start function, present in the library that manages the creation of Threads, which allows to execute the run function mentioned in the section 7.2.1.
In this way, each macro component participating in the communication between one system and another is defined. The entire communication, interaction, and closure procedure is reported in Appendix A.
The connection process consists of associating a signal with a slot. Specifically, in this case signals are associated between the back-end part and the secondary thread. The following is the portion of code that performs this function.

Listing 7.3: Signal and Slots connection procedure.

```
1    connect(thread, &RPMsg_Thread::messageReady, this, &Backend::handleMessage);
2    connect(thread, &RPMsg_Thread::error, this, &Backend::handleError);
3    connect(thread, &RPMsg_Thread::ready, this, &Backend::initNC);
4    connect(this, &Backend::sendCommand, thread, &RPMsg_Thread::handleCommand);
5    connect(this, &Backend::endThread, thread, &RPMsg_Thread::closure);
```

In the function the attributes are the object generating the signal, the signal, object presenting the slot, the slot.
The signals in question and their direction can be seen from the image 7.6.
At this point both the back-end part and the secondary thread are ready to work. Their operation depends on the interaction with the graphical part and the response of the actuators. The front-end part invokes different functions of the back-end part when particular situations occur, while the secondary thread does the same when it receives updates on the status of haptic widgets.
The operation of the system is based on the use and update of two arrays of structures, respectively called HW_WIDGET and ELEMENT. The first represents the definition of the widget that is saved and managed by the external microcontroller to perform the various haptic tasks. A widget has three parameters: the threshold at which the haptic effect is triggered, the ID of the effect that is generated upon press, and the ID of the effect upon release. These three parameters are the ones needed to store a widget in the external microcontroller, since as said in the 4.3.2 section, the external board stores inside it, at boot time, only the haptic effects but not the widgets, which are generated run-time following an external request. So this structure allows

to define the desired haptic widget which is then sent externally for initialization.
The other structure, on the other hand, has a more software meaning, as it allows to associate a widget to a particular graphic element. The structure is composed as follows:

- Graphic widget ID
- Haptic widget ID
- State of graphic widget
- State of haptic widget
- Haptic request supported

The graphic widget ID is unique for one element, instead several elements can present the same haptic widget.
The concept of state instead in the case of this application extends its possibilities. A graphical button would normally have a state in which it is pressed and one in which it is not, but if the concept of force threshold is added, the possible states become four:

- Finger present but force threshold not exceeded
- Finger present and force threshold exceeded
- Finger present and force threshold released
- Finger not present

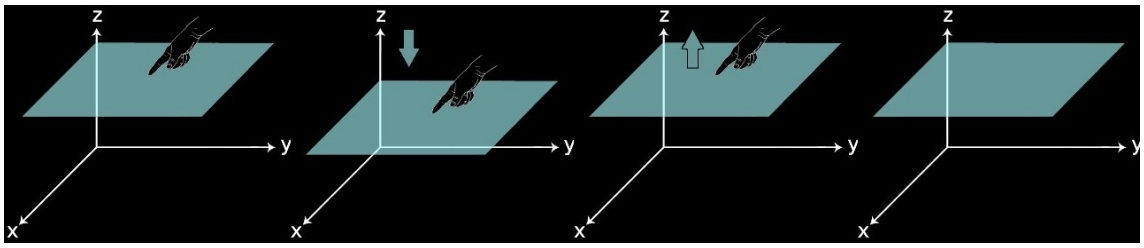

Figure 7.7: Possible system states.

The first and third states can be traced back to the same situation but are logically separated. Since the association of an effect to the release is present, it is generated only when a pressure has occurred and returns above threshold.
Finally, the haptic request corresponds to one of those listed in the table 4.2.
The back-end part is based on these definitions and structures, in particular all functions act to update the state of a graphical element by triggering an FSM that manages the system.

The main component functions of the back-end part are seven, divided into private functions, public functions and slots.

- FSM
- storeHWWidget
- sendRequest
- setElementStat
- updateElement
- handleMessage
- initNC

The first two are private functions and the last two are slots, the rest are functions that can be invoked from the graphics part.
"initNC" is a function that allows the initialization of the various elements and related widgets, and then send them to the Cortex M4 that will forward it to the external microcontroller. In particular, the ID of the graphic element and the type of request is set here while the other parameters are set by the "storeHWWidget" function.

Listing 7.4: Init procedure of an element.

```
1    _elements[0].SW_Widget_ID = 1;
2    _elements[0].request = WDT_TOUCH;
3    storeHWWidget(_elements[0].SW_Widget_ID, 0x14, 0x01, 0x01);
```

This last function is used to perform optimization operations, that is to avoid defining already existing haptic widgets but to get the ID of an already initialized one. The optimization is to cycle the widget array until it finds one identical to the one we want to set, if it is not found then new one is added and communicated externally to the haptic board for its storage. The initialization function, being a slot, is activated only when a signal appears, which notify the availability of the secondary thread to receive data.
The transmission of an information externally is done by passing the message to the secondary thread, via the "sendRequest" function. This function is called by the above function but also by those that want to transmit a request externally.
This function receives as input the ID of the graphic element and its request, based on the latter the message will be formatted in the right way. Since the array of elements is a global parameter, it can be accessed directly by the function, which obtains all the necessary information. Once the message to be transmitted has been created, a signal is emitted to pass it to the secondary thread, which will immediately transmit the message.

Listing 7.5: sendRequest function.

```
void Backend::sendRequest(int swID, int request)
{
    uint8_t command[5];
    uint8_t dlc = 0;
    command[0] = (uint8_t) request;

    switch(request)
    {
        case WDT_CONFIG:
            command[1] = _hwWidgets[_elements[swID - 1].HW_Widget_ID].th;
            command[2] = _hwWidgets[_elements[swID - 1].HW_Widget_ID].click;
            command[3] = _hwWidgets[_elements[swID - 1].HW_Widget_ID].clack;
            dlc = 4;
        break;
        case WDT_TOUCH:
            command[1] = _elements[swID - 1].HW_Widget_ID;
            command[2] = _elements[swID - 1].touchSW;
            dlc = 3;
        break;
        case WDT_START:
            command[1] = _elements[swID - 1].HW_Widget_ID;
            dlc = 2;
        break;
        case WDT_RESET:
            command[1] = _elements[swID - 1].HW_Widget_ID;
            dlc = 2;
        break;
        case WDT_MOD:
            command[1] = _elements[swID - 1].HW_Widget_ID;
            command[2] = _hwWidgets[_elements[swID - 1].HW_Widget_ID].th;
            command[3] = _hwWidgets[_elements[swID - 1].HW_Widget_ID].click;
            command[4] = _hwWidgets[_elements[swID - 1].HW_Widget_ID].clack;
            dlc = 5;
        break;
    }
    emit sendCommand(command, dlc);
}
```

So the full path to submit the request is the following one.

Figure 7.8: Complete path of sending a haptic request.

The function called "FSM" consists of a three state finite machine. Each state refers to the state of interaction with the display. The choice of three states instead of four is given by the fact that the distinction between the case of "Finger present but force threshold not exceeded" and that of "Finger present and force threshold released" is controlled internally to the same state in which they are grouped.
The states in question are:

- IDLE
- SW_TOUCH_HW_RELEASED
- SW_TOUCH_HW_PRESSED

Since "FSM" is a function and not a task of an OS it has to be triggered by something, in this case it is the change of state of an element. In case an element changes its

software or hardware state, this function will be called by the respective handlers, "handleMessage" and "setElementStat".
The first one receives the message arriving from the secondary thread, i.e. the physical change of the state of a widget, which will call the FSM function only if there is an active element and the ID of the widget arriving from outside corresponds to the ID of the widget associated with the active element.
The second function allows to change the software state of the element, i.e. if the finger is present or not on the detection zone, then calling FSM to manage this change.
The IDLE state is the first state to be considered and can only move in the direction of the SW_TOUCH_HW_RELEASED state when the finger is detected. This recognition will be communicated also to the secondary thread which will do the external notification for the execution of the corresponding command, which can be the opening of the loop of force detection and actuation or the immediate actuation.
If the FSM function is called in the SW_TOUCH_HW_RELEASED state, before updating the state, checks if the calling message has brought an update of the "hardware" state of the active element. This is done because if the force threshold is not exceeded it means that the update is in the "software" part of the element, so the finger has been removed. This situation makes the system go back to IDLE, warning the outside world. In case the force threshold has been exceeded and consequently the haptic effect on the pressure has been generated, the state can pass to the next one or return to IDLE. This condition is related to the type of request that is made and how the widget is constituted, since if the widget has as request of immediate execution and does not provide the effect on the release, it means that after the actuation the system goes back to IDLE again otherwise it continues in the next state. On the other hand, regardless of the latter condition, if the "hardware" state changes it will be notified via a signal to the front-end part, so that it can perform a function when that state changes. For example, it allows a button to be animated only when there is enough pressure to be considered valid.
The SW_TOUCH_HW_PRESSED status verifies whether a hardware release has occurred. If there has not been a “hardware” release, the activation signal could correspond to a "software" release and then the system returns to IDLE. This is the case when there is a pressure in progress but the finger slides out of the haptic area leading to an immediate activation of the release effect due to the closing of the haptic routine of the external board. If the release is neither "hardware" nor "software" the activating event is a widget that does not have the release effect, so there can be only subsequent presses or a return to IDLE. In case the finger is not removed but is pressed repeatedly the state will always remain the same, as the "hardware" release is not supported. If the release has taken place, the next status is SW_TOUCH_HW_RELEASED or IDLE if it is a WDT_START request and not a WDT_TOUCH request. In both cases the front-end part is notified of the state change.
Below is an image that illustrates the behavior of the function.

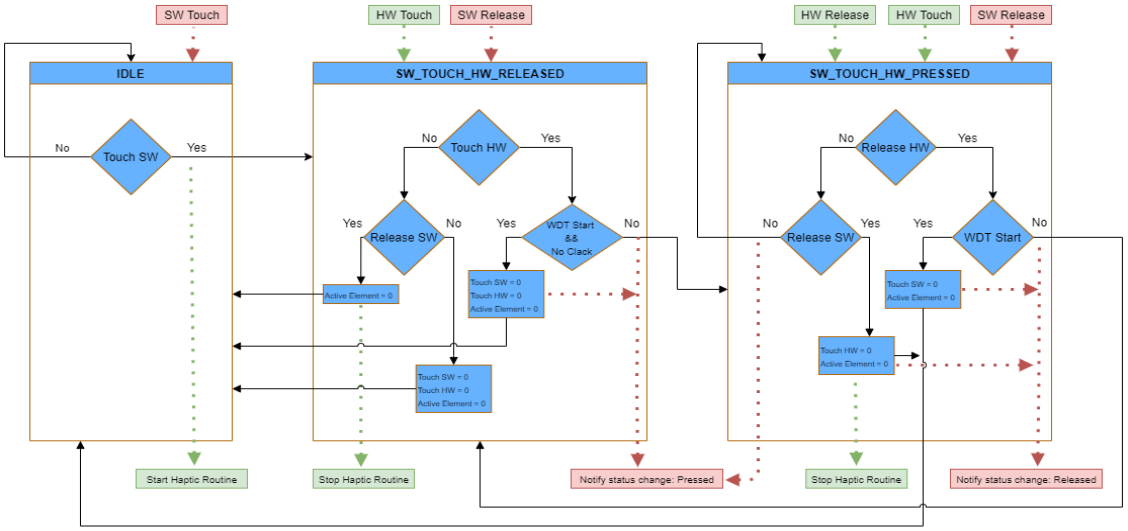

Figure 7.9: FSM function flowchart.

The green boxes are the signals that arrive or are sent by the secondary thread while the red ones are the notifications with the front-end part.
Finally, the last remaining function, among those mentioned above, is "updateElement" which allows to exploit the modification request of a haptic widget to change one or more of its three constituent elements, updating both the graphical and the haptic system.

7.4 Front-end

The front-end is the part of a project that the user can see and interact with.
As explained in the section 7.2, Qt provides QtQuick which includes QML. QML is a declarative language (similar to CSS and JSON) for designing user interface applications. QML document describes a hierarchical object tree. QML modules include primitive graphical building blocks (Rectangle, Image), modeling components, behavioral components (TapHandler, DragHandler, Animation), and more complex controls (Button, Slider). Therefore, starting from a basic element such as the rectangle, graphic and logical attributes can be added to it. These elements can be combined to build programs of different complexity.
A main element of this environment, which has been used a lot in the application, is the "MouseArea". A MouseArea is an invisible item that is typically used in conjunction with a visible item in order to provide mouse handling for that item. By effectively acting as a proxy, the logic for mouse handling can be contained within a MouseArea item. In our case, the finger emulates the mouse behavior by being tracked and identifying a press or release. Thanks to it, it is possible to update the state of an item when the events "press", "release", "enter" or "exit" are verified. The update takes place in the back-end part by calling the appropriate function, in the front-end, communicating the ID of the element and its state (1 = pressed, 0 = released).

The realized application consists of a swipe view of three pages. Each page has its own meaning and by interacting with them the different potentialities of that device emerge.
The pages are:

- Automotive dashboard
- Configuration panel
- Textures

In the main file of the application is defined the skeleton of it, i.e. what are the pages and in what order they are placed, but especially in this file is defined the "Backend" item that allows the front-end part to interact with the back-end part through appropriate signals.
The signal that moves in the direction of the graphic part is called elementStatusChanged, it communicates that the active haptic element, defined by its ID, has changed its physical state. The physical state refers to whether the impressed force has exceeded the set threshold, whether there has been a release, or simply whether haptic actuation has ended. The signal then can be decided whether to consider it or not based on the haptic element in action. The handler of this signal, defined inside "Backend" item, is constituted by a switch statement that divides the various cases according to the ID of the haptic element that has changed state. Each "case" will

perform a particular action in response to that signal and based on the state it is in.

Listing 7.6: elementStatusChanged signal handler.

```
Backend{
    id: backend
    onElementStatusChanged: {
        switch(swID)
        {
            case 1://autoSW1
                if(sts == 2)
                    page1.key1.keyAnimationPush.start()
                else if (sts == 1)
                    page1.key1.keyAnimationRelease.start()
                break;
            case 2://autoSW2
                if(sts == 2)
                    page1.key2.keyAnimationPush.start()
                else if (sts == 1){
                    page1.key2.keyAnimationRelease.start()
                break;
            case 3://autoSW3
                if(sts == 2)
                    page1.key3.keyAnimationPush.start()
                else if (sts == 1)
                    page1.key3.keyAnimationRelease.start()
                break;
            case 5://modButton
                if(sts == 2)
                    page2.modButton.keyPAnimation.start()
                else if (sts == 1)
                    page2.modButton.keyRAnimation.start()
                break;
        }
    }
}
```

Each page of the application is defined in different files called in the main one. Each file presents different graphical elements and different logics, which can be associated with other external files where an object has been structured so as to recall it without having to rewrite the code several times.
In the following sections the various pages are explained individually, explaining the objectives and how they were made.

7.4.1 Automotive dashboard page

Figure 7.10: Automotive dashboard.

This page simulates a possible automotive dashboard. The goal is to demonstrate how digital graphics combined with haptic sensations can be extremely similar to a mechanical product. In particular, an attempt is made to introduce haptics in the automotive environment for those products that do not usually provide haptic feedback, such as displays, or a try is done to demonstrate how an electrical component, such as the Niceclick, can give the same sensations as a mechanical one but with a longer life and greater programmability.
Another motivation that pleads the case for haptics in the automotive field is the ability to reduce driver distractions through feedback. Providing a tactile response to a given user action allows the user to not have to search with their eyes the item, but will know it has pressed it when it feel the sensation of pressure.
In the case of this application, a minimalist approach was sought, limiting the dashboard to three switches and two wheels. The device tries to emulate an air control system by acting on how air can flow through the vehicle and at what temperature.
The three switches, which can be seen in the middle of the image 7.10, are three tilting switches where pressing is possible only in the lower part.
In the page, these switches are based on a basic element called "AutoSW", called three times and placed in the appropriate positions. This element consists of a rectangle with different gradients of gray to create the right play of light that can make it look like a real switch. In the initialization phase we define which icon we want to insert

and the MouseArea at the point where the finger wants to be perceived.
The peculiarity of this graphic is that it is not static but is possible to interact with it. If a press is done with the right force in the appropriate area of the switch it will animate simulating to enter inside the dashboard. While if a release occurs the switch return immediately to its original position changing the color of the icon, in order to understand that the feature has been activated.
The effect is generated by moving the gradients as if the light were changing direction and adding light and dark points at the top and bottom respectively. If the switch is pressed again the icon will return to white "deactivating" the set functionality.
Having combined all the logic described in the previous sections and chapters it is possible to separate the concept of "touch" from that of "press", in this way if the finger is on the switch it will not be activated and also will not be activated even if insufficient force is applied.
When the finger is detected, the back-end part is notified of its presence by directing the signal to the external microcontroller that will open the force detection loop, which will not generate the effect until the threshold is exceeded or the finger is no longer present. If the threshold is exceeded the generation of the haptic effect is notified, which will reach the front-end part activating the animation (see the code in the 7.4 section).
When pressed, a tactile sensation is perceived that can be traced back to pressing a mechanical button, due to successive movements of the surface by the actuators.

Listing 7.7: Rocker switch initialization.

```
1   AutoSW {
2               id: sw1
3               x: 198
4               y: 158
5               iconSource: "Img/air.png"
6
7               MouseArea{
8                   anchors.topMargin: sw1.height / 1.7
9                   anchors.fill: sw1
10                  onPressed: backend.setElementStat(1,1)
11                  onExited: backend.setElementStat(1,0)
12              }
13  }
```

The wheels are placed on both sides of the dashboard, they can be slid to increase or decrease the temperature, which is shown in the two monitors at the bottom.
In this case, the haptic effect is not generated as a result of pressure, i.e. there is no sense of force, but if a finger is slid to a certain position. The position verification is done by making an evaluation of the movement start point and the current point, based on the MouseArea pixels of the wheel. Instead, the request that is made to the haptic handler is for immediate actuation (WDT_START) without doing force checks.
The perceived haptic effect is the classic one of a mechanical wheel which when a finger is slid over it the passage from one groove to another is perceived.
Moving the wheel upwards the temperature increases by half a degree at a time and if the wheel is scrolled downwards the temperature decreases, showing the changes on the screen. When the maximum or the minimum (32° and 16°) are reached, "HI"

or "LO" will be displayed on the screen and scrolling in that direction will no longer be allowed.
The animation in this case is given by alternating two different images of the wheel, making it appear to move forward and backward.

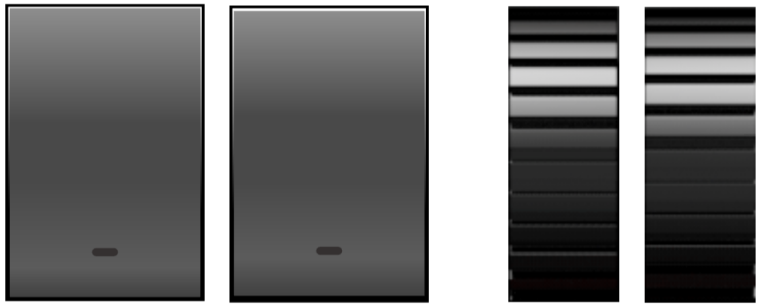

Figure 7.11: Automotive dashboard elements in different states.

7.4.2 Configuration panel page

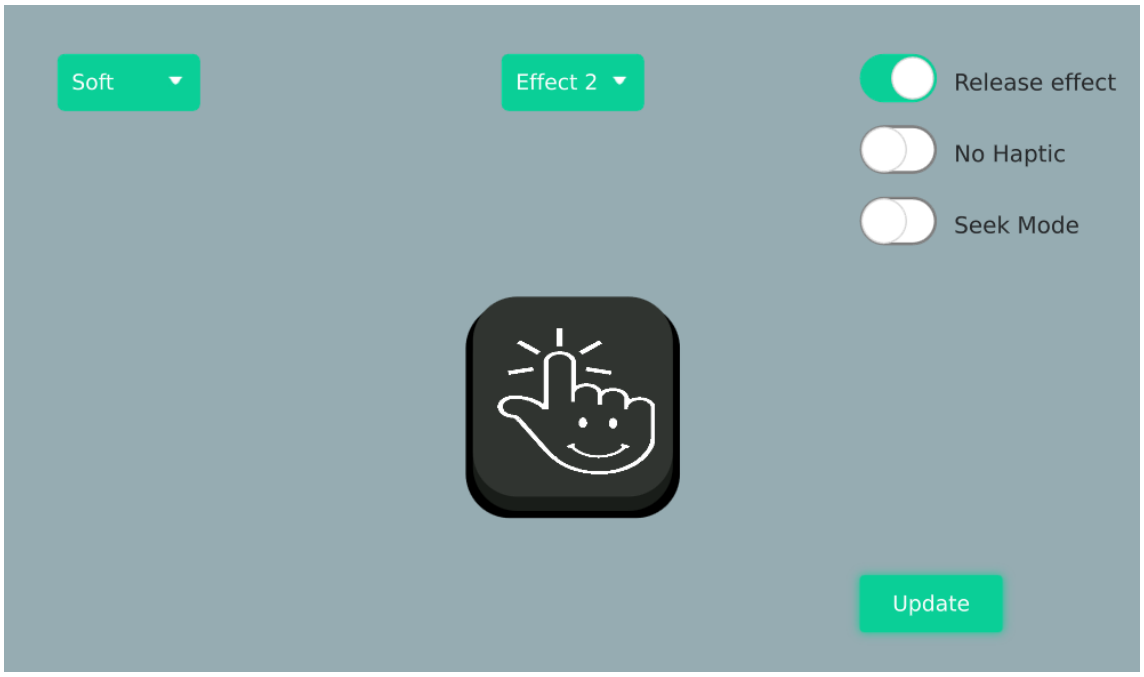

Figure 7.12: Configuration panel.

The second screen represents a configuration panel consisting of a single interacting button, whose haptic effect and force threshold can be modified.
The purpose of this page is to demonstrate the ductility of Niceclicks and their reconfigurability. Nowadays smartphones are able to generate vibrations in response to an action and other devices can give haptic sensations but have limitations in

varying the effects, do not present force readings or are very limited and above all can not be reconfigured by the user.
With this application the user can decide which effect to associate to a button, choosing it from a list, or can change the threshold of actuation by varying between a soft, medium or strong.
These two possibilities are selected from a drop-down menu and the state of the haptic element is updated by pressing the update button, which, upon detecting a change in the settings, changes its color warning the user. After the update the "updateElement" function (explained in the 7.3 chapter) is called from the "Backend" object, passing it the new data as values. The function sends, following the path already described, to the external board the modification request (WDT_MOD) of the widget with the selected ID. The corresponding widget will be modified and pressing again the button the new settings will be perceivable.
The button once confirmed the exceeding of the force threshold will activate the corresponding animation, downward movement, while if the release is detected the animation brings the button back to the original position. So even without removing the finger the button can be pressed several times and see that the graphics change according to the force applied and not by the presence or absence of the finger.
The animation consists of the upward or downward translation of the upper square, of which the button is made up, and a change of color tone to increase the three-dimensional effect of entering into the screen.

Figure 7.13: Control button pressed and released.

The function that manages the movement procedure, based on messages received from the external microcontroller, is the same that manages the automotive graphics switches, i.e. the function shown in the 7.3 section (in this case the element ID is 5).
In addition to being able to vary the characteristics of the button, it is possible to select different management modes. At the top right of the screen is possible to decide one of the following modalities:

- **Removal of the release effect**: When interacting with the button, only the effect during pressing will be perceivable but not the effect on release. The button will come back up, only when the finger is lifted from the screen, without feeling anything.

- **Removal of the haptic effect**: When interacting with the button the haptic management is not present. Thus, there will be no reading of the impressed force and there will be no actuation on either press or release. The downward movement of the button is caused by a simple touch on the screen and upward by removing the finger.
- **Seek mode**: Activating this mode displays an area around the button, highlighted by a darker color, on which a slight vibration is felt when the finger is slid. When the finger reaches the button, the vibration becomes more intense and a feedback as if there was a bounce is felt. In this way, without looking at the screen directly, it is possible to find the button only by sliding the finger on the screen. Once the button is identified the interaction with it is the standard one as per the settings.

Through these modes it is possible to highlight different features of the application. Removing the effect on release highlights how important it is, in order to emulate reality, to apply a sensation in both press and release action. Since there is a threshold of force, once returned above we would expect the button to be released, but not happening we have the feeling that something is missing because the tactile part is no longer congruent with the visual one. Consequently our perception will not be that of feeling a button.
In contrast, removing the entire haptic management highlights the difference in having feedback and not having it. If this mode is tried right after several interactions with the haptic mode on, the difference between a graphical button and a real one becomes apparent. Applying haptics allows to make the graphical button comparable to the real one.
Finally, the last mode try to support the concept of "haptic search" expressed in the 2.1 section. The search is what allows a haptic feature to be identified by guides. In this case the guide is a light vibration to an intense one in the area of interest, since the display has a smooth surface it is impossible to guide the finger through structural elements such as joints or ridges. This kind of management is useful when it is not possible to look at the screen while touching the button, allowing the identification in a simple way without moving the eyes and avoiding distractions, as in the case of driving a car[17].
The function that allows to follow the finger in the search area is based on the concept of the change of position of the mouse cursor, in this case the finger, inside a MouseArea. If the position of the finger changes inside the area, along the X or Y axis, and if the movement has been long enough, a function is called that allows to send a request, to the external microcontroller, for immediate actuation of a widget. If the position is outside of the button region, the effect generated corresponds to a slight vibration while as soon as the coordinates are on the button area the request generation refers to a different widget. The latter generates a stronger vibration for a limited time followed by another effect that gives a bounce feedback letting us know we have arrived in the correct area. A similar function and more details are explained in the next section.

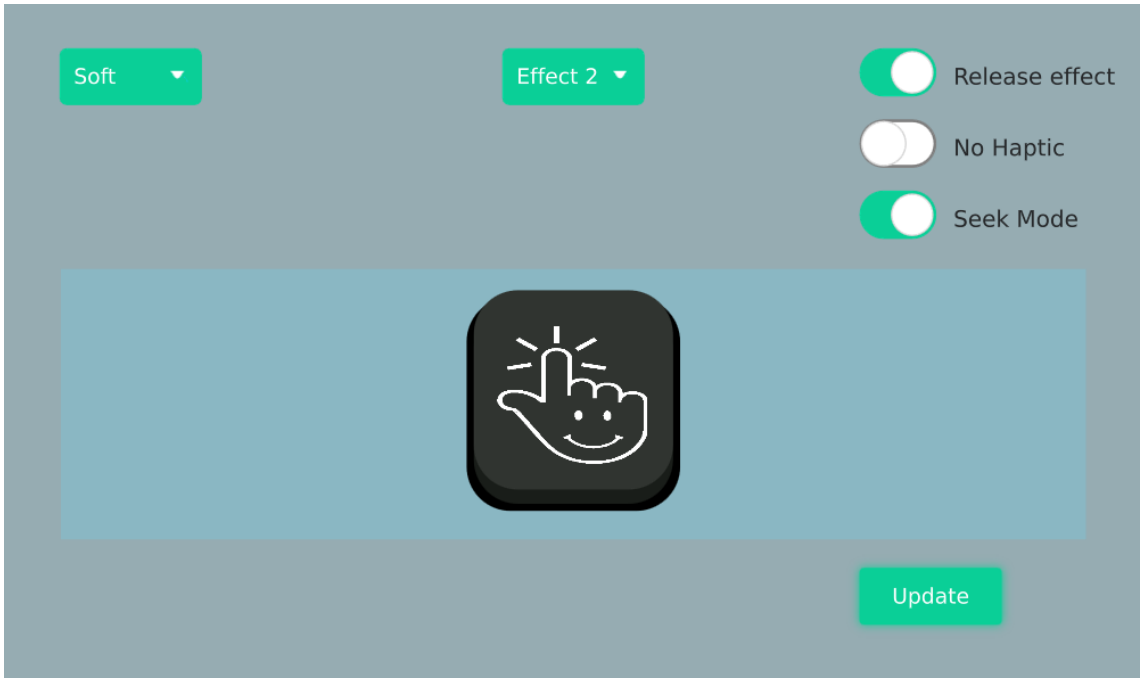

Figure 7.14: Configuration panel with seek mode active.

7.4.3 Textures page

In this page I have tried to demonstrate the ability to perceive the texture of an object.
Among the sensations that the haptic can provide, in addition to those already addressed previously, there is the perception of the material of which an object is made. The ability to perceive an object through touch can bring great benefits in various fields and contexts, for example it can be used to feel the fabric of a dress before buying it.
The problem in this case was related to the process of generating the haptic effect corresponding to a particular texture. Since there is not yet a precise method for their generation, especially if there is no reference accelerometer curve, there was no possibility to translate what is felt of an object into a sequence of pulsations. The solution was to find images that have a texture that could be associated to existing haptic effects. Then the opposite path was followed, starting from an effect I tried to understand what it could correspond to and then find a real reference to associate with it.
During the experimentation of haptic technology several effects have been realized creating a sort of library, starting from these and modifying some parameters such as repetition, time between samples and intensity, some effects were obtained that reminded patterns already felt.
In the application there are three images with which is possible to interact.

Figure 7.15: Texture page elements.

The sensation perceived by the first image, when the finger is slid, is the feeling of the finger passing between the various bars of the metal sheet, making perceive the difference in height between the area where there is a bar and the one where there is not.
The second image represents tiles and when the finger is slid a light vibration on the brown part is felt, getting the classic feeling of slight roughness of outdoor tiles. On the other hand, if the finger passes between the gray spaces the vibration becomes more intense, making the rough part more perceptible giving a feeling of resistance from the material when moving on it.
These two images to send the haptic request externally exploit the same principle addressed by the seek mode in the previous section (7.4.2). In the first image the MouseArea is extended over the whole surface and the sending of the request to the back-end part is done only when there is a sufficiently long movement along the X or Y axis.
The function that allows this logic is shown below.

Listing 7.8: Finger tracking.

```
1   MouseArea {
2               id: rodsArea
3               anchors.fill: parent
4               propagateComposedEvents: true
5               hoverEnabled:true
6
7               function mouseChanged() {
8                   if ((rodsArea.mouseX > oldPos1 + 10) || (rodsArea.mouseX < oldPos1 - 10)) {
9                       if(!rel)
10                          backend.setElementStat(6,1)
11                      oldPos1 = rodsArea.mouseX
12                  }
13              }
14              onEntered: {oldPos1 = rodsArea.mouseX; rel = false;}
15              onPositionChanged: {rodsArea.mouseChanged()}
16              onExited: {rel = true}
17  }
```

Three events are considered, the entry of the finger into the area, the change in the position of the finger and the exit of the finger from the area. The recognition of the change in position allows to call the mouseChanged function each time, which will check if the movement was sufficient to generate the haptic effect.
The logic of the second image works in the same way except that there are other

additional MouseArea, placed above the various guides between the tiles, both vertical and horizontal. In this case, every time the finger changes position, in addition to checking whether the movement was long enough, it is checked whether it has entered one of these areas. If the finger is detected on a guide the haptic request will be different, as a different ID will be referenced.
The last image depicts a trampoline, but in this case a texture is not perceived but the sensation it would give if we pressed on a trampoline. The perceived sensation is that of something elastic and soft, by pressing it will seem that the glass is soft and that the finger goes downwards and releasing it will feel the return upwards.
The choice to use an image rather than something that animates to this effect is caused by the instability of the response of the actuation, i.e. the messages received as a result of pressure and release were not reliable and upset the logic of the FSM that allows status update. Consequently, the situation in which the system was found was not predictable and therefore impossible to manage an animation on incorrect events.
The problem is due to the lack in the effect of a last pulse at maximum power, as mentioned in the section 4.1. Usually to keep the force measuring mechanism stable, in the array of pulsations, that form the haptic effect, the last pulsation is made at maximum power to re-establish the magnetic poles of the actuator. This procedure is mainly done when it also has an effect on the release, otherwise it could be avoided to insert it. The presence of this pulsation leads to an alteration in perception, as it prevails in the final effect at the expense of fluidity, a required factor in the case of this effect. Therefore in order to generate this effect in pressure and release it was decided to give up the feedback on the status of the widget.

Chapter 8

Conclusion

The purpose of this thesis was to provide haptic capabilities to a touch screen display and create a graphical interface that would have shown its potential.
The need to realize this device was aimed to test and improve the haptic technology provided to me, consisting of hardware and software, trying to expand the fields of application, integrating it into an external system, re-adapting the firmware.
Following this principle, the haptic board has become a kind of add-on for an external device, with CAN connection, from which it will receive management requests and transmit any status changes. This management is possible because it is the external system that initializes the haptic widgets and decides which operations to execute, following the established protocol.
Regarding the graphical part, the developed application allows to demonstrate how the addition of a haptic effect can guarantee greater realism to normal activities such as touching a display. It is possible to perceive shapes, figures, and thanks to force measurement another dimensional level to the interaction, compared to the typical 2D one, is added. Instead, with feedback generation, the system can lead to the four-dimensional experience. This application also allows to demonstrate how a mechanical element can be replaced by a digital one guaranteeing the same perception and avoiding the short durability of a mechanical component.
The realization of a multi-page application and the back-end management that have been done, will also allow future additions to it. Instead, possible improvements that can be made to the system concern the mechanism of force measurement, among which the accuracy and stability can be increased. Currently a new methodology of measurement is has being tested, which I personally helped to set up.
The choice to use a heterogeneous processor to manage the graphics and to control the haptic board was aimed to have the right computing power to devote to the GUI managing in parallel the CAN communication. Moreover, this choice was also aimed to test this technology evaluating if it was able, in a future application, to integrate within it also the haptic management controlled by the external microcontroller.
The result of the tests and the study confirmed that the Cortex M4 core would be able to independently manage the haptic part bringing benefits from a computational

point of view, while the Cortex A53 manages in parallel the graphics. The only change that should be taken in consideration would concern the control circuit, due to the lack of a peripheral in the M4 that instead is present in the other microcontroller. So, in the future, the entire system can be moved within this processor without performance compromise.
To conclude, the use of heterogeneous processors allows to meet the growing demand for embedded devices with high computing power and low power consumption while the addition of haptic sensations to a device opens new scenarios to different technologies, meeting the most diverse needs.

Appendix A

Startup and Closure Procedures

The startup procedure of the application is done via an icon on the display desktop. The icon is linked to a Bash script that allows to start the application executable after exporting some dependencies.

Listing A.1: Startup Bash script.

```
1    export QPEDIR
2    export QTDIR
3    export QT_QPA_PLATFORM="wayland-egl"
4    ./GUIHapDisplay
```

The application main as first action will activate the secondary core, the Cortex-M4. As described in the 5.2.2 section, through the Remoteproc component it is possible to start the remote core after loading in memory the firmware on which it will operate. Once started, the remote core will perform its boot routine enabling peripherals and OS, while waiting to establish communication with the main core.
The communication is started by loading the Linux Kernel module dedicated to the management of the RPMsg protocol (described in the 7.2 section). The initialization function of this module allows to create the channel in shared memory, the endpoint and notify the Cortex M4, which will create its own endpoint announcing the start of the service.
Finally, the graphical user interface is activated and as first operation creates the secondary thread that will open the system file representing the virtual serial, implemented by the Kernel module, for the communication between the two cores.
For what concern the haptic board, controlled by the CY8C4247AZI-L485, its startup is independent of that of the application, since it happens immediately when the board is connected to the power supply.
The startup procedure in this case consists in initializing the peripherals and the CAN communication, in order to receive possible requests from a master system.
Below there is an explanatory diagram of how the system starts up and how the data exchange takes place once all parties involved are ready to work.

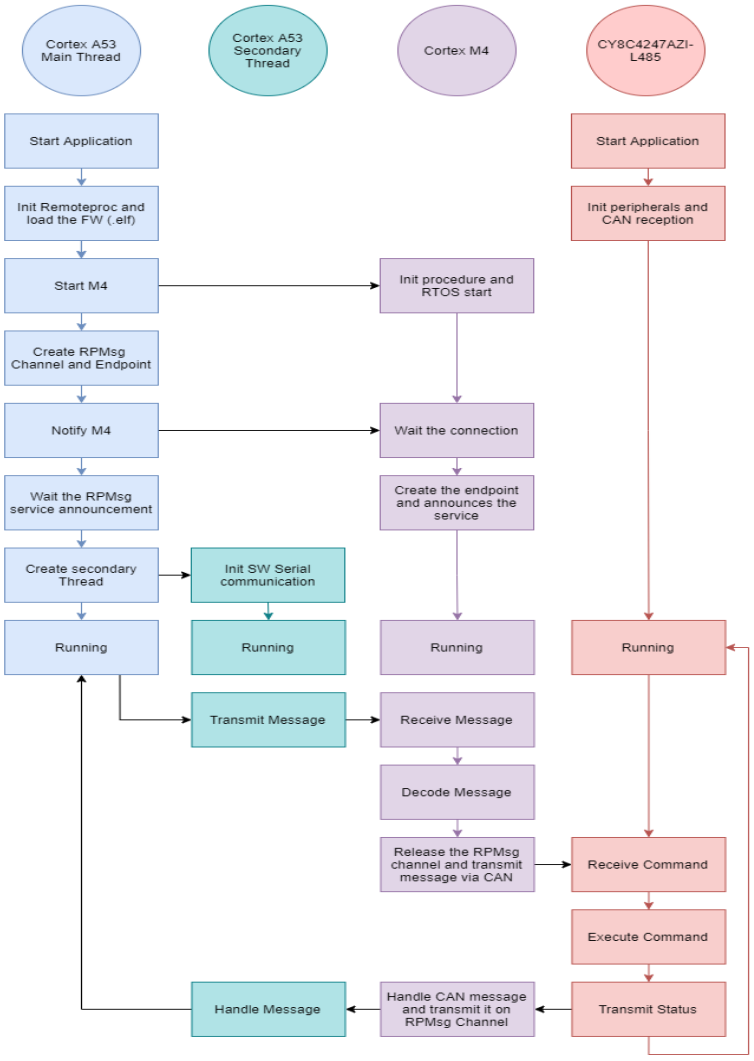

Figure A.1: Startup procedure.

The closing routine plays the opposite role of the one already reported.
It is initiated by pressing the icon in the upper right corner of the various application pages, which generates a signal connected to a slot in the back-end part containing the close routine. This routine prompts the secondary thread to send a message to the external microcontroller with the SYS_RESET configuration (see table 4.2), which allows to reboot the external board losing all the widgets initialization done by freeing the memory. Following the command sending, the secondary core will be shut down and the Kernel module will be unmounted, disabling the virtual serial and thus the communication via RPMsg. Confirmed the closure of all these elements, the secondary thread is closed, in order not to leave a zombie process in the operating system and finally closed the application itself.
Below there is a diagram that explains the communication process and the various phases faced in the closing process.

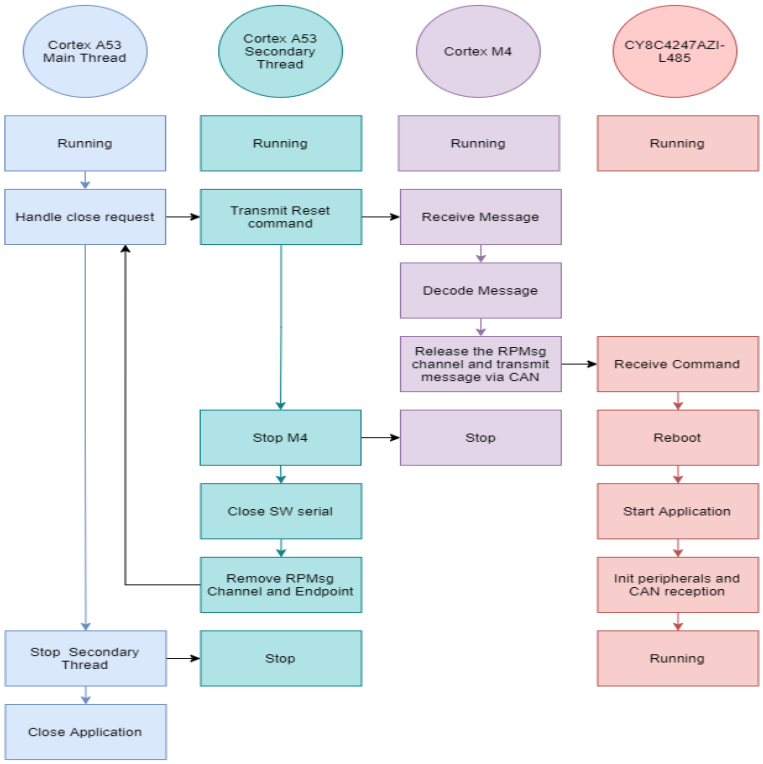

Figure A.2: Closure procedure.

Bibliography

[1] Okamura M. A. Culbertson H. Schorr B.S. «Haptics: The Present and Future of Artificial Touch Sensation». In: *Annual Reviews* 1 (May 2018), pp. 385–409 (cit. on pp. 7, 8).

[2] Carbon C. Breitshaft S. Clarke S. «A Theoretical Framework of Haptic Processing in Automotive User Interfaces and Its Implications on Design and Engineering». In: *Frontiers in Psychology* 10 (July 2019) (cit. on pp. 7, 8).

[3] N. Shin. *Touch Sensors vs. Mechanical Buttons*. 2014. URL: `www.silabs.com/community/blog` (cit. on p. 10).

[4] Eizo. *How can a screen sense touch? A basic understanding of touch panels*. URL: `www.eizo.com/library/basics/basic_understanding_of_touch_panel` (cit. on pp. 10, 11).

[5] Thomas. *All About Force Sensors*. URL: `https://www.thomasnet.com/articles/instruments-controls/all-about-force-sensors` (cit. on p. 12).

[6] *PSoC4: PSoC 4200L Datasheet*. Cypress. June 2018 (cit. on p. 21).

[7] NXP Semiconductors. *Three Reasons Why Embedded Heterogeneous Systems Are More Efficient*. 2016. URL: `https://www.nxp.com/company/blog/three-reasons-why-embedded-heterogeneous-systems-are-more-efficient:BL-3-REASONS-EMBEDDED-SYSTEMS-EFFICIENT` (cit. on p. 47).

[8] Uchiyama K. *Heterogeneous Multicore Processor Technologies for Embedded Systems*. Springer, 2014 (cit. on p. 47).

[9] *i.MX 8M Mini Applications Processor Reference Manual*. Rev. 3. NXP Semiconductors. Nov. 2020 (cit. on pp. 49, 52, 53, 72).

[10] Novak M. Aboelhassan M. Ondrej B. «Embedded multi-core systems for mixed-critical applications with RPMsg protocol based on xilinx ZYNQ-7000». In: *IEEE International Conference on Control System, Computing and Engineering* (July 2017) (cit. on pp. 55, 57).

[11] Hancock J. Anjum E. *Introduction to OpenAMP Library*. URL: `https://www.openampproject.org/docs/whitepapers/Introduction_to_OpenAMPlib_v1.1a.pdf` (cit. on p. 56).

[12] OpenAMP. *RPMsg Messaging Protocol*. URL: https://github.com/OpenAMP/open-amp/wiki/RPMsg-Messaging-Protocol (cit. on p. 56).

[13] *MCP2517FD Datasheet*. Microchip Technology Inc. 2018 (cit. on pp. 64, 65).

[14] Amazon Web Services Inc. *FreeRTOS User Guide*. 2021. URL: https://docs.aws.amazon.com/freertos/latest/userguide (cit. on p. 67).

[15] NXP Semiconductors. *RPMsg-Lite User's Guide*. 2020. URL: https://nxpmicro.github.io/rpmsg-lite (cit. on pp. 68, 69).

[16] Le Yang Peng Yang and Xin Guo. «Research on Embedded Data Display Unit Based on CAN Bus». In: *2009 4th IEEE Conference on Industrial Electronics and Applications* (May 2009), pp. 3498–3501 (cit. on pp. 82, 84).

[17] Bastian Pfleging Dagmar Kern. «Supporting Interaction Through Haptic Feedback in Automotive User Interfaces». In: *Interactions* 20 (Mar. 2013), pp. 16–21 (cit. on p. 98).

Acknowledgements

I would like to thank all the people that supported and accompanied me in my university career and in the realization of this thesis.
Foremost, the entire TRAMA Engineering team for allowing me to do my thesis with them, for helping me when I had doubts or perplexities, for giving me freedom in my choices and organization. Thank you so much you have taught me a lot and made me feel welcomed.
A special thanks goes to my company supervisor Ing. Flavio Cerruti for giving me advice, stimulating my imagination and believing in me.
Additionally, I wanted to thank my academic supervisor Prof. Luciano Lavagno for the support and availability given to me in the realization of the thesis.
Thanks to all my teammates without whom studying for exams and preparing for projects during the lockdown period would have been tough.

Furthermore, I want to thank my family for always believing in me, in my dreams and choices. You have allowed me to focus on studying without worrying about anything else by supporting and pushing me. Special thanks to my little brother Samuele for always being by my side.
Finally, my biggest thanks to my beloved Eleonora, without you my thoughts would have no order and my life would be monotonous. Thank you for all your smiles, support and especially to your forbearance.

Grazie Nonno per avermi insegnato che il lavoro duro e la dedizione sono le uniche strade che portano al risultato, e che nella vita bisogna farsi valere rimanendo sempre rispettosi. Dedico a te questa Tesi.