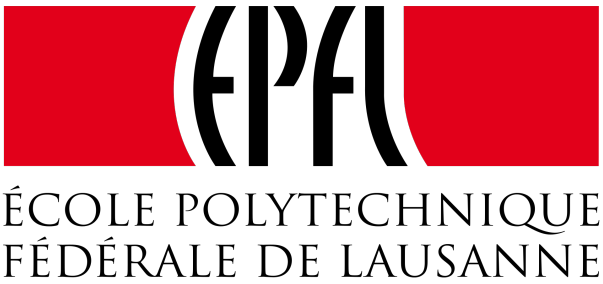


China Hardware Innovation Camp

Semester project report

HUM - 498

Chloe Dickson - chloe.dickson@epfl.ch
Matteo Yann Feo - matteo.feo@epfl.ch
Simone Aron Sanso - simone.sanso@epfl.ch

June 8, 2018

Contents

1 Introduction **2**

2 Context **3**
2.1 Team description . . . 3
2.2 Product description . . . 3
2.2.1 Interaction Design . . . 4
2.3 Work with ECAL designers . . . 4
2.4 Business + Analysis of potential users + Customer journey . . . 4
2.5 SHS research project and insights . . . 4

3 Electronic engineering **5**
3.1 Hardware . . . 5
3.1.1 Initial breadboard solution & development kit selection . . . 5
3.1.2 ESP32 Capabilities & final breadboard solution . . . 7
3.1.3 Electronic characteristics . . . 10
3.2 Main PCB . . . 12
3.2.1 Features & schematics . . . 12
3.2.2 Bill of materials (BOM) . . . 16
3.2.3 Main PCB design . . . 17
3.3 Detachable module . . . 18
3.3.1 Functionality . . . 18
3.3.2 Schematic . . . 18
3.3.3 Detachable module design . . . 18

4 Mechanical engineering - Chloe **19**
4.1 Blackbox design . . . 19
4.2 Soft PCB . . . 21
4.2.1 Washability . . . 21
4.2.2 Manufacturing options . . . 21
4.3 Soft sensors . . . 22
4.3.1 ESP32 integrated capacitive touch sensors . . . 22
4.3.2 Prototyping design . . . 22
4.3.3 Experimenting with soft sensors . . . 23
4.4 Hard <-> Soft interfaces . . . 23
4.4.1 Next steps . . . 23

5 Firmware engineering - Chloe **25**
5.1 Firmware architecture . . . 25
5.1.1 Choice of coding environment . . . 26
5.2 Description of Functions . . . 26
5.2.1 NAPaC_FW . . . 26
5.2.2 Messages . . . 27
5.2.3 WiFi connectivity and Server communication . . . 28
5.2.4 Games . . . 30
5.2.5 Capacitive touch sensors . . . 31

5.2.6 LEDs . . . 33
5.2.7 Capacitive touch sensors . . . 33
5.2.8 Sound . . . 33
5.3 Next steps - functions to be implemented . . . 34
5.3.1 WiFi save Smartconfig in EEPROM . . . 34
5.3.2 Sleep functions . . . 34
5.3.3 Sound settings and implementing a proper loudspeaker . . . 34
5.3.4 Capacitive sensors calibration and touch/squeeze recognition . . . 34
5.3.5 Error checking . . . 34

6 Software engineering **35**
6.1 The communication protocol definition . . . 36
6.1.1 An example pipeline . . . 38
6.2 The TCP Server . . . 40
6.2.1 The main component: Server manager . . . 42
6.2.2 The Message component . . . 44
6.2.3 The Database Manager component . . . 45
6.3 The Android APP for parents . . . 46
6.3.1 A client within the system . . . 47
6.3.2 WiFi SmartConfig for trivial connectivity setup . . . 49
6.3.3 QR-codes to be scanned . . . 51
6.3.4 Android Activity management . . . 52
6.4 Next steps for the software . . . 55

7 Conclusion **56**

A List of included contents **57**

B Main PCB schematic **59**

C Defined messages for the communication protocol **60**

D Android activities inherited methods **64**

E Work distribution and personal conclusions **65**

1 Introduction

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc ac ultricies felis. Donec lobortis dignissim velit, et rhoncus urna volutpat at. In hac habitasse platea dictumst. Nulla finibus ullamcorper eleifend. Suspendisse euismod placerat rhoncus. Donec rutrum eu lacus nec viverra. Maecenas luctus, dui nec scelerisque dictum, nisi lorem hendrerit ex, sed faucibus nulla metus ultricies ipsum. Nam luctus magna velit, ac luctus tortor porttitor ut. Aenean fringilla velit et magna tempus egestas.

Fusce commodo ut turpis sed hendrerit. Fusce eget quam at sapien facilisis volutpat et sed ipsum. Sed ac bibendum nibh. Integer in enim tempus, tempus massa ac, cursus lacus. Integer tincidunt diam eget ipsum aliquet tincidunt. Mauris et bibendum metus, non dignissim nulla. Curabitur tristique tortor eu dolor sagittis, vitae mattis ex tempor. Maecenas ut magna diam. Pellentesque porta ultricies varius. Mauris egestas ante efficitur fringilla suscipit. Praesent et suscipit urna. Phasellus ut suscipit velit.

Vivamus consequat nisl sed tristique gravida. Phasellus egestas gravida dui sed accumsan. Sed sit amet metus et sapien condimentum faucibus. Sed dapibus ut mauris a aliquet. Morbi est quam, tempor sed dapibus vel, consequat et est. Vestibulum vestibulum nunc sed libero elementum, non luctus lacus accumsan. Duis sed lacus massa.

Figure 1: Old but GOLD !

2 Context

In this first section of the report, we will aim to provide the reader with a complete overview for the background of the project itself. Before introducing in detail every engineering aspect that lead the development, we will

2.1 Team description

CHIC - Team descrition, goals, fieldwork

Marjane Amara
Industrial Design

Chloe Dickson
Firmware Engineering
Mechanical Engineering

Estelle Geneux
Business

Matteo Yann Feo
Software Engineering

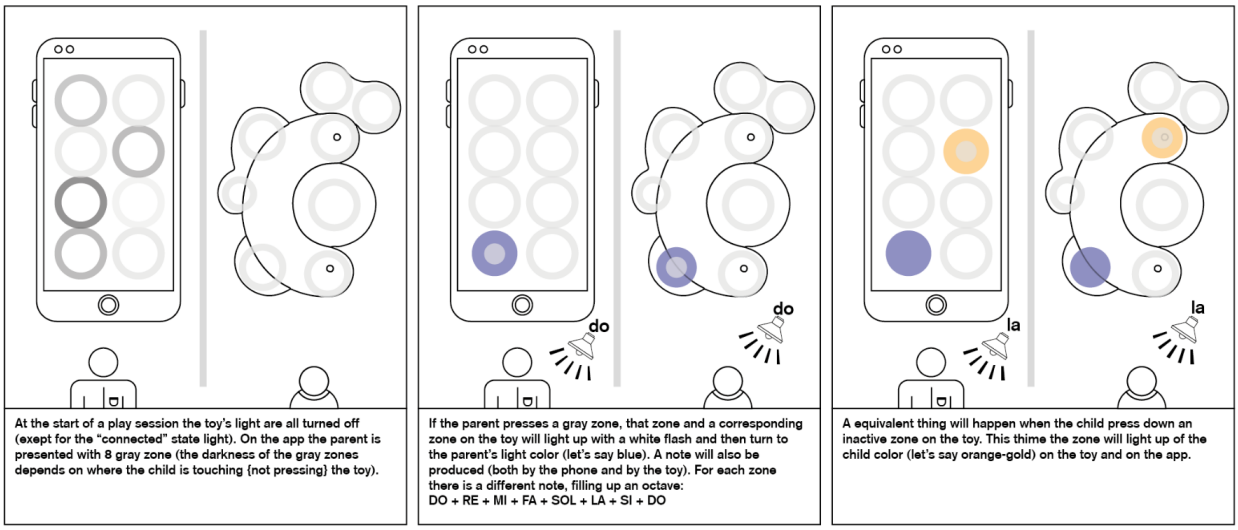
Simone Sanso
Electronic Engineering
Firmware Engineering

Luca Sassoli de Bianchi
User Experience

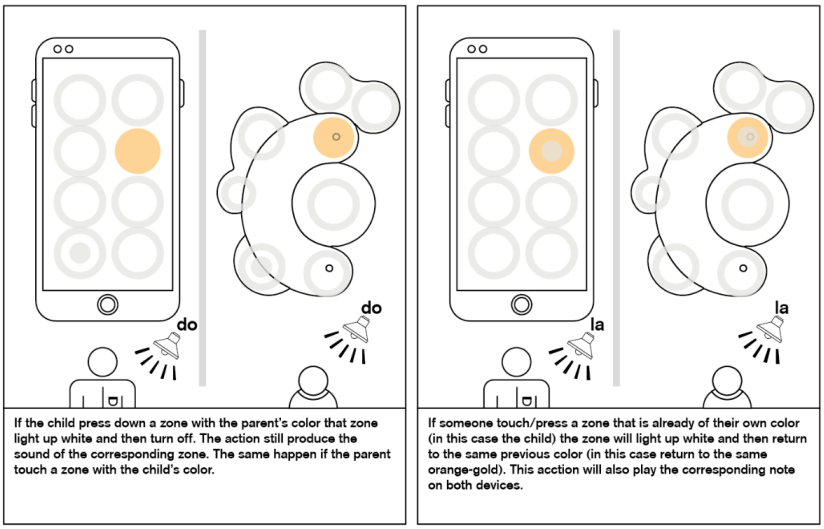
2.2 Product description

Have you ever felt like spending more time with your children? Toygether is a plush toy for kids that connects to their parent's smartphone. It enables parents to interact and play games with their children, even when far apart. Our plush toy features colourful lights and sounds to enhance the playing experience with children from 2 to 5 years old. With ever increasing working days and commute times, parents might have less time to spend with their loved ones. Toygether aims to bridge this gap by providing a screen-less interaction platform, enabling young children and parents to play together, even when they can't physically be together.

2.2.1 Interaction Design



(a)



(b)

Figure 2: Interaction design for child/parent game session

2.3 Work with ECAL designers

2.4 Business + Analysis of potential users + Customer journey

2.5 SHS research project and insights

3 Electronic engineering

In this section will be presented the work related on the electronic field.

First, the hardware subsection will explore the development kit selection, before highlighting the ESP32 capabilities and its electronic characteristics.

3.1 Hardware

This subsection will explain how the project was started, to be able to rapidly prototype the main functions to allow interacting with the electronics of the plush toy.

3.1.1 Initial breadboard solution & development kit selection

To quickly start prototyping the firmware of the plush toy, a breadboard solution was needed as fast as possible. To this end, several development kits have been explored.

The first breadboard solution is presented figure 3.

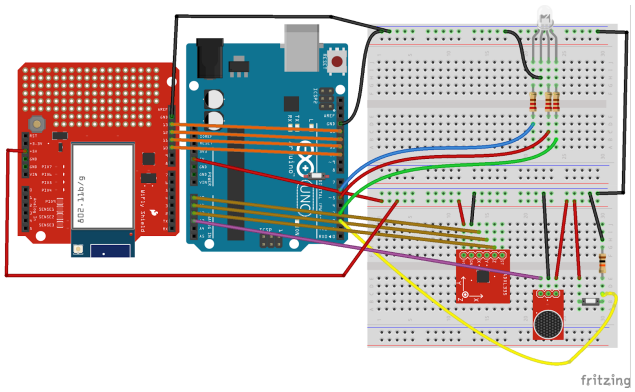


Figure 3: First breadboard solution with Arduino (realized with *fritzing*)

This early breadboard solution was equipped of the following electronic components :

- an Arduino UNO (Rev3) - ICSP board, based on the ATmega328 microcontroller,
- an Arduino WiFly Shield, to equip the system with 802.11b/g wireless connection,
- a triple axis accelerometer breakout - ADXL335, for sensing three-dimensional accelerations,
- a breakout board for Electret microphone, with a 100x opamp to amplify the recorded sounds,
- a tri-color LED with red, green and blue inside, for the visual interaction between the plush toy and the child,
- and a generic pushbutton, to be able to have a tactile interaction, again between the plush toy and the child.

At that stage of ideation, the plush toy was supposed to record the child and allow the parents or the caregivers listening to the kid.

Figure 4 illustrates the block diagram related to this early breadboard solution.

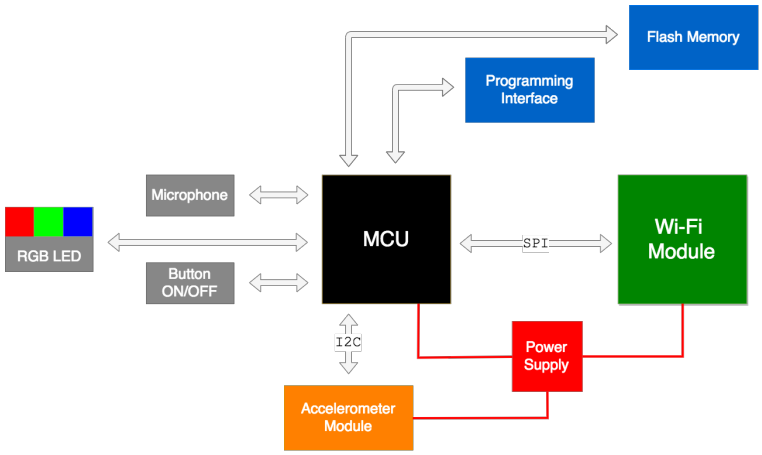


Figure 4: First breadboard solution with Arduino

As it can be seen from figure 4, the microcontroller would communicate with the Wi-Fi module via the SPI protocol and with the accelerometer module via I2C.There would have been a flash memory to store the voice recordings of the child. Obviously, the system would have also been equipped of a tri-color LED, a pushbutton and a microphone, as previously illustrated 3.

After seeking for related work on toys with this recording feature, it has been understood that it was not only too invasive for the child's privacy, but also very risky on the cyber-security domain (McReynolds et al. 2017). Therefore, the recording feature has been abandoned. The electronics in care of the interaction between the plush toy and the child have also been largely enhanced, through the use of capacitive soft sensors (more details section 4).

The final development kit that has been used is the ESP32-DevKitC, shown figure 5.

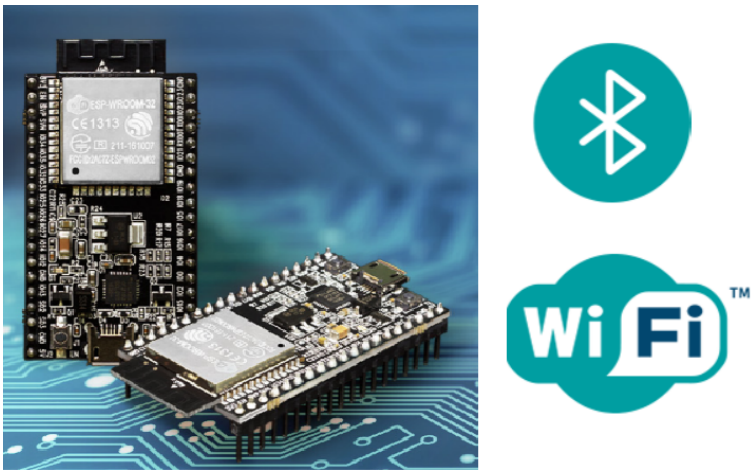


Figure 5: ESP32-DevKitC

3.1.2 ESP32 Capabilities & final breadboard solution

The reasons of this choice of development kit can be even more clearly seen with the following comparison.

SPECS/BOARD	ESP32	ESP8266	ARDUINO UNO
Number of Cores	2	1	1
Architecture	32 Bit	32 Bit	8 Bit
CPU Frequency	160 MHz	80 MHz	16 MHz
WiFi	YES	YES	NO
BLUETOOTH	YES	NO	NO
RAM	512 KB	160 KB	2 KB
FLASH	16 MB	16 MB	32 KB
GPIO PINS	36	17	14
Busses	SPI, I2C, UART, I2S, CAN	SPI, I2C, UART, I2S	SPI, I2C, UART
ADC Pins	18	1	6
DAC Pins	2	0	0

Figure 6: Comparison between ESP32, ESP8266 and Arduino UNO boards capabilities (Future-Electronics 2016)

Various capabilities of the ESP32-DevKitC are developed here below.

CPU architecture The ESP32-DevKitC has a 32-bit double core CPU, one dedicated for the wireless (Wi-Fi and Bluetooth) and the other for the logic and control.

GPIO pins There are up to 16 channels of PWM-capable pins, for dimming LEDs or controlling motors. Up to 10 channels feature capacitive touch sensors.

UART There are two UART interfaces to load code serially, feature flow control and support IrDA (Infrared Data Association).

I2C, SPI, I2S There are two I2C and four SPI interfaces to hook up all types of sensors and peripherals, plus two I2S interfaces for connecting digital audio devices.

Analog-to-Digital Converter (ADC) With up to 18 channels of 12-bit signals, the ADC range can be set, in firmware, to either 0-1V, 0-1.4V, 0-2V, or 0-4V.

Digital-to-Analog Converter (DAC) There are two 8-bit DACs to produce true analog voltages.

Thus, this development kit allows rapid prototyping and flexibility with its Wi-Fi and BLE (Bluetooth Low Energy) connectivity. The firmware is easy to program, thanks to its Arduino IDE (integrated development environment) compatibility and its peripherals.

Figure 7 shows the key components of the ESP32-DevKitC.

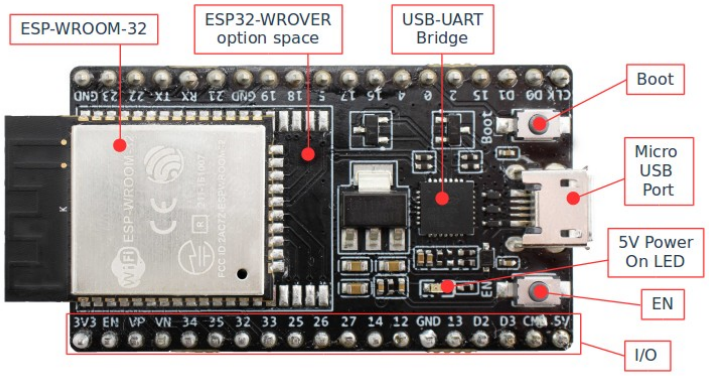


Figure 7: ESP32-DevKitC functional overview (Espressif-Systems 2016a)

Its micro USB port and the boot button allow an easy connectivity to program the ESP32 microcontroller, embedded inside the ESP-WROOM-32 unit. The development kit already includes a USB-UART bridge, translating the program coming from the computer, via USB, to the microcontroller in UART protocol. The EN pushbutton "ENables" to reset the microcontroller, starting again the program from the first line of code. The ESP32-WROVER optional space has not been used, since it is a different microcontroller chip, longer than the ESP-WROOM-32.

The large number of available input/output pins were very useful to develop our final breadboard solution, shown figure 8.

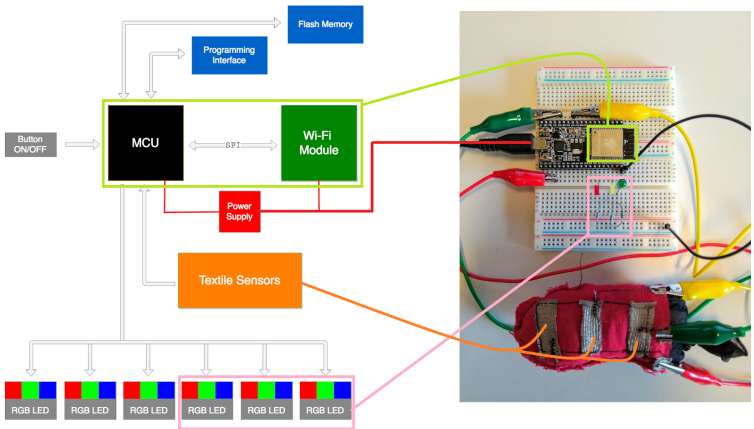


Figure 8: Final breadboard solution using ESP32-DevKitC

The main addition with the initial breadboard solution, illustrated figure 3, is the presence of capacitive textile sensors instead of pushbuttons. This replacement definitely enhanced the user experience of the child, eventually interacting by simply touching the paws of the plush toy. At that stage of the project, the shape was not defined yet and could have represented any animal, like other mainstream teddybears.

Another replacement on the hardware side was to remove the microphone, initially supposed to record the child's voice. Instead, a miniature speaker (figure 9) has been added, to be able to play sounds for every LED that lights up.

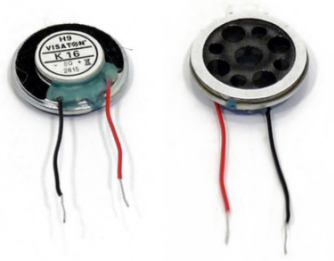


Figure 9: Miniature speaker of 2 grams, 16 mm diameter and 3.5 mm depth (Visaton 2015)

The tri-color LEDs that have been used for our final product are the APA102C (figure 10).

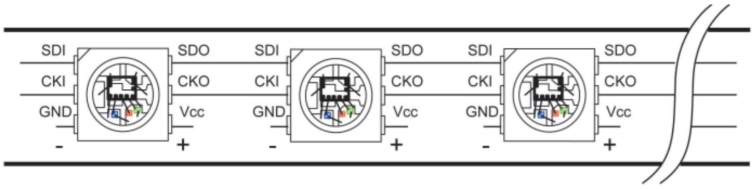


Figure 10: LED strip of tri-color RGB LEDs APA102C (LuxaLight 2015)

SDI/SDO : Serial Data Input/Output
CKI/CKO : Clock Input/Output
GND : Ground – VCC : 5V

As illustrated above, the main advantage of these LEDs is their capability of commanding a strip of N LEDs with only 2 command signals (SDI and CKI). More information on their functioning will be provided subsection 5.2.6.

3.1.3 Electronic characteristics

Now will be described the different power modes, available on the ESP32 microcontroller, allowing a reduction of power consumption depending on the state of the ESP32 microprocessor.

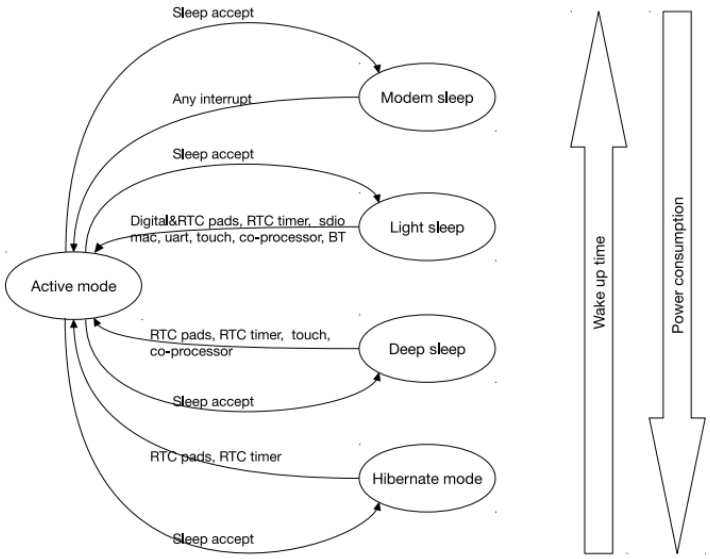


Figure 11: Power modes of the ESP32 microprocessor (Espressif-Systems 2018b)

Power mode	Description	Power consumption
Active (RF working)	Wi-Fi Tx packet 14 dBm ~ 19.5 dBm	Please refer to Table 8 for details.
	Wi-Fi / BT Tx packet 0 dBm	
	Wi-Fi / BT Rx and listening	
Modem-sleep	The CPU is powered on.	Max speed 240 MHz: 30 mA ~ 50 mA
		Normal speed 80 MHz: 20 mA ~ 25 mA
		Slow speed 2 MHz: 2 mA ~ 4 mA
Light-sleep	-	0.8 mA
Deep-sleep	The ULP co-processor is powered on.	150 μA
	ULP sensor-monitored pattern	100 μA @1% duty
	RTC timer + RTC memory	10 μA
Hibernation	RTC timer only	5 μA
Power off	CHIP_PU is set to low level, the chip is powered off	0.1 μA

Figure 12: Power consumption by power modes (Espressif-Systems 2018a)

Mode	Min	Typ	Max	Unit
Transmit 802.11b, DSSS 1 Mbps, POUT = +19.5 dBm	-	240	-	mA
Transmit 802.11b, OFDM 54 Mbps, POUT = +16 dBm	-	190	-	mA
Transmit 802.11g, OFDM MCS7, POUT = +14 dBm	-	180	-	mA
Receive 802.11b/g/n	-	95 ~ 100	-	mA
Transmit BT/BLE, POUT = 0 dBm	-	130	-	mA
Receive BT/BLE	-	95 ~ 100	-	mA

Figure 13: Radio frequency power consumption specifications (Espressif-Systems 2018a)

From figures 11, 12 and 13 have been extracted the values of power consumption for the estimation of the battery life.

- Powering 9 LEDs APA102C, drawing 20 mA of constant current each, requires to output $I_{LEDs} = 9 * 20 = 180$ mA (LuxaLight 2015).
- The maximum power of the miniature speaker is 1 W, which corresponds to $I_{speaker} = 200$ mA of current consumption for a voltage output of 5 V.
- During Wi-Fi transmissions, the microcontroller requires from 180 to 240 mA. Considering the largest possible current consumption : $I_{WiFi} = 240$ mA.
- When the ESP32 enter the *deep-sleep* mode, the microcontroller only requires $I_{DeepSleep} = 0.15$ mA.

From these values, the maximal and minimal current consumption values are:

$$I_{max} = I_{LEDs} + I_{speaker} + I_{WiFi} = 180 + 200 + 240 = 620 \text{ mA} \tag{1}$$

$$I_{min} = I_{DeepSleep} = 0.15 \text{ mA} \tag{2}$$

Using an *Energizer* EN22 non-rechargeable battery (figure 14), with a 9V nominal voltage and a battery capacity of $Cap = 625$ mAh, the maximal and minimal battery lifetime have been estimated as follows.

$$T_{max} = \frac{Cap}{I_{min}} = \frac{625}{0.15} = 4166.67\, h = 173.6 \text{ days} = 5.8 \text{ months} \tag{3}$$

$$T_{min} = \frac{Cap}{I_{max}} = \frac{625}{620} = 1.0\, h \tag{4}$$

These numbers need to be considered as a range of battery lifetime. The LEDs have been considered all simultaneously lighted on with maximal brightness and the speaker on maximal power. Also, the Wi-Fi has been considered continuously transmitting.

In the prototyping phase, an *Energizer* NH22-175 rechargeable battery will be preferred, to ease the testing procedure. With a capacity of 175 mAh, $T_{max} = 6.9$ weeks and $T_{min} = 16.9$ min.

Figure 14: NH22-175 rechargeable battery (Energizer 2018b) on the left and EN22 non-rechargeable battery (Energizer 2018a) on the right

3.2 Main PCB

In this section will be presented the work related to the design of the main PCB (printed circuit board), where the main electronics of the plush toy will be integrated.

3.2.1 Features & schematics

The first step, to begin with, was to allocate each electronic peripheral with the pins of the ESP-WROOM-32. Figure 15 summarises the pin organisation in a table, where every peripheral corresponds to a colour. The "type" column shows if the pin is used for power ('P'), input ('I') or output ('O') signals. The pin number is indicated by the 'No.' column.

Name	No.	Type	Function	NAPaC function
GND	1	P	Ground	
3V3	2	P	Power supply	
EN	3	I	Chip enable (active low) -> RESET_n	
SENSOR_VP	4	I	GPIO36, SENSOR_VP, ADC_H, ADC1_CH0, RTC_GPIO0	
SENSOR_VN	5	I	GPIO39, SENSOR_VN, ADC1_CH3, ADC_H, RTC_GPIO3	
IO34	6	I	GPIO34, ADC1_CH6, RTC_GPIO4	DI2
IO35	7	I	GPIO35, ADC1_CH7, RTC_GPIO5	CI2
IO32	8	I/O	GPIO32, XTAL_32K_P (32.768 kHz crystal oscillator input), ADC1_CH4, TOUCH9, RTC_GPIO9	
IO33	9	I/O	GPIO33, XTAL_32K_N (32.768 kHz crystal oscillator output), ADC1_CH5, **TOUCH8**, RTC_GPIO8	Touch 8
IO25	10	I/O	GPIO25, DAC_1, ADC2_CH8, RTC_GPIO6, EMAC_RXD0	
IO26	11	I/O	GPIO26, DAC_2, ADC2_CH9, RTC_GPIO7, EMAC_RXD1	
IO27	12	I/O	GPIO27, ADC2_CH7, **TOUCH7**, RTC_GPIO17, EMAC_RX_DV	Touch 7
IO14	13	I/O	GPIO14, ADC2_CH6, **TOUCH6**, RTC_GPIO16, MTMS, HSPICLK, HS2_CLK, SD_CLK, EMAC_TXD2	Touch 6
IO12	14	I/O	GPIO12, ADC2_CH5, **TOUCH5**, RTC_GPIO15, MTDI, HSPIQ, HS2_DATA2, SD_DATA2, EMAC_TXD3	Touch 5
GND	15	P	Ground	
IO13	16	I/O	GPIO13, ADC2_CH4, **TOUCH4**, RTC_GPIO14, MTCK, HSPID, HS2_DATA3, SD_DATA3, EMAC_RX_ER	Touch 4
SHD/SD2*	17	I/O	GPIO9, SD_DATA2, SPIHD, HS1_DATA2, U1RXD	
SWP/SD3*	18	I/O	GPIO10, SD_DATA3, SPIWP, HS1_DATA3, U1TXD	
SCS/CMD*	19	I/O	GPIO11, SD_CMD, SPICS0, HS1_CMD, U1RTS	
SCK/CLK*	20	I/O	GPIO6, SD_CLK, SPICLK, HS1_CLK, U1CTS	
SDO/SD0*	21	I/O	GPIO7, SD_DATA0, SPIQ, HS1_DATA0, U2RTS	
SDI/SD1*	22	I/O	GPIO8, SD_DATA1, SPID, HS1_DATA1, U2CTS	
IO15	23	I/O	GPIO15, ADC2_CH3, **TOUCH3**, MTDO, HSPICS0, RTC_GPIO13, HS2_CMD, SD_CMD, EMAC_RXD3	Touch 3
IO2	24	I/O	GPIO2, ADC2_CH2, **TOUCH2**, RTC_GPIO12, HSPIWP, HS2_DATA0, SD_DATA0	Touch 2
IO0	25	I/O	GPIO0, ADC2_CH1, TOUCH1, RTC_GPIO11, CLK_OUT1, EMAC_TX_CLK	
IO4	26	I/O	GPIO4, ADC2_CH0, **TOUCH0**, RTC_GPIO10, HSPIHD, HS2_DATA1, SD_DATA1, EMAC_TX_ER	Touch 1
IO16	27	I/O	GPIO16, HS1_DATA4, U2RXD, EMAC_CLK_OUT	
IO17	28	I/O	GPIO17, HS1_DATA5, U2TXD, EMAC_CLK_OUT_180	
IO5	29	I/O	GPIO5, VSPICS0, HS1_DATA6, EMAC_RX_CLK	
IO18	30	I/O	GPIO18, VSPICLK, HS1_DATA7	
IO19	31	I/O	GPIO19, VSPIQ, U0CTS, EMAC_TXD0	
NC	32	-	-	
IO21	33	I/O	GPIO21, VSPIHD, EMAC_TX_EN	Piezo
RXD0	34	I/O	GPIO3, U0RXD, CLK_OUT2	
TXD0	35	I/O	GPIO1, U0TXD, CLK_OUT3, EMAC_RXD2	
IO22	36	I/O	GPIO22, VSPIWP, U0RTS, EMAC_TXD1	DI1
IO23	37	I/O	GPIO23, VSPID, HS1_STROBE	CI1
GND	38	P	Ground	
Color code	Signification	I/O		
	By default	-		
	8 Touch sensor	I		
	9 LEDs	O		
	Piezo	O		
	Flash program	I		
	Reset button	I		

Figure 15: ESP-WROOM-32 pins allocation

Once this work was done, the electronic schematic of the main PCB could be started. Based on the schematic of the ESP32-DevKitC (Espressif-Systems 2017), allowing existing functionality to be kept in the design of the main PCB.

The main features of the design will be explored below.

ESP32 Module As previously mentioned, the ESP-WROOM-32 is the main module of the main PCB, since the intelligence of the plush toy is stored on this component. C21 and C22 are decoupling capacitors, cancelling eventual small fluctuations of the input power (3V3).

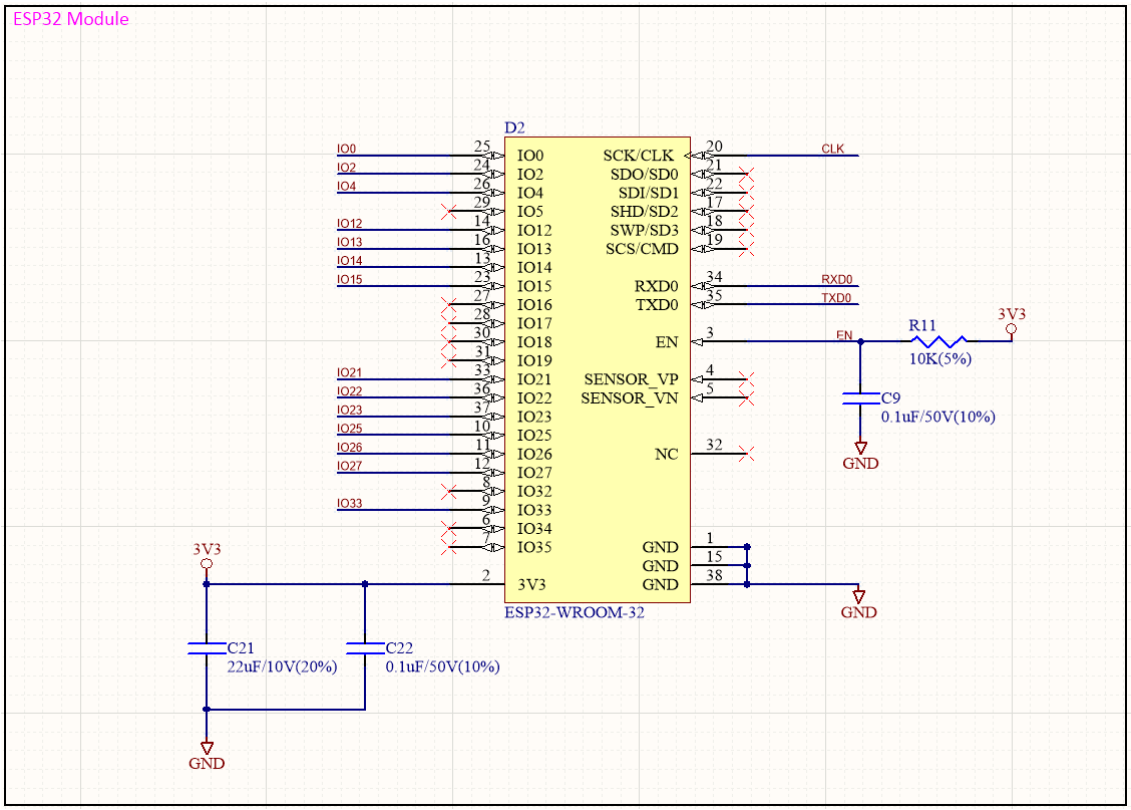


Figure 16: ESP32 module schematic

Power supply As can be seen figure 17, two different constant voltages are needed in our electronic assembly. The LEDs APA102C require an input voltage of 5V, while the rest of the electronic components need 3V3. Therefore, the ACT4060A step down converter has been used, with 2A of output current (Active-Semi 2009) in order to supply enough power for all the 9 LEDs. On another hand, the LM1117 low-dropout linear regulator has been used to ensure a 3V3 voltage translation with 800 mA of output current (Texas-Instruments 2016), either from the 5V voltage coming from the micro-USB, or from the outputted voltage of the ACT4060A component.

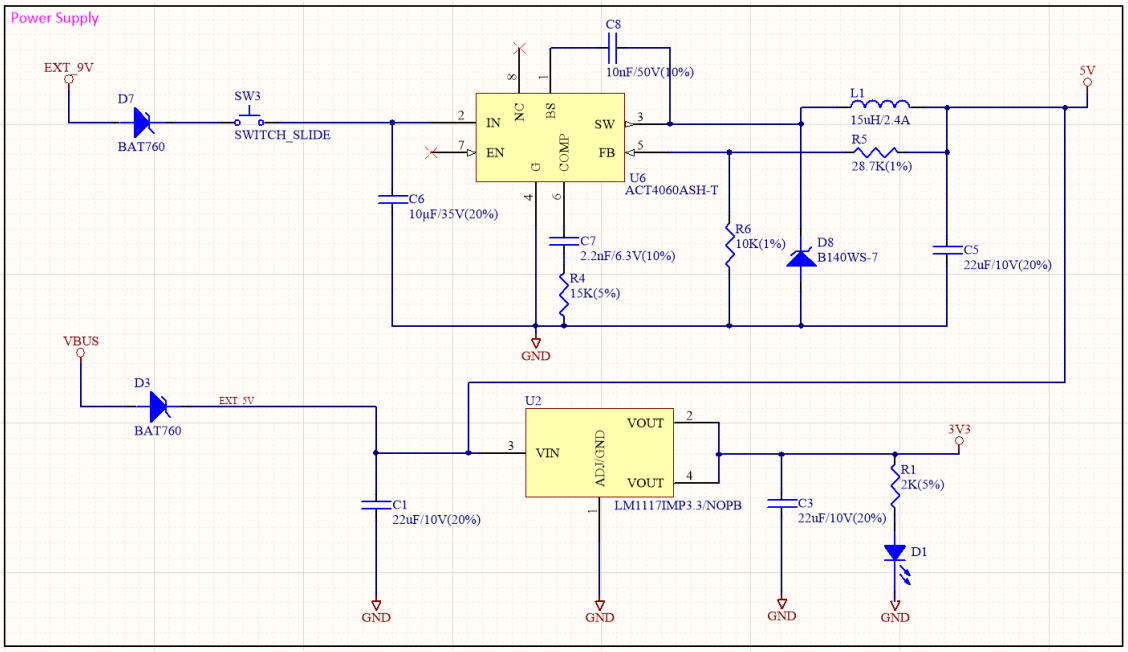


Figure 17: Power supply schematic

Micro USB & USB-UART The component on the top left corner of figure 18 is the micro-USB connector. The two S8050-G-NPN transistor allow to program the microcontroller. Holding down the Boot button and pressing the EN button initiates the firmware download mode (refer to figure 20). Then, the user can download firmware through the serial port. Finally, the CP2102N-GQFN24 component corresponds to the USB-to-UART bridge controller (Silicon-Labs 2017).

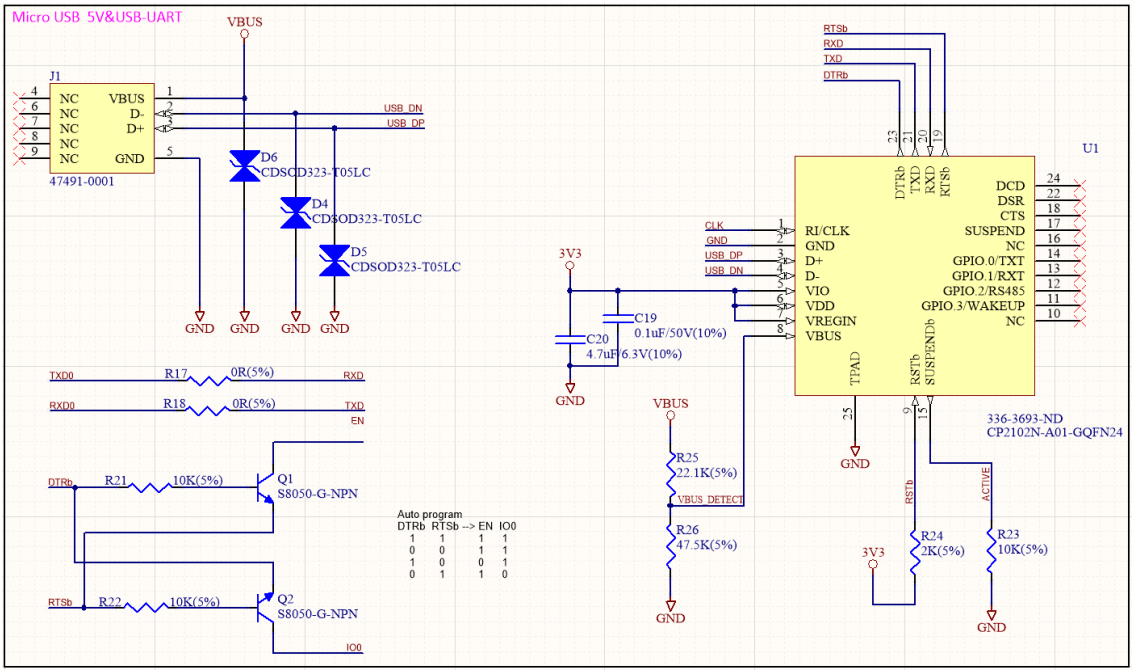


Figure 18: Micro USB & USB-UART schematic

LEDs Voltage Translation As already explained previously, the LEDs APA102C require an input voltage of 5V, also for the command signals. Therefore, two SN74LVC2T45 components, dual-bit and dual-supply bus transceiver with configurable voltage translation (Texas-Instruments 2017), have been used to translate the voltage from the microcontroller at 3V3, to 5V.

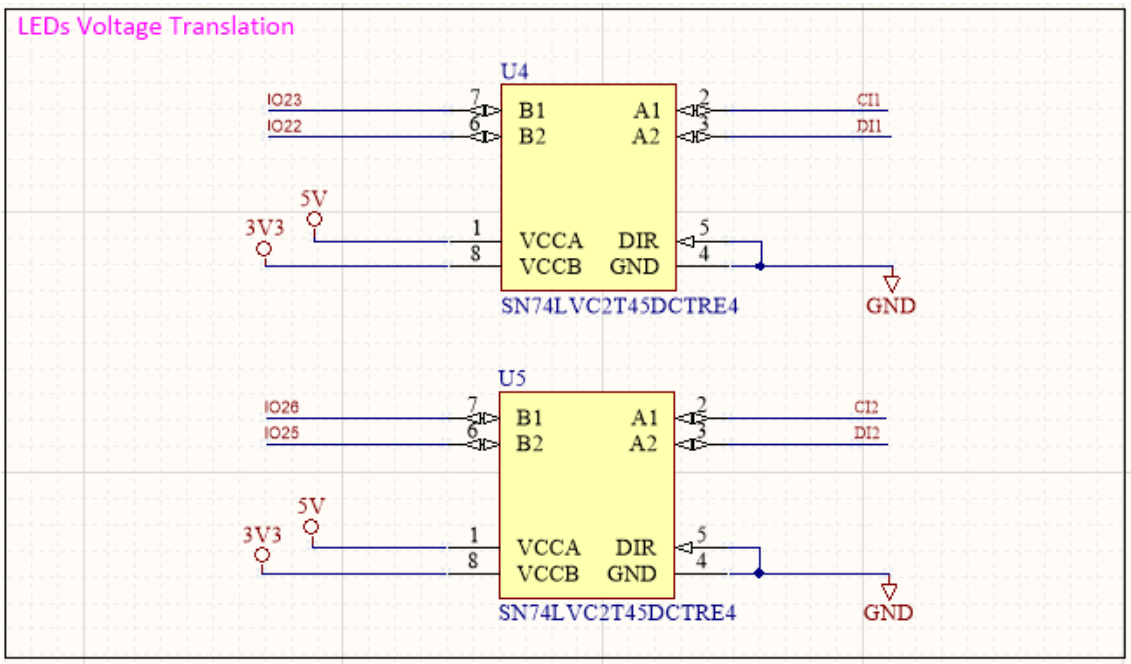


Figure 19: LEDs voltage translation schematic

Switch Buttons Two switch buttons PTS645 are used for the boot and the EN/reset of the microcontroller, as mentioned previously.

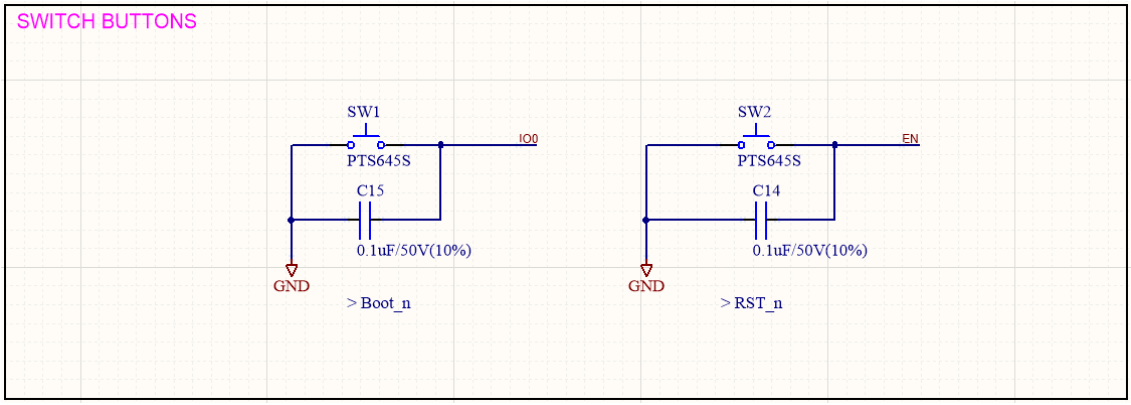


Figure 20: Switch buttons schematic

Connectors Two connectors of 21 pins are interface the PCB with the battery supplying 9V power, the LED strip and the capacitive touch sensors. All the signals that have been doubled are here for debug purposes and design constraints. In fact, a wide pitch between

two pins to be connected with textile wires was needed to prevent them intersecting each other, thus avoiding short circuits.

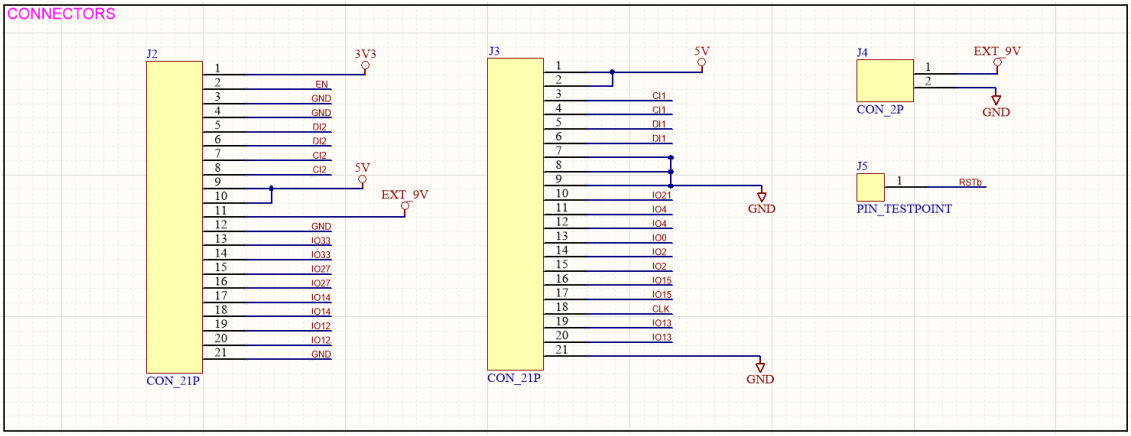


Figure 21: Connectors schematic

The full schematic can be found in appendix, figure 47.

3.2.2 Bill of materials (BOM)

The resulting bill of materials is tabulated figure 22.

Quantity	Manufacturer 1	Manufacturer Part Number 1	Supplier Order Qty 1	Supplier Unit Price 1	Supplier 1	Supplier Currency 1	Supplier Subtotal 1
5	Murata	GRM188R71H104KA93D	5	0,04	Distrelec	CHF	0,21
4	Murata	GRM188R61A226ME15D	4	0,31	Digi-Key	CHF	1,23
4	Yageo	RC0402JR-0710KL	4	0,06	Newark	CHF	0,24
3	Bourns	CDSOD323-T05LC	3	0,81	Mouser	CHF	2,43
2	Sullins	PREC021SAAN-RC	2	0,37	Digi-Key	CHF	0,73
2	Yageo	RC0402JR-070RL	2	0,06	Newark	CHF	0,12
2	Yageo	RC0402JR-072KL	2	0,06	Newark	CHF	0,12
2	Diodes	BAT760-7	2	0,43	Avnet	CHF	0,86
2	ITT C&K	PTS645SL43-2-LFS	2	0,13	Avnet	CHF	0,25
2	Texas Instruments	SN74LVC2T45DCUT	2	0,55	Farnell	CHF	1,11
1	Murata	GRT188C80J475KE01D	1	0,3	Mouser	CHF	0,3
1	TDK	C0603X7R0J222K030BA	1	0,1	Mouser	CHF	0,1
1	KEMET	C0603C106M8PACTU	1	0,45	Mouser	CHF	0,45
1	TDK	C1608X7R1H103K080AE	1	0,2	Mouser	CHF	0,2
1	Silicon Labs	CP2102N-A01-GQFN24	1	1,33	Digi-Key	CHF	1,33
1	Espressif Systems	ESP-WROOM-32	1	3,75	Mouser	CHF	3,75
1	Texas Instruments	LM1117IMP-3.3/NOPB	1	0,8	Distrelec	CHF	0,8
1	Rohm	SML-D12U1WT86	1	0,21	Mouser	CHF	0,21
1	Samsung	RC0402F103CS	1	0,18	Digi-Key	CHF	0,18
1	Yageo	RC0402JR-0715KL	1	0	Newark	CHF	0
1	Yageo	RC0402FR-0722K1L	1	0	Arrow	CHF	0
1	Yageo	RC0402JR-072K7L	1	0,06	Newark	CHF	0,06
1	Yageo	RC0402FR-0747K5L	1	0,02	Future Electronics	CHF	0,02
1	Diodes	B140WS-7	1	0,38	Mouser	CHF	0,38
1	Taiyo Yuden	NRS8040T150MJGJ	1	0,49	Mouser	CHF	0,49
1	Comchip	SS8050-G	1	0,23	Mouser	CHF	0,23
1	Wurth Electronics	450302014072	1	2,26	Distrelec	CHF	2,26
1	Active-Semi	ACT4060ASH-T	1	0,58	Mouser	CHF	0,58
1	Molex	47491-0001	1	1,03	Distrelec	CHF	1,03

Figure 22: Electronic bill of materials

3.2.3 Main PCB design

3.3 Detachable module

3.3.1 Functionality

3.3.2 Schematic

3.3.3 Detachable module design

4 Mechanical engineering - Chloe

A connected plush toy is not a common engineering project. We achieved good results by collaborating closely with the ECAL designers, in particular Chloe and Marjane worked together on the plush toy, Soft PCB and blackbox design.
In addition to a "regular" project tasks such as designing a PCB and housing, we had to design a plush toy, and more importantly, soft electronics that would integrate inside the plush toy without being felt by the child. Designing electronics with textile comes with its own specific challenges, such as ensuring the flexibility of all components while avoiding short circuits from non-insulated "cable" threads or avoiding interferences in capacitive touch sensors.

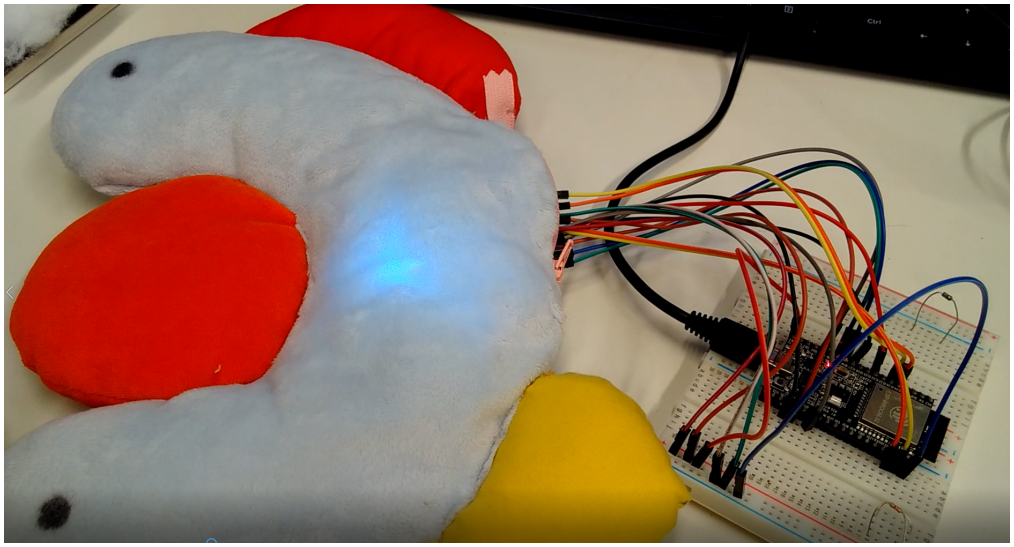


Figure 23: Second prototype of our connected plush toy, as presented at MS5.

Terminology:

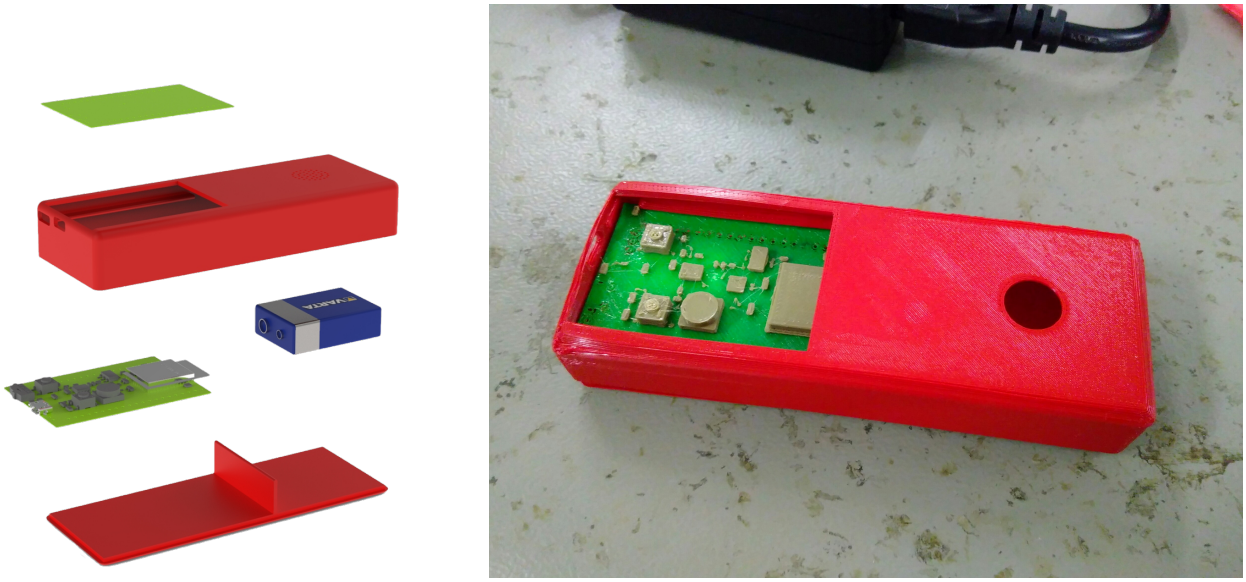
Blackbox Inevitable "Hard" electronics such as microcontroller and battery, fitted in a slim housing inside of the plush toy.

Soft PCB Textile electronics for interactions, such as LEDs and capacitive touch sensors, designed with conductive textile and threads to be soft and "invisible" to touch inside of the plush toy

Skin Soft textile layer surrounding all of the electronics, not taking into account any of the electronics components ...

4.1 Blackbox design

The 3D modelling of the blackbox was carried on by Marjane as part of her industrial design tasks. However, we collaborated closely on brainstorming, defining shapes, connectivity and positioning of the blackbox in the prototype.



(a) 3D rendering

(b) 3D printed blackbox with 3D printed PCB

Figure 24: First iteration of the blackbox design, including PCB, battery and loudspeaker

Blackbox description The blackbox holds the PCB, battery and speaker. It connects to the Soft PCB through a secondary PCB breaking out the pin connections into textile conductors. The blackbox should be easily removable by the caretaker, give easy access to the battery for replacement/recharging and be easy to connect to the the soft PCB. In our current version, the blackbox is connected to the soft pcb through pins; in a future version a more user-friendly connector will be considered.

Textile considerations The blackbox has to fit inside of a plush toy and be as little intrusive as possible. We decided to make the blackbox in a rounded, elongated ("banana") shape following the curves of the plush toy. A pocket accessible through a zipper is designed to hold the blackbox and define the connections with the "soft PCB".

Figure 25: Layers of the first prototype: outer skin, plush stuffing and soft PCB

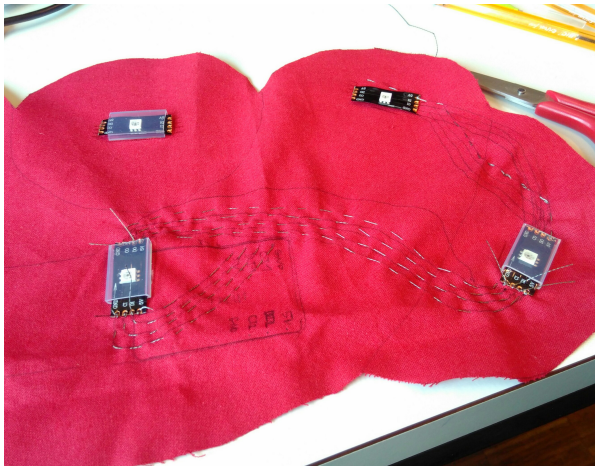
PCB design In our current version of the PCB (iteration 1), we decided to break out the circuit with many pins to keep prototyping options open for connectivity of hard to soft interfaces. The main PCB is linked to the Soft PCB through a secondary hard PCB, attached to the textile, to which conductive threads are tied, linking the elements (LED, sensors).

To achieve the smallest possible form factor, we decided to re-desing the shape of the PCB for a future version according to the desired shape of the blackbox. For this, we take into consideration the hard/soft connections, buttons, positioning of connectors, battery and speaker elements.

4.2 Soft PCB

The "Soft PCB" is a 3-layer 2D fabric "PCB" with a first layer of textile and conductive touch pads, an insulating layer and a second textile layer with LEDs and conductive thread or textile conductor tracks linking the LEDs. The tracks are connected to pins in the current version of the prototype and will be linked to the secondary PCB in the next iteration of the prototype. In a future iteration of the prototype, we will consider a more compact, robust and user-friendly connector.

(a) Textile capacitive sensors on soft PCB



(b) Conductive thread linking the LEDs

Figure 26: Details of top and lower layers of soft PCB

4.2.1 Washability

Our goal for the plush toy would be to be fully machine-washable once the blackbox is removed. The electronics (LEDs, connectors) would ideally be sealed in silicon while the textiles can endure a small number of washing cycles.

4.2.2 Manufacturing options

For the first two prototypes, the LEDs were hand sewn to the soft PCB. The touch sensor pads were either machine sewn or heat-bonded to the textile support. As it is, the soft PCB takes several hours of tedious handwork to manufacture. We are currently experimenting

with using the heat-bond conductive fabric on both sides of the soft PCB on which LEDs will be directly soldered (Fig. 27).

Figure 27: Test sample of heat-bonded conductive tracks

4.3 Soft sensors

4.3.1 ESP32 integrated capacitive touch sensors

In our project, we take advantage of the ESP32's integrated capacitive touch sensors. These are linked to conductive textile pads on the Soft PCB.

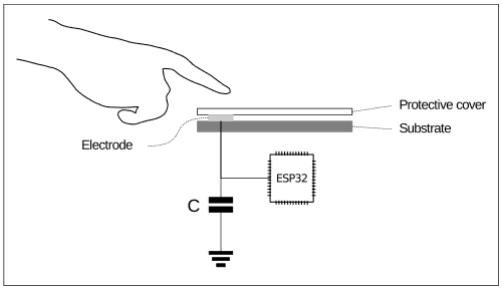


Figure 28: ESP32 built-in touch sensor

4.3.2 Prototyping design

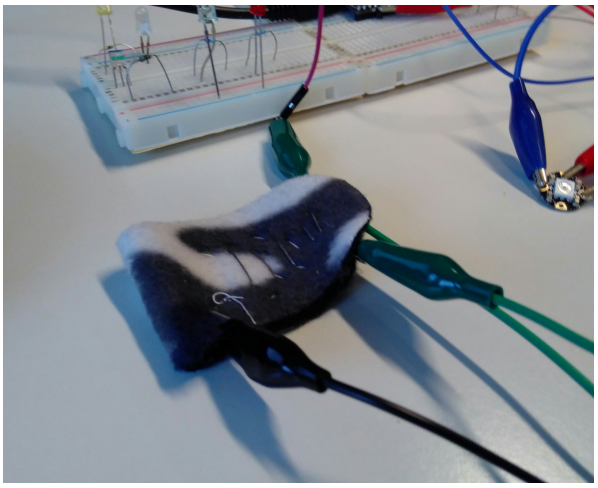
For our prototypes, we have experimented with different test codes, different materials and different touch pad designs.

Conductive materials We experimented with conductive knit and woven textile from Adafruit, conductive paint from Bare Conductive and heat-bond conductive fabric from Less EMF. The latter is the best in terms of manufacturability as it can easily be attached

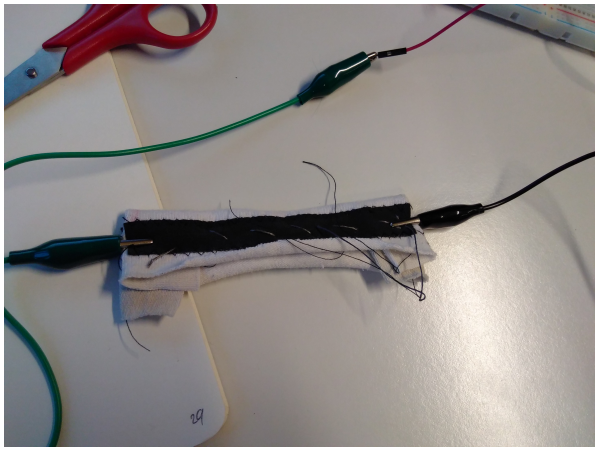
to the soft PCB fabric and is less susceptible to fraying and creating short circuits. The conductive paint is great for prototyping and fixing small connectivity issues but is water-soluble so it will not be considered as a reliable maufacturing option for the prototype (if sealed in acrylic it may be considered, especially for manufacturing speed such as in silk-screen printing).

4.3.3 Experimenting with soft sensors

We experimented several types of soft sensors and materials: bend sensors with Velostat; stretch and pressure sensors with Eontex and a potentiometer thread from Adafruit. After considering manufacturability, ease of integration, repeatability, interaction possibilities and costs, we decided to keep only the touch/press sensors.



(a) Velostat bend sensor



(b) Eontex stretch sensor

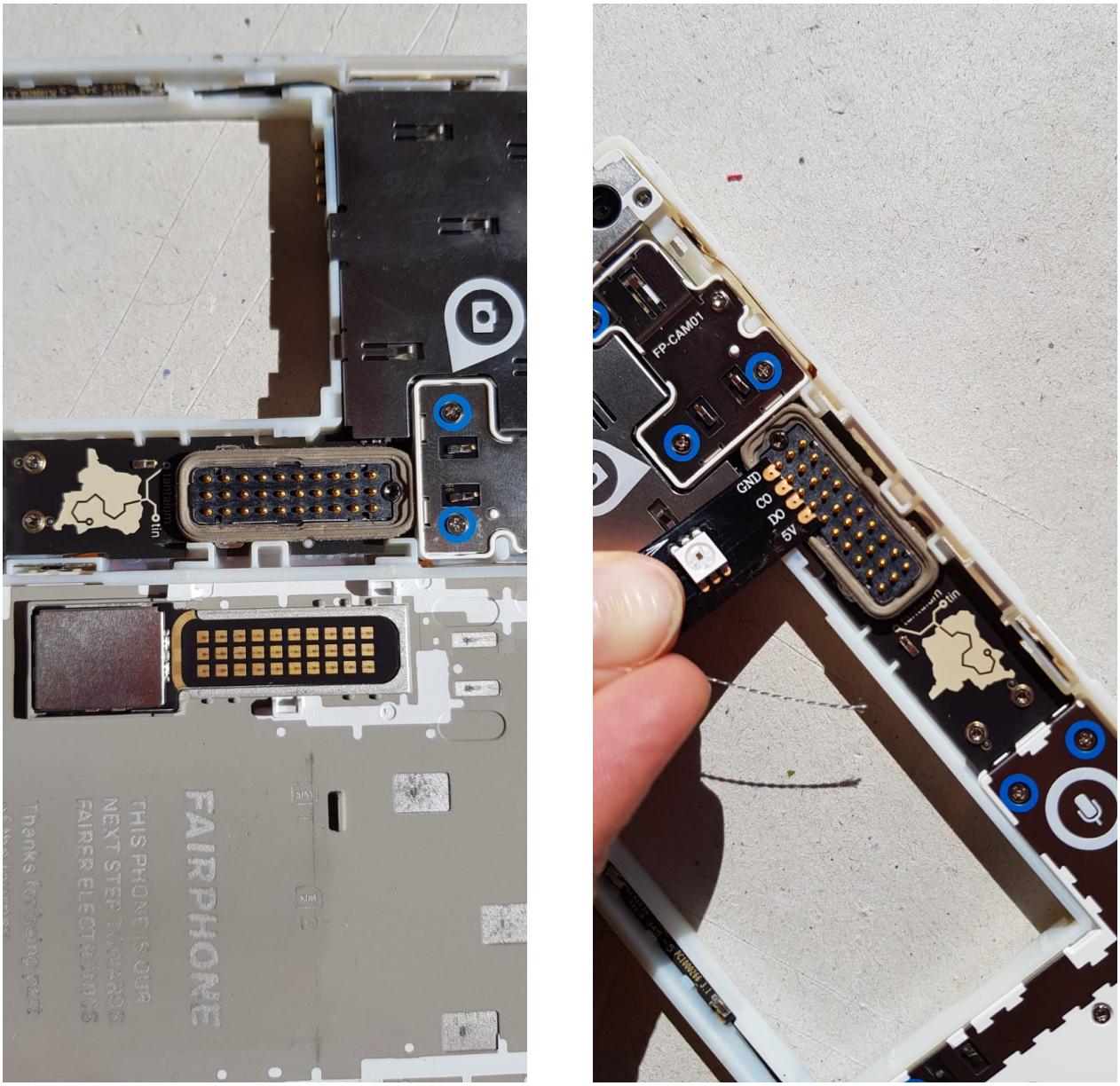
Figure 29: Examples of two soft sensors we experimented with

4.4 Hard <-> Soft interfaces

In our first prototype, we used banana cables attached to safety pins to ensure connectivity of the textile pads to the microcontroller. The second prototype had crimp pins attached to the conductive threads. The third prototype (MS6) will probably have the conductive threads/textile tracks attached to the secondary PCB.

4.4.1 Next steps

For China or the next semester, we would like to have a compact, robust and user-friendly connector. Such connector could be designed with pogo pins on one side (PCB or textile) and connecting pads on the opposite side to ensure proper connectivity in a user-friendly and compact packaging. This design has been inspired by the connectors for different modules found in the Fairphone (Fig. 30).



(a) Pogo pin connector

(b) Connector with LED element

Figure 30: Example of pogo pin connector from Fairphone's electronics

5 Firmware engineering - Chloe

5.1 Firmware architecture

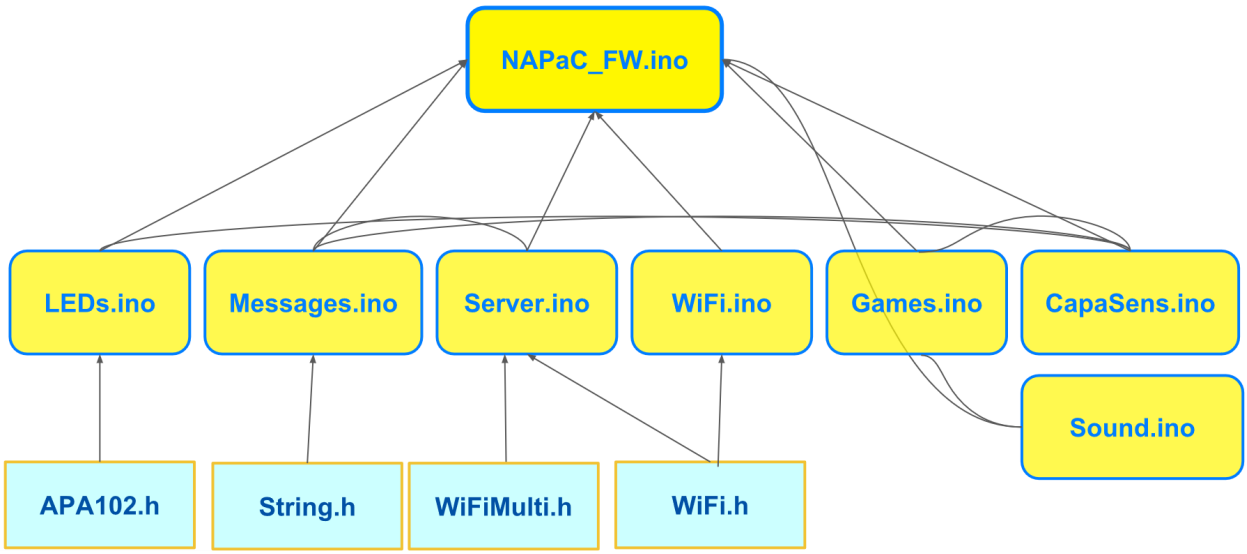


Figure 31: Firmware architecture diagram with units and libraries

The chosen architecture for our Arduino program is illustrated in Figure 31. The main file (NAPaC_FW.ino) calls each separate unit which may call eachother depending on the implemented functions. Each unit is conceived as a "library", with its setup function and functions relative to a hardware component or interaction.

NAPaC_FW.ino Used as the main file for our firmware. Runs through setup functions for each unit then calls the other functions (game interaction) from the loop.

LEDs.ino Based on the APA102 library, this unit defines LED pins and sets up LED strips and output pins during setup. In this unit, we define principal actions for LEDs such as turning on in defined colour, blinking, and turning on the LEDs in appropriate colours according to the game steps defined in figure 32.

Messages.ino This unit defines the "alphabet" in accordance with the communication protocol defined in Table 6.1. This unit also holds functions for sending defined interaction messages to the server.

Server.ino The Server module had functions for connecting to the server, reading and sending messages as well as close server connection.

Games.ino The Games unit is the main file for interactions. It has functions defining a solo game and the game with parents, and keeps record of the Zone status (off, on, colour) for the plush toy. It also has functions defining the beginning and end of game sessions which interact with the server to open and close games.

CapaSens.ino In this unit, we set up the capacitive touch sensors and define the touch status with the touch library provided by ESP32.

Sound.ino The Sound unit enables the microcontroller to play sounds through the buzzer. We have defined the standard Western muscial scale (Do-Re-Mi-...) and implemented several sound tests and jingles for the games.

5.1.1 Choice of coding environment

After experimenting with both ESP32 Eclipse framework and the Arduino environment, we decided to chose Arduino as our coding platform for the widely available examples and ease of programming.

Libraries In our project, we use libraries for the APA102 LEDs, WiFi configuration and capacitive touch sensors. Their main parameters and working principles are described below.

5.2 Description of Functions

5.2.1 NAPaC_FW

Setup The setup function runs through the induvidual setup functions of each unit: LEDs, Capacitive Sensors, Sound. It then proceeds to find WiFi SmartConfig (or connect to WiFi using an existing one), connects to server then sends a "presentation" message to the server sending its ID.

```
void setup(){
    Serial.begin(115200);
    delay(1000);

    setup_LEDs();
    setup_capa();
    setup_sound();

    setup_wifi_smartconfig();
    connect_to_server();

    hello();
    first_message();
}
```

Main Loop In the main loop, the program waits for a message requesting the beginning of a game session. In a final version of the firmware we could expect functions such as checking for child's presence, sleep mode, etc.

```
void loop(){
    message = read_message();
    if (!message.equals("")){
      messageID = message.substring(16, 20).toInt();
    }
```

```
6
7
8       switch(messageID){
9         case 2001:
10          Serial.println("Message 2001 received from server");
11          accept_game_request();
12          break;
13      }
14  }
```

5.2.2 Messages

The messages unit is used to deal with sending messages to the server. These messages are defined in Section 6.1.

Alphabet setup This snippet of code sets up the characters required form the communication protocol.

```
1 char EOT[1] = {char(4)};
2 char RS[1]  = {char(30)};
3 char US[1]  = {char(31)};
```

first_message The microcontroller sends a message to the server with its unique identifier, allowing the server to link it with its IP address.

```
1   char message1[22]={0};
2   sprintf(message1,"%c0020%cS001%cP314%c0001%c",
       STX[0],RS[0],RS[0],RS[0],EOT[0]);
3   send_message(message1);
4   Serial.println("First Hello to server sent!");
5 }
```

accept_game_message Sends the 2002 message meaning the child has accepted the parent's game request

```
1   char message1[22]={0};
2   sprintf(message1,"%c0020%cU123%cP314%c2002%c",
       STX[0],RS[0],RS[0],RS[0],EOT[0]);
3   send_message(message1);
4   Serial.println("Message sent: Child accepted game
       session!");
5 }
```

LED_on_message Sends message 2003 with LED ID of LED turned on by the child

```
  char message1[31]={0};
  char led_id = (char)LEDid + offset;
  sprintf(message1,"%c0020%cU123%cP314%c2003%c1%c%c%c",
     STX[0],RS[0],RS[0],RS[0],RS[0],US[0],led_id,EOT[0]);
  send_message(message1);
  Serial.print("Message sent: Child turned on LED");
  Serial.println(LEDid);
}
```

LED_off_message Sends message 2004 with LED ID of LED turned off by the child. This message is very similar to message 2003 above.

hello Prints a welcoming message to serial.

5.2.3 WiFi connectivity and Server communication

We use SmartConfig to send the WiFi configuration to the microcontroller from the parent's smartphone. The WiFi configuration can also be save to the program for easier programming. Our goal for China or next semester is to save a new SmartConfig to the EEPROM memory then retrieve it upon launch to connect to a known WiFi network.

WiFi Smartconfig The Arduino SmartConfig libraries allow us to easily connect to a WiFi with configuration send by a nearby smartphone.

```
void setup_wifi_smartconfig(){
  WiFi.mode(WIFI_AP_STA);
  WiFi.beginSmartConfig();

  //Wait for SmartConfig packet from mobile
  Serial.println("Waiting for SmartConfig.");
  while (!WiFi.smartConfigDone()) {
    blink_LED(0,green);
    Serial.print(".");
  }

  Serial.println("");
  Serial.println("SmartConfig received.");

  //Wait for WiFi to connect to AP
  Serial.println("Waiting for WiFi");
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
    Serial.println("WiFi Connected.");
    Serial.print("IP Address: ");
```

```
23      Serial.println(WiFi.localIP());
24      set_LED(0,green);
25      delay(1000);
26      set_LED(0,off);
27  }
```

Connection to server The code below allows the microcontroller to connect to the server in order to enable communications with the parent's smartphone.

```
1  #include <WiFi.h>
2  #include <WiFiMulti.h>
3
4  WiFiMulti WiFiMulti;
5
6  // Use WiFiClient class to create TCP connections
7  WiFiClient client;
8  const uint16_t server_port = 6789;
9  const char* server_host = "192.168.1.10";
10
11 /*
12  * Waits until connects to the TCP server, at the given host ip
        and port
13  */
14 void connect_to_server()//char * server_host
15 {
16      Serial.print("connecting to Server ");
17      Serial.println(server_host);
18
19      while (!client.connect(server_host, server_port))
20      {
21          Serial.println("connection to server failed");
22          Serial.println("wait 5 sec...");
23          delay(5000);
24      }
25 }
```

Read message from server This function reads a message sent by the parent's smartphone and transmitted through the server. It reads the message and saves it into a string that it returns.

```
1  String read_message()
2  {
3      String stringa = "";
4      char c;
5
6      c = client.read();
```

```
7       if (c == char(2)){        //STX[0] = char(2);
8       do{
9           stringa += c;
10          c = client.read();  // lettura di un byte
11          //Serial.println(c);
12      }while(c != char(4));//EOT[0] = char(4);
13      Serial.println(stringa);
14      }
15      return stringa;
16  }
```

5.2.4 Games

The main part of our firmware and most visible to the user is the "game" interaction. For now, we have set one type of interaction, or one game, but could imagine implementing other ones. A solo game is also available for testing. Figure 32 presents the flowchart for the game interaction. The goal of the "game" is that either the parent or child can chose to turn on one or several LEDs which will also light up on the game partner's device. A sound corresponding to the zone will also be played. The LEDs turn on the colour of the player who turned them on and the other player can turn them off and then on in their colour.

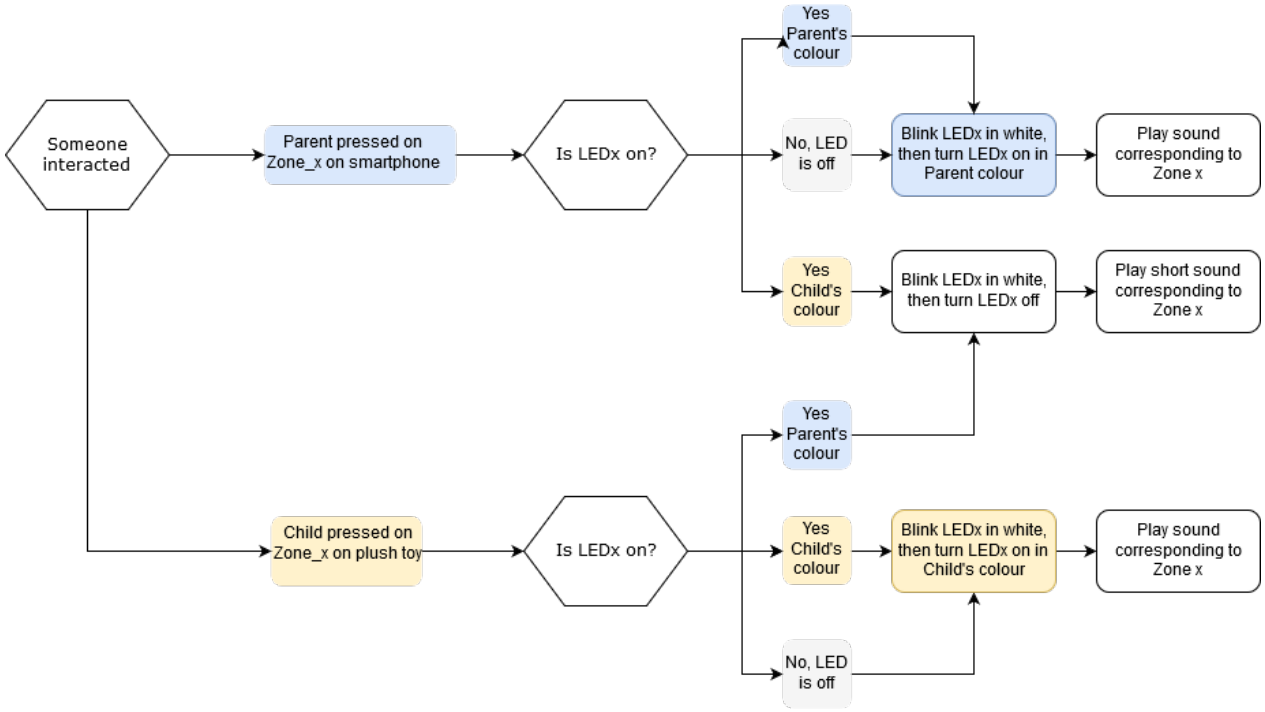


Figure 32: Flowchart for game interaction

5.2.5 Capacitive touch sensors

The capacitive touch sensors are crucial for the child's interaction with the plush toy. The functions used in the firmware are listed below. In the prototype presented in MS5, we only distinguished between touched and untouched, but hope to find the right settings to have a better sensitivity to different kinds of touching or pressing through the skin and plush stuffing.

CapaSens Functions

setup_capa Sets up the capacitive touch sensor pins as inputs then reads the initial value of each touch sensor.

touch_read_value(touch_id) Reads the value on sensor of given ID.

capa_touched(touch_id) Returns PRESSED if the touch sensor is being touched or RELEASED if not. Further improvement would include different states of touch (touch, light press or strong press).

presence Returns 1 if any sensor is touched, meaning the child is holding or playing with the plush toy.

test_touch_values Test function which prints the sensor values in serial.

test_if_touched Test function printing in serial the ID and value of any touched sensor. ...

Influence of sensing parameters The ESP32 built-in capacitive sensors have three parameters (illustrated in Fig. 33) that can be set by the user: measure time, voltage range and charge/discharge speed. These parameters will affect the sensitivity of the touch sensors. A slow charge/discharge cycle will lead to less sensitivity while a fast cycle will lead to a high sensitivity. We still need to find the right set of parameters to ensure detection of touch through textile layer while limiting the risk of false positives and interference between sensors. With the right set of parameters, we will be able to detect and differentiate touch and press on a single touch sensor through layers of fabric.

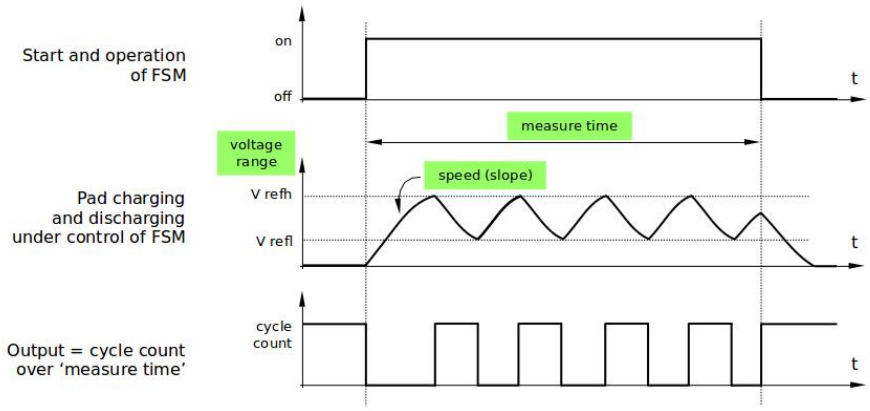


Figure 33: ESP32 built-in touch sensor parameters

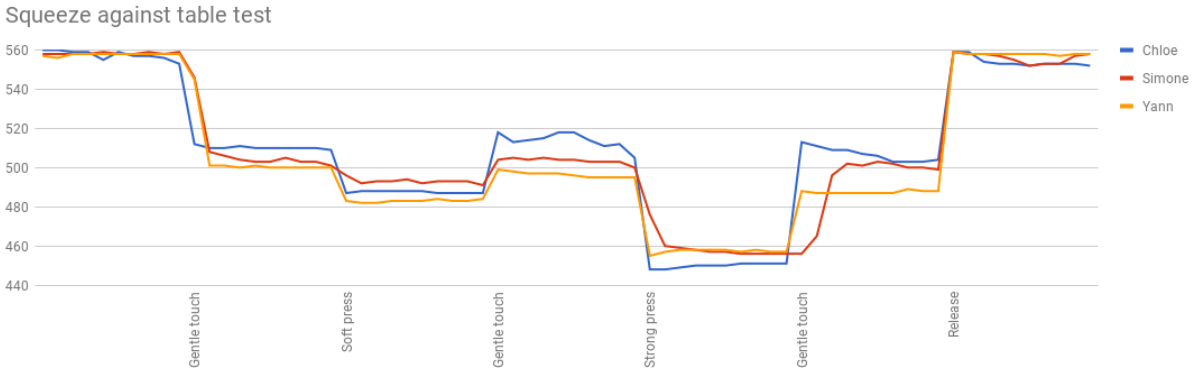


Figure 34: Test values for different touch/squeeze patterns with different users

5.2.6 LEDs

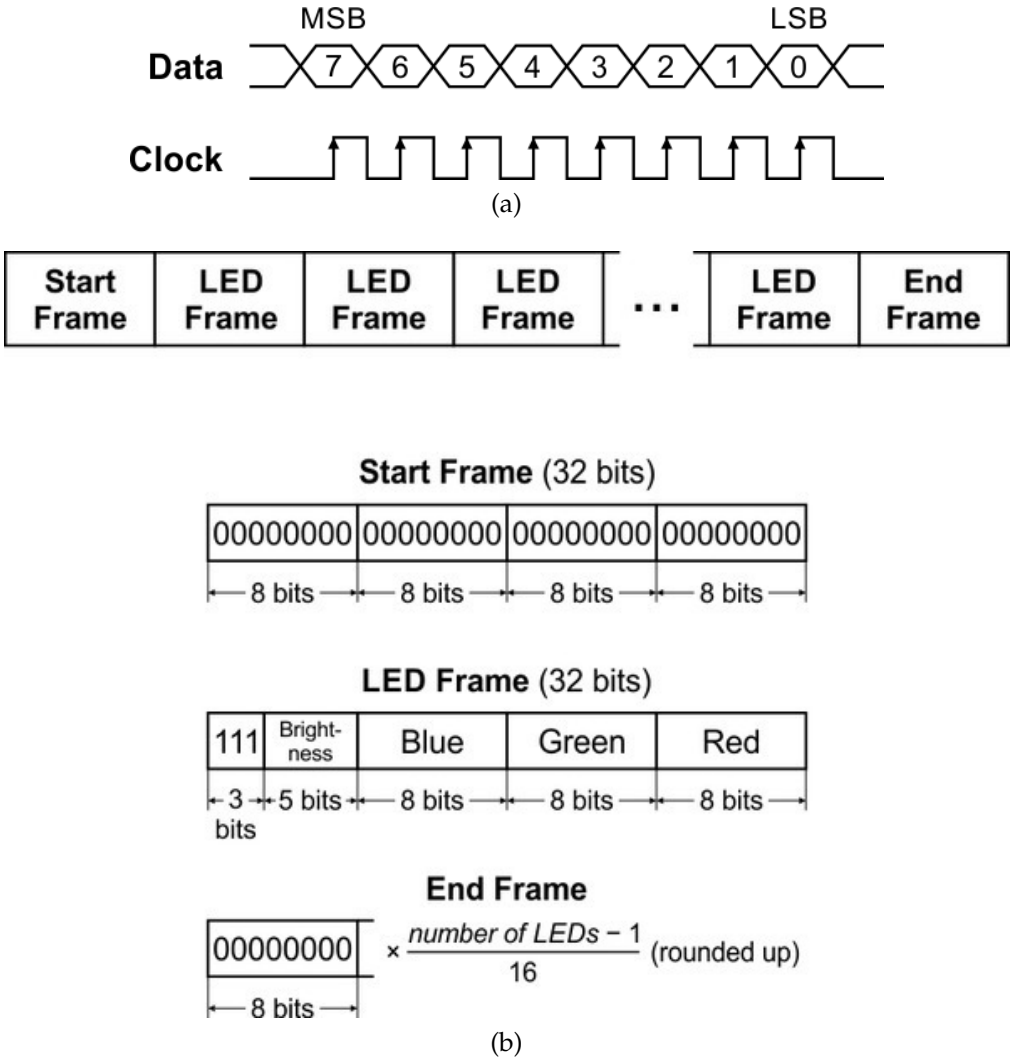


Figure 35: APA102 communication protocol

5.2.7 Capacitive touch sensors

The Arduino environment does not allow easy tweaking of the capacitive touch sensors described in 5.2.5. This can be done in C programs available from the ESP32 GitHub. We are still testing the parameters to find the right sensitivity and parameters, which depend on the shape and connections of the touch sensors.

5.2.8 Sound

The Arduino Tone function and sound libraries have not been implemented yet for ESP32. In order to circumvent this problem, we use ESP32's LED PWN abilities to control our buzzer (described in Section 3.1.2). With the PWM, the duty cycle and frequency can be set. We used a fixed duty cycle (which influences the loudness of sound) and used the frequency parameter to set up tones.

```
int freq = 2000;
int channel = 0;
int resolution = 8;


void setup_sound(){
  pinMode(soundPin, OUTPUT);
  ledcSetup(channel, freq, resolution);
  ledcAttachPin(21, channel);
}


void play_tone(int tone){
  ledcWriteTone(channel, gamme[tone-1]);
  delay(200);
  ledcWriteTone(channel, 0);
}
```

5.3 Next steps - functions to be implemented

5.3.1 WiFi save Smartconfig in EEPROM

5.3.2 Sleep functions

5.3.3 Sound settings and implementing a proper loudspeaker

5.3.4 Capacitive sensors calibration and touch/squeeze recognition

5.3.5 Error checking

6 Software engineering

During the previous section, the different aspects of the project have been described both in electronic and mechanical terms. However, it is important to recall to the reader that the overall work is built in a quite complex system that incorporates together multiple actors. In fact, while we have been describing in detail for the past sections the development of one actor (a connected plush toy), the overall system is composed of different ones. Those actors are considered heterogeneous with each other in both form and functionality. However, in order to have a working prototype, every single piece of the puzzle needs to perfectly fit within the bigger picture. Generally speaking, in the coming sections, we will indicate as the *Toygether system* the set of different actors interacting together. Such system is illustrated in figure 36 by an over-simplified diagram which, nevertheless, aims to deliver an intuition of the different actors interacting together, without unnecessary networking details.

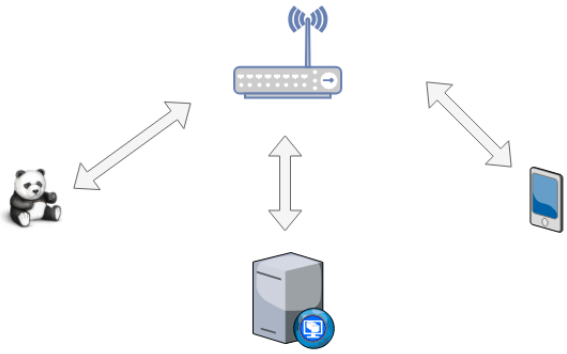
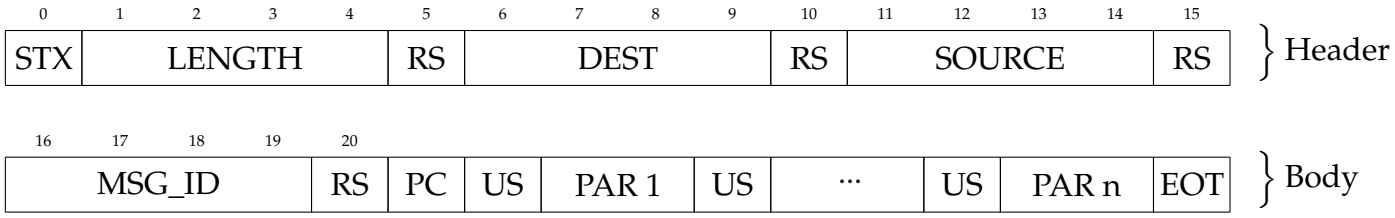


Figure 36: Simplified diagram of the overall system

The previous figure introduces the three main categories of actors in the system: the **sever**, the **plush toy** and the **android user**. As already mentioned, the previous illustration corresponds to an over-simplification of the actual system by drawing a single representative for each class. However, it is crucial to understand that in a "real" context (outside of the purely prototyping domain) multiple clients and servers (in a distributed design) could operate in parallel. During the upcoming sections, in which design developments for each element will be discussed in detail, such vast reality needs to be considered to avoid architectures that fail to scale.

The next subsections will firstly introduce the reader to the "common language" each actor will be required to speak in the system (subsection 6.1). After the communication protocol has been defined, the server will be illustrated in detail (subsection 6.2); before eventually discuss the other client-class in the system defined as the Android APP for parents (subsection 6.3).

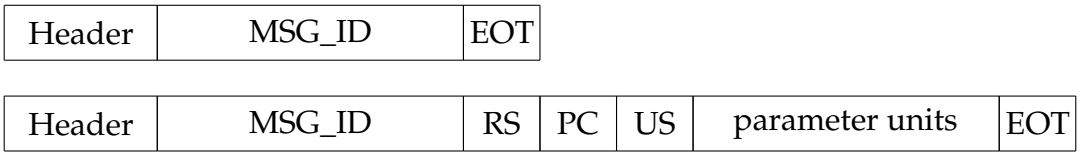
6.1 The communication protocol definition



In this first section, the basic standards applied to each message within the communication protocol will be presented. Each data/info exchange between different actors of the system is required to be constructed following those same rule. The result will be a working communication channel for the required application between different modules.

The diagram above shows the standard composition of a message. Reference numbers have been defined to locate the character position of elements within the not-variable portion of the message. The structure is defined with the support of four ASCII control character: STX (Start of text), EOT (End of transmission), RS (Record separator) and US (Unit separator). Each message must start with the control character STX and end with EOT. This rule delimits the boundaries for each message both client and Server can retrieve from the communication stream. Basic correctness checking could be then implemented by testing these rules on both hands of the channel. The message is substantially made of a sequence of **records**, each separated from the others by the character RS. Each record contains inside data that is always expressed in Big Endian with reference to the diagram (i.e. the least significant digit of LENGTH is at the 4th character of the message). Records that contain different pieces of data are partitioned in **units**, each separated by the character US. For example, the last record of the message structure contains the optional parameters that are partitioned in different units for each particular parameter (plus a unit containing the count of parameters inside the message).

Inside each message it is possible to identify a **Header** and a **Body**. The former is extremely static, having a fixed length (16 bytes) and fixed number of records. The Body, on the other hand, contains both a record indicating the kind of message (MSG_ID) and therefore is fixed in length up to this point, but can be extended with a variable number of parameters to follow. Generally speaking, messages can be found in either one of the following forms.



While the Header can be easily described and implemented, the Body requires an in-depth description of every single function. A list of all defined body-description can be found in the appendix which illustrates each possible MSG_ID and how it can be built (with or without parameters) for any occurring event (see appendix C). In the following

page, a table containing a description of each record/unit is presented as well.

Table 1: Description of components within a message

Field	Type	Length	Description
STX	c	1	Byte used to indicate the beginning of any new message. It is represented with the ASCII code 2, the "start of text" character
RS	c	1	Byte used to indicate the separation between two record structures. It is represented with the ASCII code 30, the "record separator" charachter
US	c	1	Byte used to indicate the separation between two unit structures (inside a record). It is represented with the ASCII code 31, the "unit separator" charachter
EOT	c	1	Byte used to indicate the end of any new message. It is presented with the ASCII code 4, the "end of transmission" character
LENGTH	d	4	Record describing the length of the whole message, from STX to EOT included
DEST	d	4	Record describing the unique identifier of the destination. Each identifier is an alphanumeric code composed of a letter (S for server, P for plush toy and U for users on mobile APP) followed by a numeric code
SOURCE	d	4	Record describing the unique identifier of the source. The identifier is defined in the same way as for the destination record
MSG_ID	d	4	Record describing the content of the message itself via a unique numeric identifier
PC	d	1	Byte containing the number of optional parameters contained in the message (without counting PC itself)
PAR x	d	variable	Each parameter unit describes a piece of data specific of the message sent. The length is variable as well. Each parameter is divided by a US separator

Type
c : ASCII control character, no useful information but required for the structure
d : data record/unit containing useful information of the message

The communication protocol has been designed such that the server would act as the middleman between plush toys and Android users. Each interaction between clients will be parsed by the server, which will redirect received messages to the right destination. This design choice brings two main benefits to the system: firstly, each communication is checked for correctness by the server, protecting clients from vulnerability due to bad-constructed messages; secondly, this design introduces a higher level of abstraction for each client, which are not required to deal with networking details (IP address, ports, etc).

An example of such abstraction could be illustrated with the so-called "introduction message", labelled with MSG_ID equal to 0001 (see appendix C for reference). The following illustration represents an example of an introduction message a plush toy could construct upon connecting to the server. In fact, following the protocol, each client is required to send a message to the server (labeled S001 in the example) which contains the MSG_ID record equal to 0001, as well as the the SOURCE record set to the unique identifier of the client itself. This operation is necessary as we aim to have an environment for which messages sent between clients do not require any networking related information (IP address and port), but could only rely on identifiers (P314, U123).

STX	20	RS	S001	RS	P314	RS	0001	EOT

The described protocol encourages such abstract identifiers associated with each client (and server). Therefore, upon connecting to the server, clients are supposed to identify themselves with the "introduction message" which will allow a direct matching between the abstraction (P314) and their current network information (xyz.jkw.abc.def), transparent to all actors in the system.

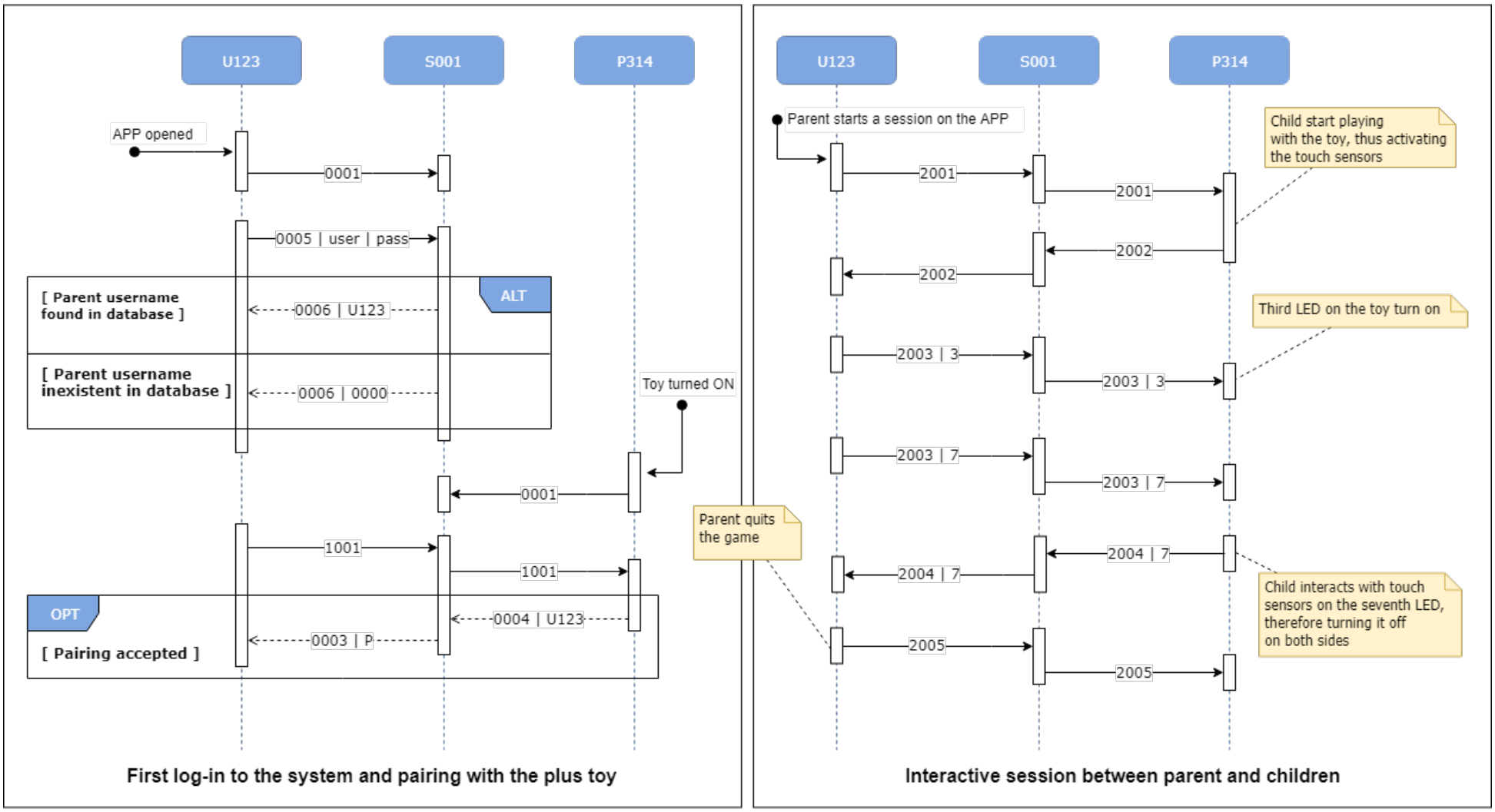
6.1.1 An example pipeline

In order to provide a clear description of the adoption of such communication protocol, figure 37 on the following page illustrates a possible pipeline between three actors. The communication is held by a Server (labeled S001) and two clients, a plush toy (P314) and a parent as Android user (U123). The diagram presented has been created by following the UML standard for sequence diagrams.

The diagram in figure 37(a) illustrates message exchanges during the pairing procedure between a parent APP and a toy. To such intent, the parent log-in via the application by typing both required user-name and password. Such information is sent to the server, which will alternatively return either the user identifier or an error code (see appendix C). At this stage, the parent will request to the toy the "pairing privilege". This latter is required to accept such request (for security reasons) by interacting with the touch sensors. If a successful result is obtained before a set timeout, the parent and the toy are associated together and will be able to interact in the future.

Finally, the diagram in figure 37(b) showcases trivial interchanges of messages during a playing session. Parents can initiate a session, which is then accepted by the child by playing with the toy. At this stage, both parent and child can interact with each other by turning on/off LEDs and playing sounds at each end of the channel.

N.B. Messages in both diagrams are indicated as "MSG_ID | PAR_1 | PAR_2" for simplicity



(a) (b)

Figure 37: UML Sequence diagram for a communication pipeline example

6.2 The TCP Server

In this subsection, we aim to illustrate the overall development of the TCP server operating the *Toygether system*. As already mentioned, the server needs to be designed in order to bridge together plush toys and android users because communication between clients will always pass by the server itself. The software is required to implement the TCP protocol as well as the previously defined guidelines for the communication protocol (see subsection 6.1).

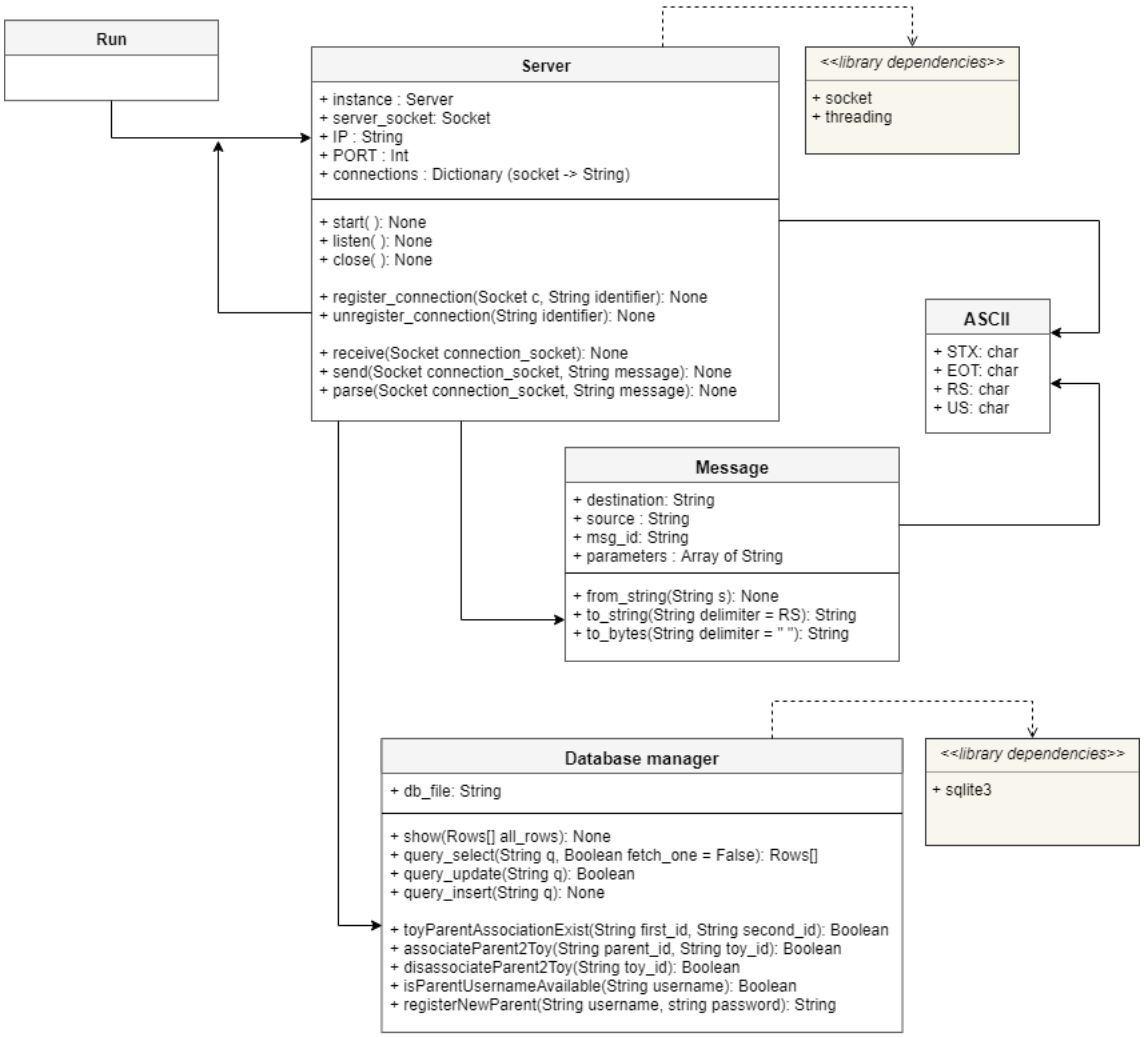


Figure 38: UML class diagram for the Server software

Figure 38 illustrates, by adopting a UML class diagram, the overall structure of the server software. Thanks to numerous design choices, the complexity of the final architecture has been encapsulated in a relatively low number of constructing blocks. The aim of the upcoming pages will be to introduce each of the three main components of the structure (main server, message and database manager) in further detail, by reporting the reasoning on different implementation decisions encountered.

Thanks to a stack diagram, illustrated in figure 39, the reader can obtain an intuition regarding dependency levels between functions of the developed result. In particular, in order to implement the software, a choice of libraries has been made to obtain the base-code upon which the project could have been built on. Python standard libraries provide capable tools for our needs: firstly, the *socket* library predisposes the necessary means to manage a TCP stream of incoming and outgoing communications with different clients; secondly, the *sqlite3* library offers a collection of APIs to integrate SQL querying with an SQLite database. On top of such libraries, our components have been defined (see the next pages for further details).

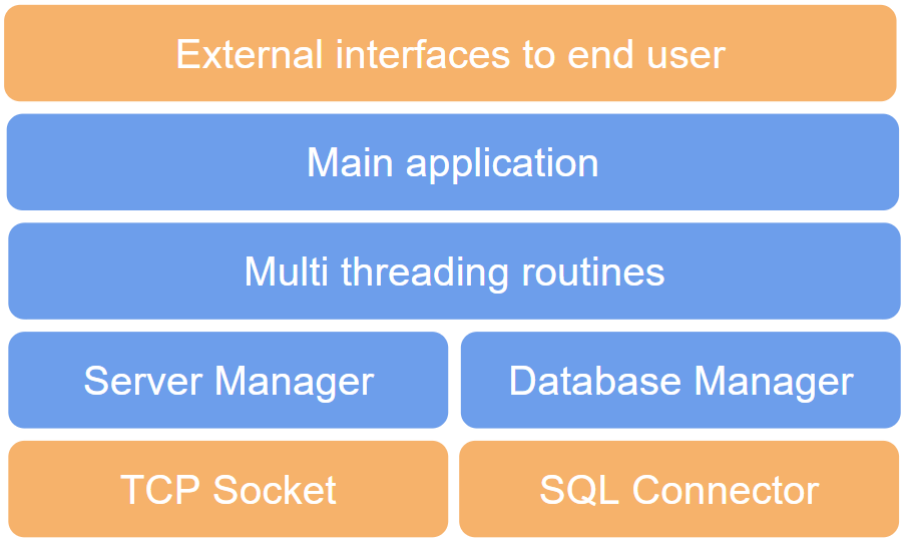


Figure 39: Stack diagram for the Server software dependencies

Before concluding this general introduction of the server software, a small extra detail in regards to the design choices needs to be appointed. Following guidelines in the book "*Design patterns: elements of reusable object-oriented software*" by Gamma (1995) for well-construct software, the server application has been built as a so-called **Singleton**. This coding design prevents multiple instantiating procedures of the object within the same environment, which would be meaningless in our context. The software is, therefore, enriched with a class-method that transparently handles the design pattern implementation.

```
@classmethod
def get_instance(cls):
    if not cls.__instance:
        cls.__instance = Server()

    return cls.__instance
```

```
from server import Server
Server.get_instance().start()
```

6.2.1 The main component: Server manager

The fundamental operations managed by the server are handled by this main component. The objective of the following paragraphs will be to introduce the reader to all the methodologies applied in order to obtain a TCP server application, capable of handling the traffic of multiple connected clients. While developing the server, particular importance has been given to the design needs illustrated in the previous sections (parallel execution of clients, for example). In order to implement the component, the standard Python socket library has been studied and adopted for tasks of I/O stream management between server and clients.

The component is executed by the *start()* function call, which main lines of code are illustrated below. The reader should be aware that, in order to convey programming information without overheads in terms of clarity, parts of the excess code with respect to the basic logic itself (i.e. try-catch clauses, output printing, etc.) have not been reported. In order to obtain a complete overview, the final code can be referenced. The function's task is to initiate an incoming stream for clients which aim to connect to the server afterwards. To this intent, the component predisposes an open stream by binding it to a defined IP address and port, which will then used as the reference point by connecting clients. For the constructed prototype, the choice of such values has been trivial (IP address equal to 192.168.1.10 and port equal to 6789).

```
1  def start(self):
2      # Create a TCP/IP socket
3      self._server_socket =
4          socket.socket(socket.AF_INET, socket.SOCK_STREAM)
5
6      # Bind the socket to the port
7      self._server_socket.bind((self._IP, self._PORT))
8
9      # Start listening on the stream for incoming connections
10     self.listen()
```

After the initialisation of the stream, the server starts the listening procedure which will wait for an incoming connection. The next code includes a simplified pipeline for such procedure to be accomplished by the server. Whenever a new connection is identified, the server instantiates a socket that will handle all future exchanges. In fact, the server is able to manage multiple clients by keeping track of a unique socket for each of them. Within a socket, the communication between a client and a server is handled via point-to-point methodologies which isolates them from the rest of the environment. Furthermore, in order to parallelize the execution of all socket related tasks by the server, each new incoming connection is assigned to a thread. The parallel programming design allows the software to continue accepting new connections, which as mentioned before halts the main thread until a client is reached, while fully operate with each client and their socket.

Due to the parallel environment, the server works in, peculiar situations might occur. For instance, multiple clients might request a connection with the server at the same time while the latter has not yet instantiated a thread for the previous connection made. In such

situation, a queue is built (in the prototype context with five clients in hold) to shunt the workload. Such queue is dependant on the running system and will be further examined in the future when scaled up solutions will be faced.

```
def listen(self):
    # Listen for incoming connections
    self._server_socket.listen(5)

    while flag_keep_listening:
        # Wait for a connection
        client_socket = self._server_socket.accept()

        # Instance a thread for the new client and run it
        threading.Thread(target=self.receive,
                         args=[client_socket]).start()
```

Each client-thread main task is to provide a listener for incoming messages received by such connected client. The communication is managed with a stream of bytes that are recorded by the software until the intended message has been fully broadcast (i.e. the end of transmission marker EOT has been detected, as previously described in the subsection 6.1). Messages, afterwards, are parsed to determine which possible event they might trigger on the server itself or whether they need to be forwarded to a different client if correctly constructed.

```
def receive(self, connection_socket):
    while flag_keep_receiving:
        # Construct a new incoming stream
        stream = connection_socket.recv(1).decode("utf-8")
        while not input_stream.endswith(ASCII.EOT):
            stream += connection_socket.recv(1)..decode("utf-8")

        # Retrieve a Message object from the stream
        message = Message.from_string(input_stream)

        if message:
            self.parse(connection_socket, message)
```

Although both *parse()* and *send()* functions could be fully explored as well, their execution is trivial compared to the previously presented components. The former provides an inspection of the received message and conditionally execute tasks according to the MSG_ID contained. The latter, on the other hand, employs the Python library tools to broadcast a string constructed message on the stream. Such operation is handled, as well, in a different threaded execution to avoid interruptions on the receiving task.

6.2.2 The Message component

The message's structure, introduced during the previous discussion over the communication protocol (see subsection 6.1), is particularly appropriate to be used in conjunction with the help of the component we are about to illustrate. In fact, the pragmatic definition of records and units that construct each message can be managed by a class, whose attributes are associated with the message's records themselves. For this intent, the message component of the server is built to store *source*, *destination*, *message id* and *parameters* of each new incoming/outgoing communication.

Furthermore, the message component becomes extremely useful whenever messages are required to be parsed by the server. The following code, for instance, provides a class-method which allows instantiating of new components via string arguments. In the previous subsection, the *receive()* function of the server would instantiate a message component by using such method: given a correct lecture of the byte-stream (interpreted as a string), the message component can be constructed which would eventually provide easy access to the contained records and parameters.

```
1  @classmethod
2  def from_string(cls, s):
3      # Check basic correctness of the string-message
4      if not (s.startswith(ASCII.STX) and s.endswith(ASCII.EOT)):
5          return None
6
7      # Obtain records from the string
8      rec = s[1:-1].split(ASCII.RS)
9
10     # If the there is NO parameters, build a message
11     if len(rec) == 4:
12         return cls(rec[1], rec[2], rec[3])
13
14     # If the there extra parameters, retrieve their units first
15     if len(rec) == 5:
16         params = rec[4].split(ASCII.US)
17         return cls(rec[1], rec[2], rec[3], params[1:])
18
19     return None
```

```
1  # Example string message that could be retrieved from a stream
2  # N.B. control char STX, RS, US, EOT are not visible
3  message = Message.from_string("0000 U123 P314 2003 1 7")
4
5  print(message.destination)      # console> U123
6  print(message.source)           # console> P314
7  print(message.msg_id)           # console> 2003
8  print(message.parameters)       # console> ['7']
```

6.2.3 The Database Manager component

The last building block of the server application is composed of a series of functions to manage interactions with the main database. This latter corresponds to the basic data structure needed to store non-volatile information about the system status.

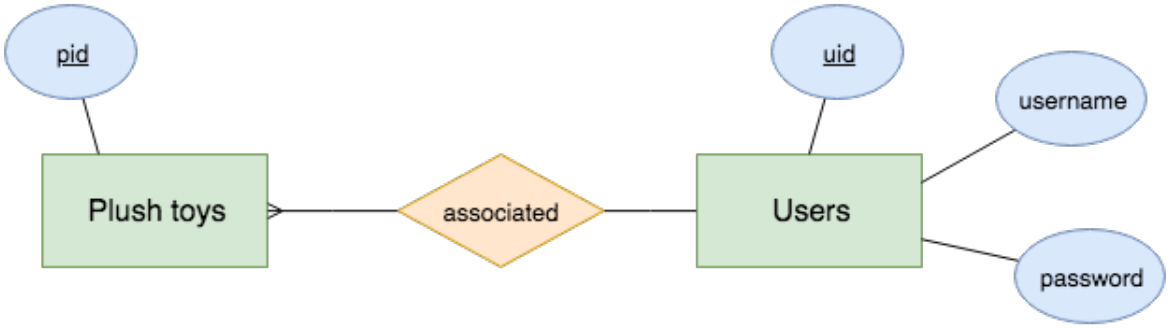


Figure 40: Entity relationship diagram for the database structure

By adopting an Entity-Relationship diagram, Figure 40 illustrates how the logical structure has been designed. The amount of information (at the present state) is not particularly big. However, it is crucial to store basic information about the system current state, by keeping track of registered users (to allow them sign-in privileges) and associated plush toy (once a pair of android app and toy have been paired together). This latter detail is of crucial importance for the server, as it covers the second advantage of the communication protocol designed (see section 6.1). As already mentioned, the advantages of the communication protocol can be identified in both a higher abstraction and a message-forwarding filtering. Therefore, the Database Manager component can be used to consult the database itself, checking that SOURCE and DEST of the message are associated (paired) before forwarding to the destination. This operation filters incorrect or malicious messages, restricting communication to pairs of clients which have authorized the link.

The following API has been implemented to correctly face database consultations by the server. Note that the reported list is only an intuition for the reader on the result obtained with this component. The communication with the database has been implemented via SQL and the provided python libraries for reaching the structure itself.

```
1 def ToyParentAssociationExist(first_id, second_id):
2
3 def associateParent2Toy(parent_id, toy_id):
4
5 def disassociateParent2Toy(toy_id):
6
7 def isParentUsernameAvailable(username):
8
9 def registerNewParent(username, password):
```

6.3 The Android APP for parents

In this final subsection, we will cover the development of the Android APP. The application has been implemented alongside the advancements provided by the MID (Media & Interaction Design) team member, who defined the overall "look&feel" of the end result. As happened for the conjunctive work between industrial designer and electronic/mechanical domains, integration of the disciplines brought along unprecedented constraints. In fact, the final result we obtained required constant trade-offs between user-interaction and development feasibility. Figure 41 generally showcases the obtained navigability of such application, which will be covered in its most important components during the upcoming pages of the report.

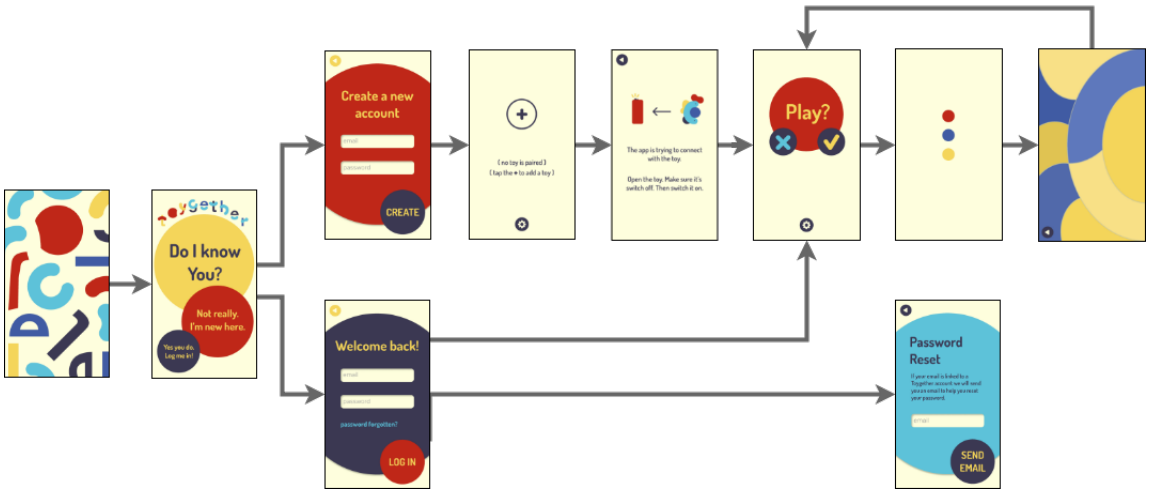


Figure 41: Android app overview

The objective of the APP, by recalling the overview of the *Toygether system* covered in section 6, is to fully implement a client for the parent access. To this intent, the Android APP integrates elements we previously mentioned during the development of the plush toy itself (for instance, the interactive game-play). However, the parent APP intrinsically determines a new set of challenges that are correlated with "behind the scenes" set-up taken care by the father/mother upon the first usage. In fact the parent figure, during their experience with the application, is required to, for example, both manage their personal account by procedures of log-in/log-out and provide a pairing pipeline to associate toy-parent together. The latter challenge has been a major discussion during the process that would design the application and its user-interaction during the different brainstorming addressing the topic.

After illustrating how the APP was designed to best implement the client requirements (subsubsection 6.3.1), the focus will be given to two major components facilitating the pairing pipeline: the WiFi SmartConfig (6.3.2) and the QR-code reader (6.3.3). Many other functions and efforts during the implementation of the APP won't be described in the report under the same spotlight as the previously mentioned ones. The decision is associated with our desire to allocate major focus of the report to exploratory and implementing phases that were fundamental to the project.

Following the methodology we adopted to present the server (subsection 6.2), a stack diagram is illustrated in figure 42. The representation is useful to underline the different dependencies between each component. Moreover, it showcases the adoption of three different libraries/API upon which the customized implementation has been built over. In particular. the Android APP has been developed with the adoption of the followings: the Java socket library for the client capability; the Barcode API by Google for the QR-code recognition and finally the SmartConfig protocol designed by EspressIf (manufacturer of the micro-controller inside the plush toy).

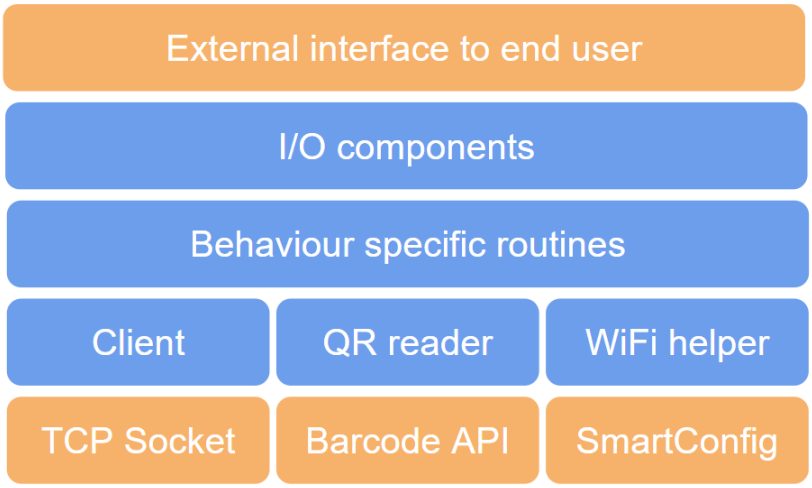


Figure 42: Stack diagram for the Android APP dependencies

6.3.1 A client within the system

The Android APP is required to fully integrate into the *Toygether system* depicted in the previous sections. Thanks to the Java standard library for TCP sockets, this operation has been able to be achieved. The development mostly encountered the same guidelines we introduced during the presentation of the server software. However, due to the usage in the Android architecture, such client component have been implemented following another famous *Design Pattern* by Gamma (1995): the **Observer** pattern.

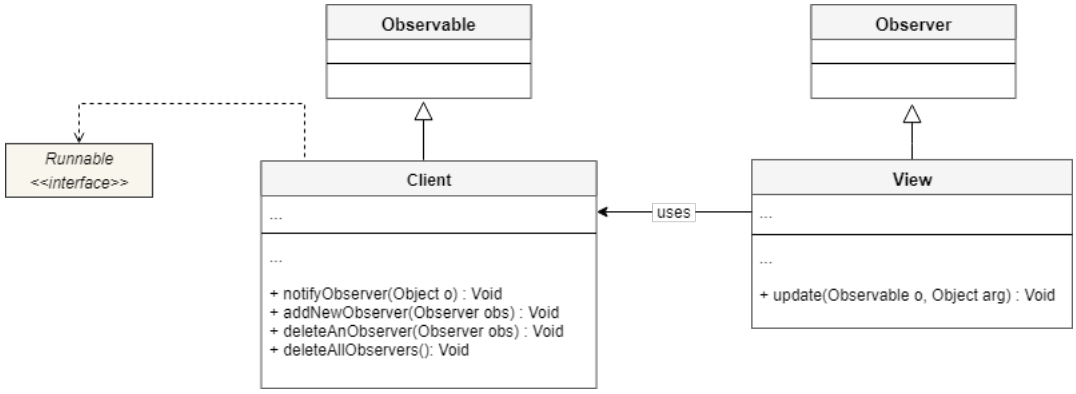


Figure 43: UML class diagram for the observer pattern in the Android APP

Figure 43 illustrates such pattern in a UML class diagram. During the execution of the application, the user is presented with a graphical user interface (GUI) that is managed by a so-called *View*: a class which task is to bridge the I/O from the user with the back-end execution. In order to produce the intended result for our prototype, the application is required to trigger particular events (change colour to an element of the GUI during playing sessions) upon receiving a message from the server (MSG_ID equal to 2003 with reference to the communication protocol in appendix C). Therefore, the *View* is designed to be an *Observer* with respect to the *Observable* Client class. The latter, whenever a message is received, notifies such change to the whole list of *Observers* (i.e. the View), which will then parse the message obtained during the automatic call of the *update()* function.

The overall implementation of the client component, as already mentioned, does not change too much from what has been introduced with the server software. The two following code-box report to the reader a simplified version (same conventions applied previously regarding try-catch clauses and output prints) of the initialisation function *init()* as well as the *receive()* one. One can notice that the *receive()* implementation uses, as seen for the server, a Message object. Due to the multiple strength identified for such component during the server discussion, the class has been translated to be used in this context too.

```
1  private void init() {
2      // Bind the socket to the port
3      socket = new Socket(ServerIP, serverPort);
4
5      // Initialize the input and output streams
6      this.in = new BufferedReader(
7          new InputStreamReader( socket.getInputStream() )
8          );
9      this.out = new PrintStream( socket.getOutputStream() );
10 }
```

```
1  private void receive() {
2      // Build the incoming string-stream until EOT is found
3      String input = String.valueOf( (char) this.in.read() );
4      while (! input.endsWith(ASCII.EOT()))
5          input += (char) this.in.read();
6
7      // Build a Message object from such stream
8      Message m = new Message( input );
9
10     // Notify observers if the Message obtained is legit
11     if (m.isLegit())
12         notifyObservers(m);
13 }
```

6.3.2 WiFi SmartConfig for trivial connectivity setup

The project developed and illustrated in the past sections of the report depicted a connected device in detail. As we have already mentioned while describing the firmware in the subsection 5.2.3, the smart plush toy is connected to the *Toygether system* via a WiFi connection. In order to accomplish a successful pairing with the plush toy to a parent device, one need beforehand to set-up the WiFi parameters of the toy such that it can be reached from the Android APP.

During the section of WiFi connectivity in the firmware part, the so-called `SmartConfig` hot-word has been used. In fact, the real challenge that occurred for both engineering and design point of view has been to implement a pipeline for configuring the micro-controller WiFi module regardless of the limited input (no screen and keyboard for such demanding task). Thanks to the development conducted by EspressIf (manufacturer of such micro-controller), we have been introduced to the possibility to configure such details by using the Android APP itself during the first pairing process.

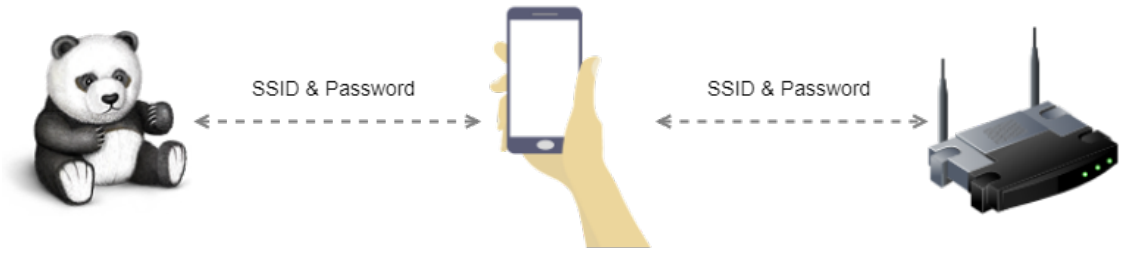


Figure 44: SmartConfig illustration

Figure 44 illustrates the interactions between the plush toy (on the left), the common WiFi router we all have at home (on the right) and the android APP itself. The goal is the ability to share the WiFi connection established on the smartphone (previously configured with the router) with the plush toy itself. Such operation is accomplished by the ESP-TOUCH protocol by Espressif-Systems (2016b): any WiFi enabled device (the smartphone for instance) can send UDP packets towards the WiFi access point (AP), activated by the micro-controller during this phase. Information about SSID and password of the wanted network are encoded into the Length field (see diagram below) of a sequence of such UDP packets retrieved by the micro-controller.

6	6	2	3	5	Variable	4
DA	**SA**	**Length**	**LLC**	**SNAP**	**DATA**	**FCS**

} UDP packet

The implementation of such protocol is based on the released Android library EspressIf deployed. Thanks to such base-code, both the UDP processing and the data encryption is safely managed. On top of such functions, we have implemented and/or adapted the specific design we need for our application in basically two classes: the first one will allow the retrieval of data like the SSID and hence covering the right part with respect to figure 44; on the other hand, the second class will use the EspressIf library to define the required UDP packets with the encoded data to send.

The goal is to give to the reader a general intuition on how such classes have been used to manage a functional *SmartConfig* pipeline. Firstly, the WiFi details (i.e. the SSID of the connected network for instance) need to be retrieved. The following code-boxes illustrates a few basic functions for such task.

```
public String getWifiConnectedSsid() {
    WifiInfo mWifiInfo = this.getConnectionInfo();
    String ssid = null;

    // Check if WiFi connection exists
    if (mWifiInfo != null && this.isWifiConnected()) {
        int len = mWifiInfo.getSSID().length();

        // Check SSID name format for retrival
        if ( mWifiInfo.getSSID().startsWith("\"")
                && mWifiInfo.getSSID().endsWith("\"") )
        {
            ssid = mWifiInfo.getSSID().substring(1, len - 1);
        } else {
            ssid = mWifiInfo.getSSID();
        }
    }
    return ssid;
}
```

```
private boolean isWifiConnected() {
    NetworkInfo mWiFiNetworkInfo = getWifiNetworkInfo();
    boolean isWifiConnected = false;

    if (mWiFiNetworkInfo != null)
        isWifiConnected = mWiFiNetworkInfo.isConnected();

    return isWifiConnected;
}
```

Android APIs have been fully exploited to retrieve the information about current networking details of the device. Such implementation allows the user to experience the most trivial setup possible because the information can be automatically provided. During the pairing procedure, the Android APP will require the user to be connected to the home WiFi. The only manual configuration needed will be the input of the WiFi password which cannot be taken automatically for security reasons.

As soon as the WiFi information required has been correctly retrieved, the implementation of the described ESP-TOUCH protocol can be achieved with the help of the library.

6.3.3 QR-codes to be scanned

The last building block of the APP that has been explored with major importance is the QR-code reader. In fact, in the communication protocol described in subsection 6.1, it has been repeatedly mentioned that each client (plush toy or Android user) would be assigned an unique identifier. Moreover, each interaction is made by constructing messages whose SOURCE and DESTINATION field are only represented by such identifiers, requiring to only keep track of the server network location to function. While the pairing procedure has been already defined with a set of messages within the communication protocol, such interactions require the source client to know the identifier of the intended destination.

The final product integrates a QR-code, like the one illustrated in figure 45, printed on the so-called "blackbox" introduced during the previous sections. Such illustration will be the "ID card" of each plush toy joining a new household, containing all the possible details that are required to uniquely identify such client. Along with the multitude of data that can be encapsulated, the plush toy assigned identifier will be contained. The Android APP can, therefore, retrieve such identifier by scanning the QR-code on the blackbox in order to construct the pairing message request (pipeline illustrated in detail during the communication protocol introduction, figure 37 for reference).

Figure 45: An example QR-code

The Google Camera API provides a customizable set of libraries to facilitate barcode reading tasks. In particular, the API provides the correct tools to efficiently read and decode QR-code data. To this intent, the API can be invoked upon pushing a button inside the current View and running a so-called "Activity-for-result". The upcoming section 6.3.4 will draw a clear overview around the concept of Activities in Android development, but the main understanding at this phase is that the current view delegates the QR-code reading procedure to the library and halts for the result to be delivered back.

```
1 button_scan.setOnClickListener {
2     val i = Intent(context, BarcodeCaptureActivity::class.java)
3     startActivityForResult(i, BARCODE_READER_REQUEST_CODE)
4 }
```

In fact, invoking the Google Camera API will activate the camera interface on the Android APP, which will wait for a QR-code to be read. Once such event is triggered, the activity of task will idle and the execution focus will return to the previous one that has been halted for the results to be received back. The following function *onActivityResult()* is automatically invoked in order to handle the data returned. Therefore, the method will check the correctness of the code retrieved in the scanned QR-code and subsequently share it with the next phase of the pairing pipeline. The reader needs to remember that once such code has been retrieved it will be used to communicate with the intended plush toy and accomplish the pairing procedure via the messages described previously.

```
override fun onActivityResult(request: Int,
                              result: Int, data: Intent?)
{
    // Check if the result is related to barcode before
    if (request != BARCODE_READER_REQUEST_CODE)
        super.onActivityResult(request, result, data)


    // Check for result correctness before accepting it
    if (result == CommonStatusCodes.SUCCESS && data != null)
    {
        val b_obj = BarcodeCaptureActivity.BarcodeObject
        val barcode = data.getParcelableExtra<Barcode>(b_obj)
        val p = barcode.cornerPoints

        passTheCodeToTheNextActivity(barcode.displayValue)
    }
}
```

6.3.4 Android Activity management

In this final part of the Android overview of the previous pages, we aim to illustrate the important so-called "Activity Life-cycle". This part will be clarifying concepts about Activities introduced in the previous pages and that has been left unexplained to the reader.

Due to intrinsic non-deterministic execution of mobile applications (i.e. the mail APP shows the inbox view when launched from its icon, but shows the mail composition view if called by another app for instance), the coding design cannot be the same as the one usually seen on firmware (based on a main function). The code illustrated in section 5 has been designed such that execution of it (turning the micro-controller ON) would deterministic-ally start at the same point every time. The Android official documentation defines activities as an object both encapsulated and focused, which provide I/O to the user (i.e. a GUI) for some result. By using activities, the Android APP is partitioned in a collection of isolated running environment, which can be directly be executed (i.e. the compose mail activity can be instantiated without passing by other parts of the code), and

thus providing a correct paradigm for the application development. At the beginning of this Android-focused subsection, figure 41 illustrated the general navigability of the developed APP by showing each view. Each of such views, therefore, have been implemented in the application as an activity-object.

Each activity inherits from the superclass a set of commonly used functions that will be helpful to handle their life-cycle behaviour later on. Appendix D provides to the reader a complete table of such methods alongside with description of their usage. Activities are usually defined with **at least** function *onCreate()* and *onPause()* implemented. Figure 46 on the following page illustrates how the life-cycle of activities evolves alongside the different methods used to drive the correct behaviour.

In subsubsection 6.3.1, the observer pattern has been introduced for its usage between views and the client software. We have illustrated the possibility for a particular view (i.e. an activity according to the current discussion) in the navigability of the APP to become an *Observer* of the client software. Whenever the latter would receive a new incoming message, the view would be notified of such change and receive the message to be parsed. However, we cannot give *Observer* privileges to all activities at the same time for the intended application. According to the communication protocol (see appendix C for reference) once the parent has requested a playing session to the paired plush toy, the latter needs to accept it by sending a message (MSG_ID equal to 2002) upon having the kid interacting with it. During this time, the parent is presented with a "waiting" activity until such message is not retrieved. During this period, the activity needs to be set to observe the client on hold for an interaction coming from the plush toy. As soon as the playing session is accepted, the "game" activity is proposed to the user in the foreground. In order to enable the interaction with the plush toy in both directions (sending and receiving messages for the game), this new activity needs to become an *Observer* as well. However the "waiting" activity has been pushed to the background by the execution of the game and, therefore, no longer requires to observe the client. The illustrated scenario is implemented by fully exploiting the Activity Life-Cycle described, which enables the application to promptly update its behaviours on activity changes. Following the information in figure 46, the required design is translated by correctly using functions *onResume()* and *onPause()* as shown in the following code-box.

```
1 override fun onResume() {
2     super.onResume()
3     Client.getInstance().addNewObserver(this)
4 }
5
6 override fun onPause() {
7     super.onPause()
8     Client.getInstance().deleteAnObserver(this)
9 }
```

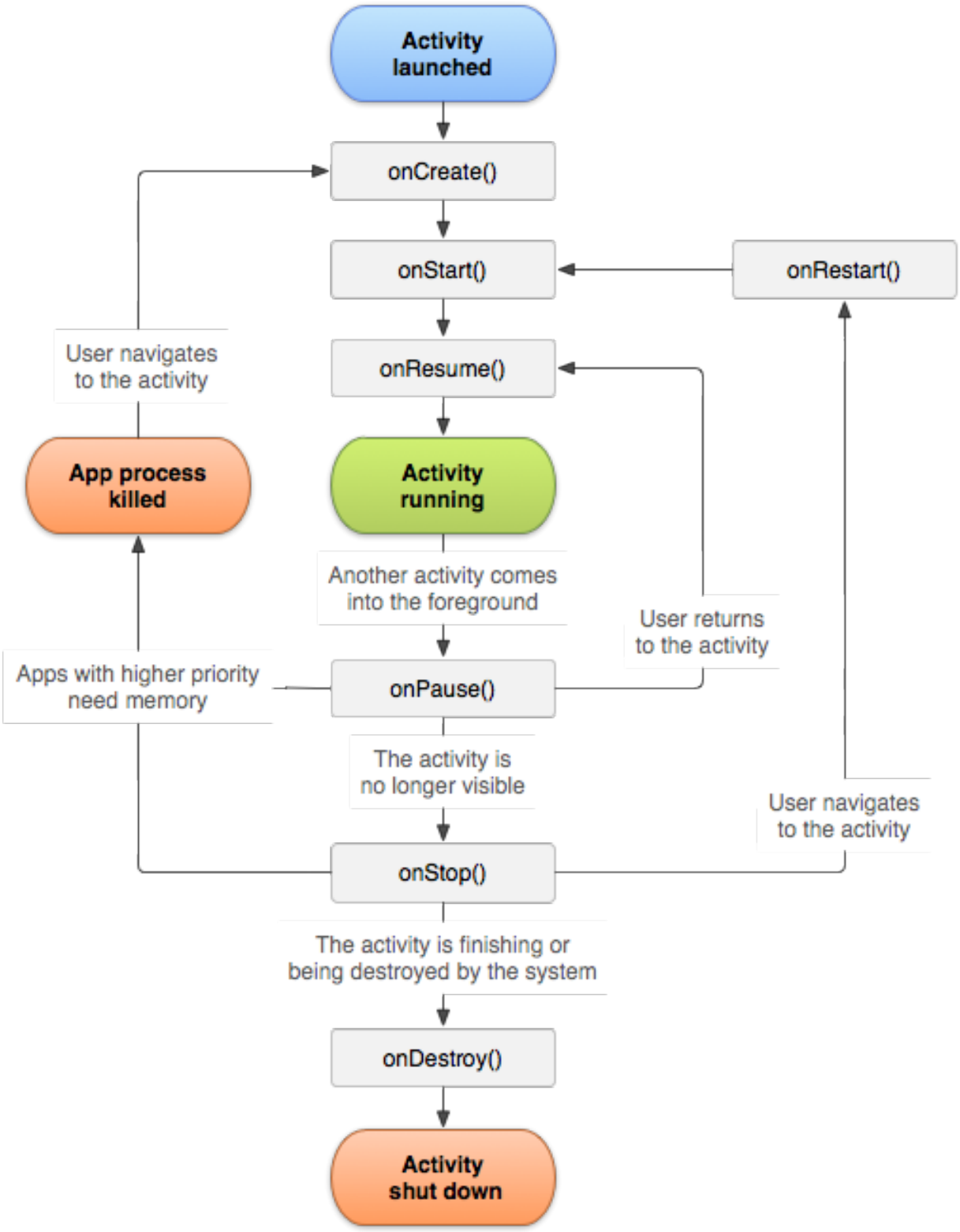


Figure 46: The life-cycle of an activity into an Android APP

6.4 Next steps for the software

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

7 Conclusion

We are done here.

A List of included contents

In the following appendix, a list of figures and a list of tables of the whole report is presented to the reader. This shall help him navigating through the content of the report, the different diagrams and tabulations.

List of Tables

1 Description of components within a message 37
2 Methods inherited for Activities in Android 64

List of Figures

1 Old but GOLD ! 2
2 Interaction design for child/parent game session 4
3 First breadboard solution with Arduino (realized with *fritzing*) 5
4 First breadboard solution with Arduino 6
5 ESP32-DevKitC 6
6 Comparison between ESP32, ESP8266 and Arduino UNO boards capabilities (Future-Electronics 2016) 7
7 ESP32-DevKitC functional overview (Espressif-Systems 2016a) 8
8 Final breadboard solution using ESP32-DevKitC 8
9 Miniature speaker of 2 grams, 16 mm diameter and 3.5 mm depth (Visaton 2015) 9
10 LED strip of tri-color RGB LEDs APA102C (LuxaLight 2015) 9
11 Power modes of the ESP32 microprocessor (Espressif-Systems 2018b) 10
12 Power consumption by power modes (Espressif-Systems 2018a) 10
13 Radio frequency power consumption specifications (Espressif-Systems 2018a) 10
14 NH22-175 rechargeable battery (Energizer 2018b) on the left and EN22 non-rechargeable battery (Energizer 2018a) on the right 11
15 ESP-WROOM-32 pins allocation 12
16 ESP32 module schematic 13
17 Power supply schematic 14
18 Micro USB & USB-UART schematic 14
19 LEDs voltage translation schematic 15
20 Switch buttons schematic 15
21 Connectors schematic 16
22 Electronic bill of materials 16
23 Second prototype of our connected plush toy, as presented at MS5. 19
24 First iteration of the blackbox design, including PCB, battery and loudspeaker 20
25 Layers of the first prototype: outer skin, plush stuffing and soft PCB 20
26 Details of top and lower layers of soft PCB 21
27 Test sample of heat-bonded conductive tracks 22
28 ESP32 built-in touch sensor 22
29 Examples of two soft sensors we experimented with 23

30 Example of pogo pin connector from Fairphone's electronics 24
31 Firmware architecture diagram with units and libraries 25
32 Flowchart for game interaction 30
33 ESP32 built-in touch sensor parameters 32
34 Test values for different touch/squeeze patterns with different users 32
35 APA102 communication protocol 33
36 Simplified diagram of the overall system 35
37 UML Sequence diagram for a communication pipeline example 39
38 UML class diagram for the Server software 40
39 Stack diagram for the Server software dependencies 41
40 Entity relationship diagram for the database structure 45
41 Android app overview 46
42 Stack diagram for the Android APP dependencies 47
43 UML class diagram for the observer pattern in the Android APP 47
44 SmartConfig illustration 49
45 An example QR-code 51
46 The life-cycle of an activity into an Android APP 54
47 Full schematic of the main PCB 59

B Main PCB schematic

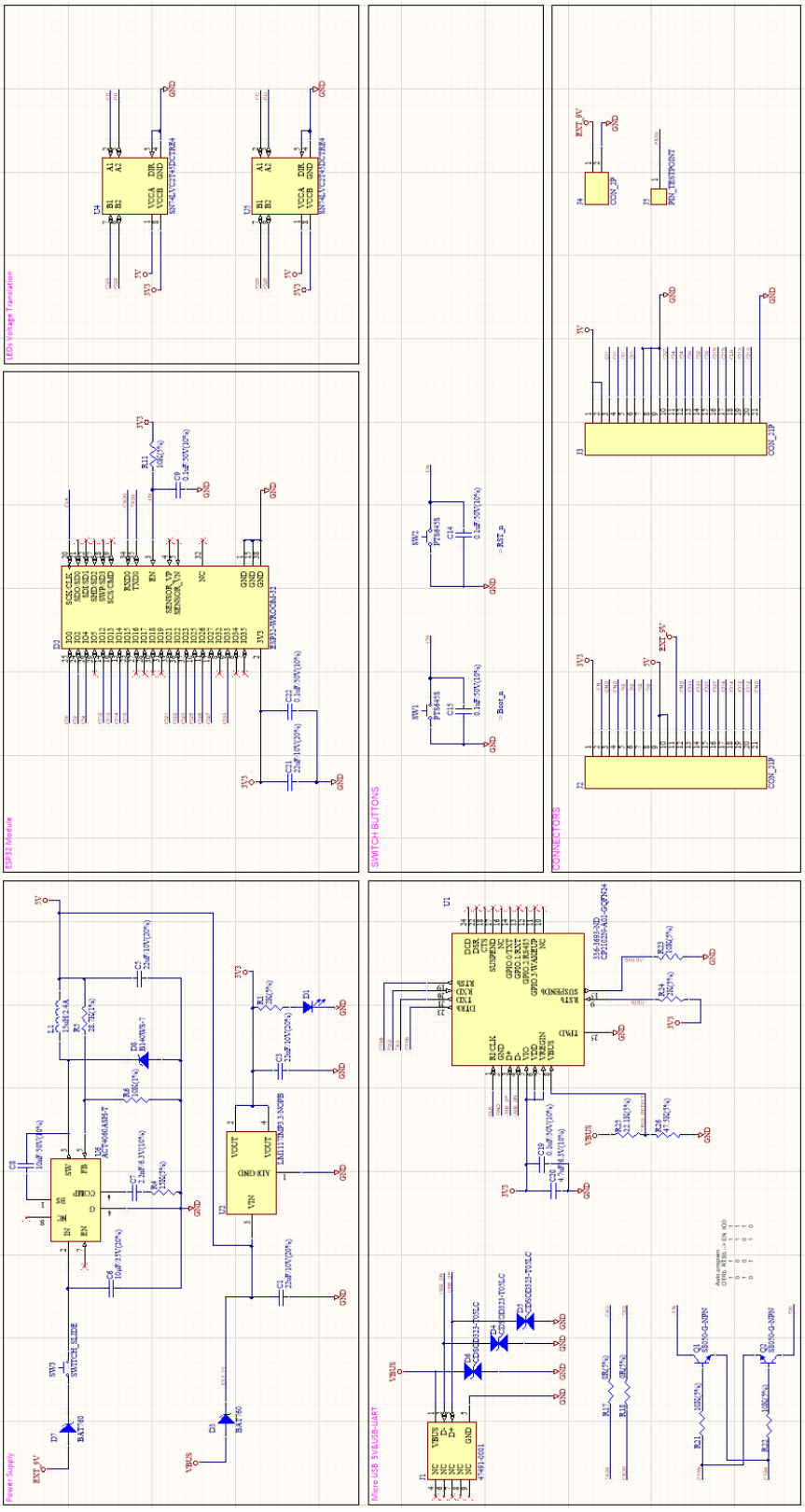
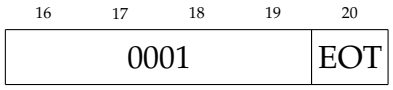


Figure 47: Full schematic of the main PCB

C Defined messages for the communication protocol

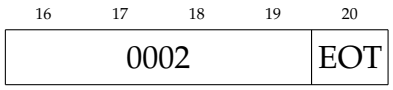
In the following appendix, every defined message is described in detail.

(0001) Introduction



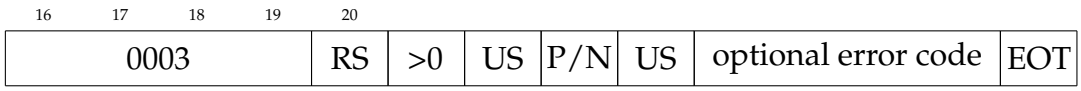
The message is sent from a client to the server upon connection. The SOURCE record is set to the Client identifier. When such identifier is yet unknown (before a user proceed with the log-in operation), the SOURCE record is set to 0000 by default. The server will associate the new connection (usually identified by its networking terms of IP address and telecommunication port) to the received identifier for future reference. If the message contained an empty SOURCE record (0000), such connection will be kept waiting until an identifier is assigned.

(0002) Close connection



The message is sent from a client to the server whenever the former quits its online status. Such event might be trigger by different reasons according to the client in use. The server closes any open socket with such client, freeing up some allocation space, and prepares to re-accept it during a future re-connection.

(0003) Acknowledgment of the requested operation



The message is sent from the server to a client whenever the latter requested an operation to be accomplished. It is constructed with up to two parameters: the former indicates whether the result of the operation was **P**ositive or **N**egative; on the other hand, the former is optionally added to a negative response in order to indicate extra details of the status.

Optional error codes

0001 Combination (Toy, Parent) does not exist

(0004) Paring acknowledgment by the toy

16	17	18	19	20				
0003				RS	1	US	User Parent ID	EOT

The message is sent by the toy-client to the server upon accepting a previously received request of paring by a parent-client. It is composed of a single parameter which contains the unique identifier of such parent. The server takes care of updating the database of associated pairs, as well as sending an acknowledgement to the parent which requested the pairing.

(0005) Requester for Parent to enter the system (sign up/in)

16	17	18	19	20								
0005				RS	3	US	F	US	username	US	password	EOT

The message is sent by the parent-client to the server whenever the former requires to either sign-up or sign-in to the system. The message is constructed with three parameters: a flag indicating whether the operation is sign-in (0) or sign-up (1), the user-name (an email address) and a password (encrypted in SHA-254).

(0006) Response for Parent to enter the system (sign up/in)

16	17	18	19	20				
0006				RS	1	US	User Parent ID	EOT

The message is sent by the server to the parent-client in response to a previously received request of access (MSG_ID equal to 0005). The only parameter is made of the assigned parent identifier that will be used by the client as SOURCE record for future messages.

Attention: if the parameter contains 0000, the access/registration in the system was refused by the server (possibly the user-name is either incorrect in the first case or already present for the second one).

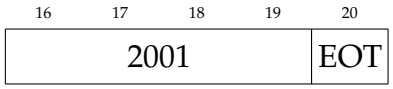
(1001) Paring request to the toy

16	17	18	19	20
1001				EOT

The message is sent by a parent-client to a toy-client in order to obtain "pairing privileges'. Without such privileges, the interactions between clients will be filtered by the server to prevent malicious behaviours. The request will be followed by a positive acknowledgment

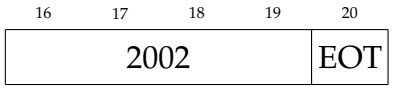
(see message with MSG_ID equal to 0003) if the pairing is successful within a timeout limit. If such response is not retrieved before the timeout, the request is considered refused and needs to be resent again.

(2001) Playing session started by the parent



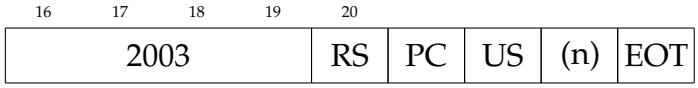
The message is sent by a parent-client to a paired toy-client in order to initiate a playing session with the latter. Then, the former client will wait for the toy to accept such playing session with a message which MSG_ID is equal to 2002.

(2002) Playing session accepted by the Toy



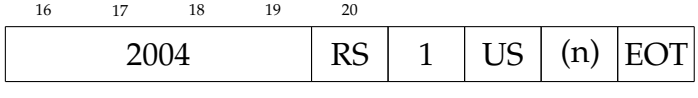
The message is sent by the toy-client to the parent-client in response to a playing session request by the latter.

(2003) Interactive area activated



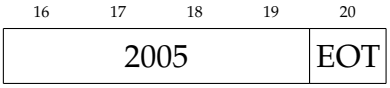
The message is sent by either a toy-client or a parent-client to the associated pair. It is used to indicate which area (LED on the toy, graphic on the Android APP) is requested to turn ON during the interaction. The only parameter of the message is composed of an integer number between 1 and 8, which indicates the area of interest. For example, if the parent intends to turn ON the 7th LED of the plsuh toy, the parameter of the message will contains the number seven.

(2004) Interactive area de-activated



The message is sent by either a toy-client or a parent-client to the associated pair. It is the reversed message for the previously described with MSG_ID equal to 2003. Each number passed as parameter of the message will be turned off as either LEDs on the plush toy or graphical interface on the Android APP.

(2005) Playing session terminated



The message is sent by the parent-client to the toy-client whenever the former decides to quit the playing session. The plush toy, upon receiving the message, is able to reset its status and be prepared for a possible new interaction in the future.

D Android activities inherited methods

Table 2: Methods inherited for Activities in Android

Method	**Description**
onCreate()	Called when the activity is first created. This is where you should do all of your normal static set up: create views, bind data to lists, etc. This method also provides you with a Bundle containing the activity's previously frozen state, if there was one. Always followed by onStart().
onRestart()	Called after your activity has been stopped, prior to it being started again. Always followed by onStart()
onStart()	Called when the activity is becoming visible to the user. Followed by onResume() if the activity comes to the foreground, or onStop() if it becomes hidden.
onResume()	Called when the activity will start interacting with the user. At this point your activity is at the top of the activity stack, with user input going to it. Always followed by onPause().
onPause()	Called when the system is about to start resuming a previous activity. This is typically used to commit unsaved changes to persistent data, stop animations and other things that may be consuming CPU, etc. Implementations of this method must be very quick because the next activity will not be resumed until this method returns. Followed by either onResume() if the activity returns back to the front, or onStop() if it becomes invisible to the user.
onStop()	Called when the activity is no longer visible to the user, because another activity has been resumed and is covering this one. This may happen either because a new activity is being started, an existing one is being brought in front of this one, or this one is being destroyed. Followed by either onRestart() if this activity is coming back to interact with the user, or onDestroy() if this activity is going away.
onDestroy()	The final call you receive before your activity is destroyed. This can happen either because the activity is finishing (someone called finish() on it, or because the system is temporarily destroying this instance of the activity to save space. You can distinguish between these two scenarios with the isFinishing() method.

E Work distribution and personal conclusions

Who did what and when? We can all write something like max 10 lines to say "I did this and this and this" and with whom we did it.

E.1 Tools

We used Trello, Slack, GDrive, Teamweek and a nice cardboard pannel but it was a bit too many tools and in the end we only used half of thembut while we used them they were cool to see who was doing what.

E.2 Who did what?

E.2.1 Chloe

I baked a cake and it was delicious. I went very often to ECAL and did many Skype calls with Estelle and Yann tried to explain his black magic to me. I took orders from Luca for the interactions and gave orders to Simone about the PCB shape.

E.2.2 Simone

E.2.3 Yann

E.3 Our goals for China and next semester

E.3.1 Chloe

I want to save WiFi SmartConfigs to EEPROM, implement sleep functions and implement a nice sound system. In mechanical design, I would like to implement a sleek user-friendly connector and bring the blackbox to its minimal form factor to minimise electronics intrusion in toy.

E.3.2 Simone

E.3.3 Yann

E.4 What I learnt

E.4.1 Chloe

I learned how to implement FW for a HW prototype from scratch while using libraries and existing functions and tweaking them to my needs.I adapted to designer work method to collaborate successively, and learned how to manage my work and team in a large project with many team members and supervisors with each their own tasks and requirements.

E.4.2 Simone

E.4.3 Yann

E.5 Timeline

Gantt chart for Spring semester

Gantt chart previsions for MS6/China and Autumn semester

References

Active-Semi (2009). *ACT4060A Wide Input 2A Step Down Converter Datasheet*. URL: https://active-semi.com/wp-content/uploads/ACT4060A_Datasheet.pdf.

Energizer (2018a). *EN22 Product Datasheet*. URL: https://datasheet.octopart.com/EN22-Energizer-datasheet-10978801.pdf.

— (2018b). *NH22-175 Product Datasheet*. URL: http://data.energizer.com/pdfs/nh22-175.pdf.

Espressif-Systems (2016a). *ESP32-DevKitC V4 Getting Started Guide*. URL: http://esp-idf.readthedocs.io/en/latest/get-started/get-started-devkitc.html.

— (2016b). *ESP-TOUCH User Guide*. URL: https://www.espressif.com/sites/default/files/documentation/30b-esp-touch_user_guide_en_v1.1_20160412_0.pdf.

— (2017). *ESP32-DevKitC Schematic V4*. URL: https://dl.espressif.com/dl/schematics/esp32_devkitc_v4-sch.pdf.

— (2018a). *ESP32 Datasheet V2.2*. URL: https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf.

— (2018b). *ESP32 Technical Reference Manual V3.3*. URL: https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf.

Future-Electronics (2016). *ESP32 Development Board (WIFI - Bluetooth)*. URL: https://store.fut-electronics.com/products/esp-32.

Gamma, Erich (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Education India.

LuxaLight (2015). *LED Strip APA102C Specification*. URL: https://www.ledtuning.nl/sites/default/files/downloads/201702/datasheet_apa102_1.pdf.

McReynolds, Emily et al. (2017). "Toys that listen: A study of parents, children, and internet-connected toys". In: *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, pp. 5197–5207.

Silicon-Labs (2017). *USBXpress Family CP2102N Datasheet V1.2*. URL: https://www.silabs.com/documents/public/data-sheets/cp2102n-datasheet.pdf.

Texas-Instruments (2016). *LM1117 800-mA Low-Dropout Linear Regulator Datasheet*. URL: http://www.ti.com/lit/ds/symlink/lm1117.pdf.

— (2017). *SN74LVC2T45 Dual-Bit Dual-Supply Bus Transceiver With Configurable Voltage Translation*. URL: http://www.ti.com/lit/ds/symlink/sn74lvc2t45.pdf.

Visaton (2015). *Miniature Speaker K16-8 Ohm Specification*. URL: https://www.mouser.ch/datasheet/2/700/k16_8-1285084.pdf.