Mark Awarded: __%

University of South Wales
Prifysgol De Cymru

UNIVERSITY OF SOUTH WALES- FCES

Assessment Cover Sheet and Feedback Form

2016/17

Module Code: NG3S900	**Module Title:** Advanced Embedded Systems	**Lecturer: BM**
Assignment No: 1/1	**No. of pages in total including this page: 155**	**Maximum Word Count: N/A**

Assignment Title: Case study: **Set of Tasks**

Part-1: Smart Autonomous Bot

and

Part-2: Smart Autonomous Bot - Porting with RTOS - µCosIII or alike

Tasks details: *see attached*

Date Set: 1st Nov'2016	**Submission Date: 28th March 2017**	**Feedback Date: 20** days from due

Section A: Record of Submission

Record of Submission and Plagiarism Declaration

I declare that this assignment is my own work and that the sources of information and material I have used (including the internet) have been fully identified and properly acknowledged as required in the referencing guidelines provided.

Student Number: 14031329

You are required to acknowledge that you have read the above statement by writing your student number(s) above.

(If this is a group assignment, please provide the student numbers of **ALL** group members)

Details of Submission

Note that all work handed in after the submission date and within 5 working days will be capped at 40%. No marks will be awarded if the assignment is submitted after the late submission date unless mitigating circumstances are applied for and accepted.

- IT IS YOUR RESPONSIBILITY TO KEEP A RECORD OF ALL WORK SUBMITTED.
- An electronic copy of your work should be submitted via Blackboard.
- Work should also be submitted to the member of academic staff responsible for setting your work.
- Work not submitted to the lecturer responsible may, **exceptionally**, be submitted (on the submission date) to the reception of the Faculty of Advanced Technology, which is on the 2nd floor of G block (Room G221) where a receipt will be issued.

Mitigating Circumstances: if there are any exceptional circumstances which may have affected your ability to undertake or submit this assignment, make sure you contact the Faculty Advice Shop on 01443 482540 (G221).

You are required to acknowledge that you have read the above statements by writing your student number (s) in the box:	**Student Number:** **14031329**

Section B : Marking and Assessment

This assignment will be marked out of 100% This assignment contributes to 30% of the total module marks. This assignment is bonded / non- bonded. Details :	It is estimated that you should spend approximately **24** hours on this written assignment.
Learning Outcomes- http://icis.southwales.ac.uk	

This assignment addresses the following learning outcome(s) of the module:

LO1. The student will be able to apply formal design methodologies in the development of embedded solutions

LO2: The student will be able to critically analyse an embedded system problem and select the appropriate design methodology to implement a software solution.

For this assignment, the following learning outcomes supplements should also be taken into account:

LO3.Critically evaluate a User Requirement Specification and identify the appropriate design methodology required to provide solutions meeting the functionality and requirements defined.

LO4.Will be able to design complex embedded solutions using a pre-emptive RTOS that provide reliable applications meeting measured hard real time constraints.

Marking Scheme	Marks Awarded	Marks Available
Task-1-Background search & Design Methodology to develop detailed**:** (a) Analysis (b) Hardware requirements (c) Software Requirements		**15**
Task-2- Design (a) Layout diagram/PCB (b) Bot Chassis + BOM (c) Write the Algorithm		**15**
Task-3- Implementation (a) Assemble the Bot components (b) Write the code (c) Test including designing a testing strategy		**20**
Task-4-Background search & Design with RTOS µCosIII. Identify the: (a) Porting features 1. CPU components 2. Libraries components 3. RTOS core components 4. Ports e.g. Renesas processor(s) 5. Source core code components 6. Creating tasks 7. Scheduling and resource management 8. Tasks communications (b) Board Support Package (BSP) for specific development boards (Evalboards) http://micrium.com/rtos/ucosiii/overview/		**10**
Task-5: Consider the autonomous Bot application developed previously or similar, using the YRDKRX63N featuring the RX63N microcontroller. (a) Discuss the disadvantages of such linear/sequential infinite loop approach. For this, it is advisable to carry out a new analysis of the URS in view of producing a multitasking system. Inter-task communication should be considered when required. (b) In light of the arguments in Task-5, state the advantages of introducing RTOS in embedded systems (c) Draw a block diagram/flow diagram showing the changes you intend to implement to the solution of Part-1 or similar https://www.youtube.com/watch?v=EmLIHhfEEm4		**10**

Task-6- Implement the changes (Code in RTOS) For this code, specify clearly: • Creation of tasks • Organisation of tasks • Communication between tasks The role of RTOS in the overall control of the application		**15**
Task-7: Full Report Write a Full Report		**15**
Total		**100**

PLEASE NOTE ASPECTS OF YOUR REPORT					
	Very Well	Well	Adequately	Poorly	Very Poorly
Introduction					
Methods					
Quality of artefact (H/W) and code (S/W)					
Analysis, conclusions etc.					
Conclusion					

PART C : MARKER'S FEEDBACK

Lecturer's Comments:

Feedback/feed-forward (linked to assessment criteria):

Lecturer's signature:	**Date:**	**Mark awarded:**

All marks are subject to confirmation by the Board of Examiners

[This page was intentionally left blank]

University of South Wales – Year 3 MEng Computer Systems Engineering

14031329 – Miguel Santos

NG3S900 – Advanced Embedded Systems

Smart Autonomous Bot

Using Micrium µCOS-III and Renesas RX63N

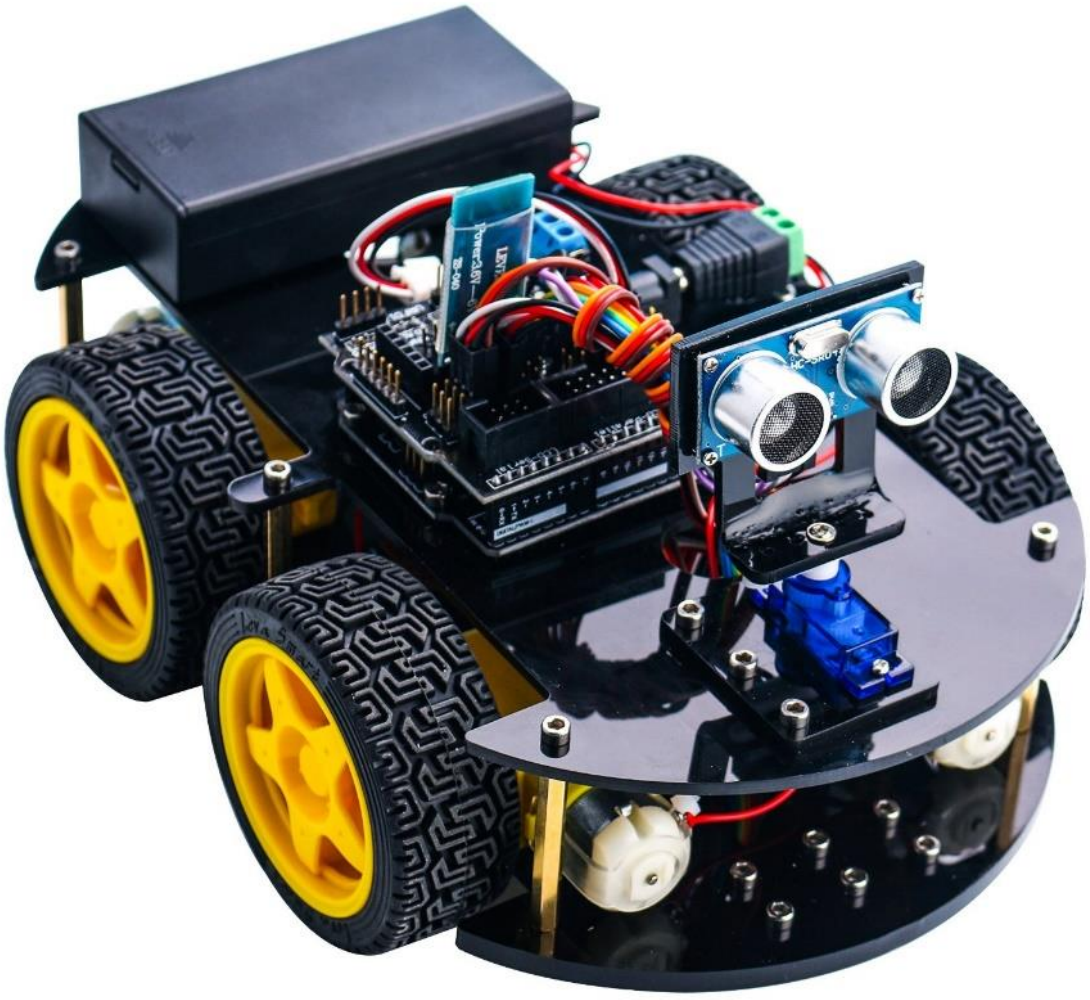


TABLE OF CONTENTS

TABLE OF FIGURES

INTRODUCTION

In the ambit of the module NG3S900 – Advanced Embedded Systems, this report will cover the first and only assignment that was given by the lecturer for the 3rd year students.

This document is divided into 6 main sections. They cover the tasks 1, 2, up until 6.

The first task will list a set of requirements (Hardware and Software) for a given Use Case scenario. It will also contextualise the User Requirements Specifications (URS) in order to provide a good perspective of what the system should look like, what it should do, how it should behave, and how it is to be built.

The second task will take the information presented in the first task and will develop on it in a more physical way. This means, Electronic Schematics, PCB layouts, the physical structure of the final system and the algorithm of the software that will be run on this system will be shown here.

The third task will show how the system will be physically built and how the code will be written using the linear program method. It also describes how the project is to be tested.

The second part of the report, which starts with the 4th task, starts by identifying and explaining the architectural components of the Micrium RTOS.

The 5th task weighs the advantages and disadvantages of the linear program approach versus the RTOS approach.

Finally, the 6th task will contain information of how the code was ported from linear code (from task 3) to RTOS code.

TASK 1 – BACKGROUND RESEARCH AND DESIGN METHODOLOGY

1.A) ANALYSIS

The case study provided in this assignment is called the Smart Autonomous Bot. This case study aims at analysing, designing, developing and implementing a small car that can drive itself through a specific environment by using a set of sensors in order to prevent collisions.

The car will be conceived using electronic (ECAD) and software (IDEs) tools available to the students by the University.

The User Requirement Specifications (URS) are:

Original URS	Analysis - Comprehension
1. USW is thinking of automating its mail delivery to various buildings in its Trefforest campus. Such a system would involve a vehicle that will be able to navigate safely and autonomously, through a route, recognizing its position and avoiding any hazards in its path. For the purpose of this project, a scaled down prototype should be produced that should consist in its first prototype: • Mechanical components to be determined • Electrical components that include a Power Unit capable to drive the mechanical parts and provide enough juice for the rest of the system. • Electronic components built around a command and control unit-CCU that will allow the vehicle to scan its environment to determine the path to follow on a point to point strategy.	• A self-driving vehicle that carries a package/load for delivery throughout the USW Campus. • This vehicle must learn and recognize the path in order to prevent unwanted collisions. • Three major components are present in this design: the mechanical parts, the electrical components and the software which controls both mechanical and electrical parts.

2. The prototype is expected to be programmable so it can serve many purposes. For this, any future enhancements would ideally be accommodated in the original design. For example, it is viewed that many of these units might be built and used jointly for a purpose that might arise in the future. So, communication between each vehicle or via a central command should be envisaged.	• The prototype needs to be designed as a multi-purposed vehicle. • It will require a communication link with other vehicles. • These vehicles could be connected to one central server (centralized communication). • The alternative option would be to implement a mesh network and use that instead for communication. The user/programmer would treat any mesh node as the entry point to the network (decentralised communication). • It would also be a good idea to introduce the feature of wireless re-programming.
3. Any other features that are omitted or not obviously stated in this document that will enhance the vehicle might be incorporated prior to an approval based on the planning (timescale & budget).	• More features may be added in the future, which will be liable for approval due to the project timescale and budget.

4. The function of the system is to monitor and report on the state of each leg of the path. Monitoring consists of scanning the environment, checking for any obstacles in view of avoiding them. Reporting consists of allowing processor to interrogate the sensors and working the location and the distance of any obstacles over a period of time the format of a scanning is: Sensor no. Sensor Location. Distance This consists of a sensor number, the location of the sensor which originated the reading, and, finally, the distance of the obstacle. Each reading is issued every second.	• The system/CPU will periodically request/poll and store the data from the sensors used in the mechanism for locomotion which will indicate possible collisions and will offer a way to prevent them in the future. • This data will be packed in memory RAM and stored with the format: Sensor no. Sensor Location. Distance on any non-volatile memory storage, such as Flash memory, EEPROM or SD Card.
5. On receiving a reading that is outside safety limits the system should take appropriate actions to steer the AV safely and securely away from the obstacle(s).	• Whenever the sensor readings indicate an imminent collision, the microcontroller will control the motors to steer away from the obstacle. • A good feature that could be implemented would be to scan the environment first and decide what direction to turn next. • Also, the use of GPS and machine learning algorithms could help the software's algorithm in pre-recognising static obstacles before the vehicle even gets close to them.

6. The system should allow the operator to check on the state of operation of each sensor over the past 48 hours via an interface display that should be as flexible as possible.	• The machine should record every sensor data for a period of at least 48 hours so that the operator can examine the occurrences throughout that time. Also, the vehicle should include an interface which allows the operator to easily and quickly analyse the data. • Multiple interfaces could be used to visualise this data, such as an LCD screen, the Wi-Fi link or even Bluetooth.
7. The system response should be the maximum possible achievable	• The vehicle should act as quick as possible whenever an obstacle is encountered, in order to prevent catastrophic failure.
8. The system should be programmed in C. All file handling should be performed by calls on operating system routines. The fixed price for the contract will be £20,000 which should take into account in its conception the suggestions in Appendix-A.	• The software must be written using C language with the help of Real Time Operating Systems (RTOS) which will allow the handling of log data files. • Not only is RTOS good for using software stacks (TCP/IP, File systems, etc…), it is most importantly used for time-constrained applications. This is the main reason why this project is using RTOS. • The allocated budget for this project is £20,000

Following is a table that summarises the user requirements previously described by using nouns, verbs and adverbs:

Nouns & Noun Phrases	Verbs and Adverbs
Autonomous Sensor Motor Servo Data Position Velocity Speed Wireless Communication Hazard Wall Line Display Storage Log Files RTC Time C Language RTOS Responsive Reliable Multi-Purpose	Read Sensor Scan environment Detect wall Decide direction Drive forward / left / right / back Communicate wirelessly Determine global position (GPS) Store events into log files

Finally, by reading the previous analysis in detail, we can conclude that the car is made of 3 major components:

1. Mechanical Parts – Bot chassis, DC and Servo motors
2. Electrical Components – Microcontroller Board, PCB, Resistors, Capacitors, H-Bridge, Wireless Modules, Sensors, etc…
3. Software for controlling the mechanical parts based on what the sensors output into the Microcontroller

Each of these three will be described with great detail on the following sections of this report.

On the following image is a high level view diagram that illustrates both the car's hardware and software requirements.

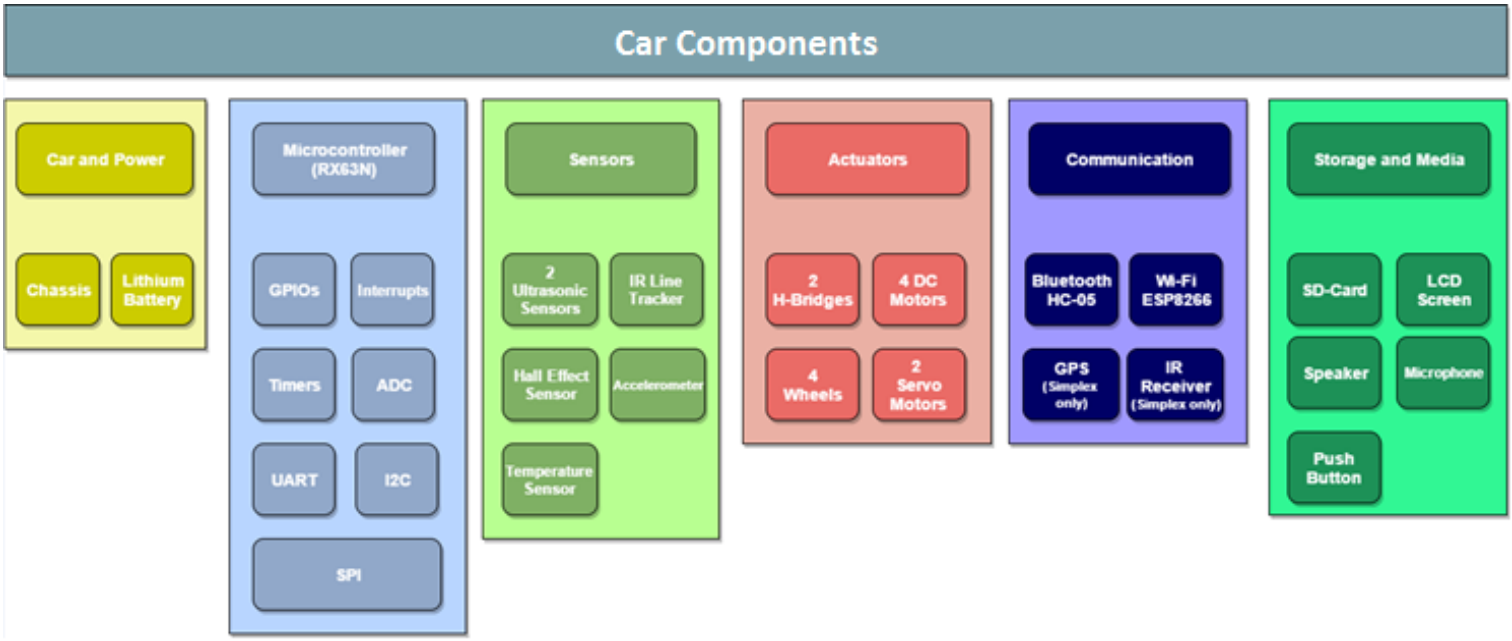


Figure 1 - High Level View of the Hardware and Software Requirements

As can be seen, each component has been categorised into different sections using different colours to help with the identification.

Out of these components, the ones that can't be programmed are the car chassis and lithium battery.

This diagram is the result of the initial analysis from the URS. It will give a basis to the Software Requirements task, and clearly to the rest of the project.

We can therefore conclude that the development of the URS and its Analysis are the most important stages of a project cycle, even if no technical implementation is involved.

1.B) HARDWARE REQUIREMENTS

In this section, the hardware requirements used in the bot car will be listed and this will be followed with an explanation of why the component was added to the system.

1.B.1.1) CAR CHASSIS

The actual structure of the car. This holds the entire car together, including the microcontroller, the sensors and the motors.

Figure 2 - Car Chassis

1.B.1.2) MICROCONTROLLER RENESAS BOARD YRDKRX63N

This board will be responsible for measuring the sensors and for controlling the motors.

Figure 3 - YRDKRX63N Renesas Board

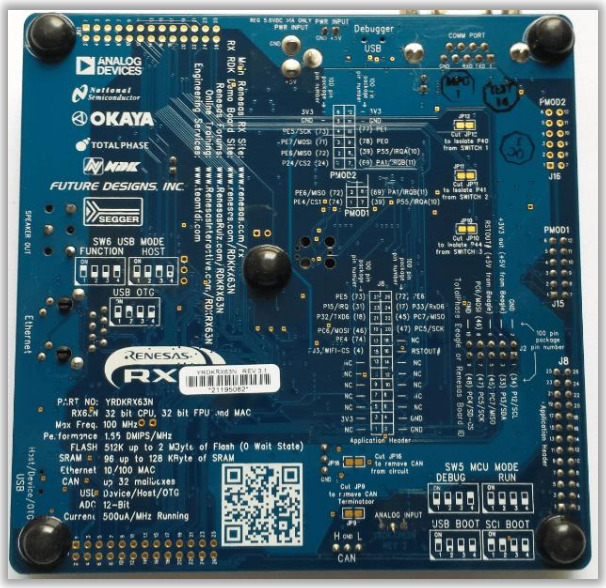


Figure 4 - YRDKRX63N Renesas Board - Back side

1.B.1.3) FOUR DC MOTORS + WHEELS

Used for the locomotion of the car.

Figure 5 - DC Motors + Wheels

1.B.1.4) H-BRIDGE L298N X 2

Used to control the four motors. Each H-Bridge is capable of controlling 2 motors, thus, 2 H-Bridges will be required in order to achieve high RPM on the 4 DC motors.

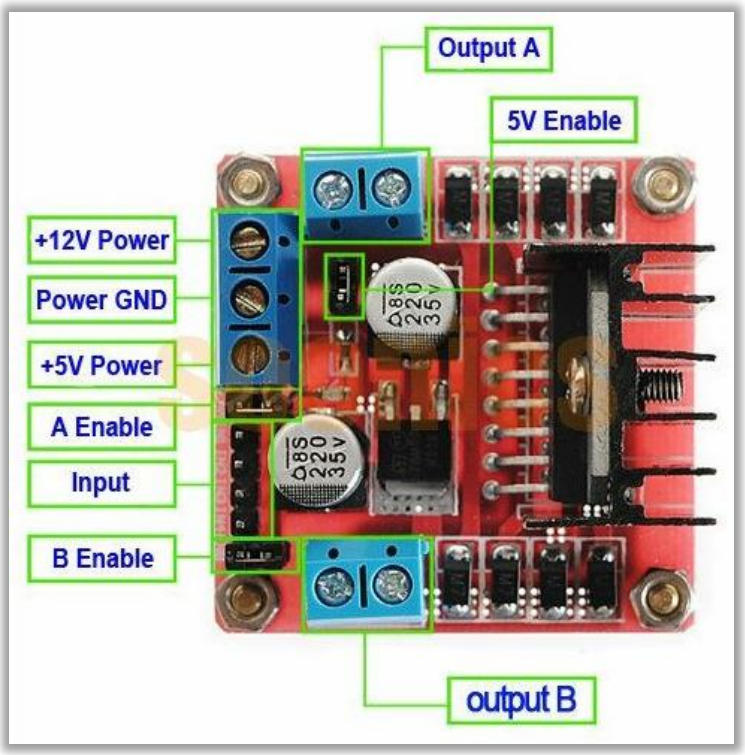


Figure 6 - H-Bridge L298N

1.B.1.5) ULTRASONIC SENSOR HC-SR04 X 2

This sensor is capable of detecting the presence of obstacles by emitting and receiving ultrasonic pulses into the nearby environment.

Two sensors will be used. The first will be mounted on a structure, which will scan the environment dynamically. The second sensor will be fixed on the front of the car, which will be used for detecting walls and will serve as a redundant sensor.

Figure 7 - Ultrasonic Sensor HC-SR04

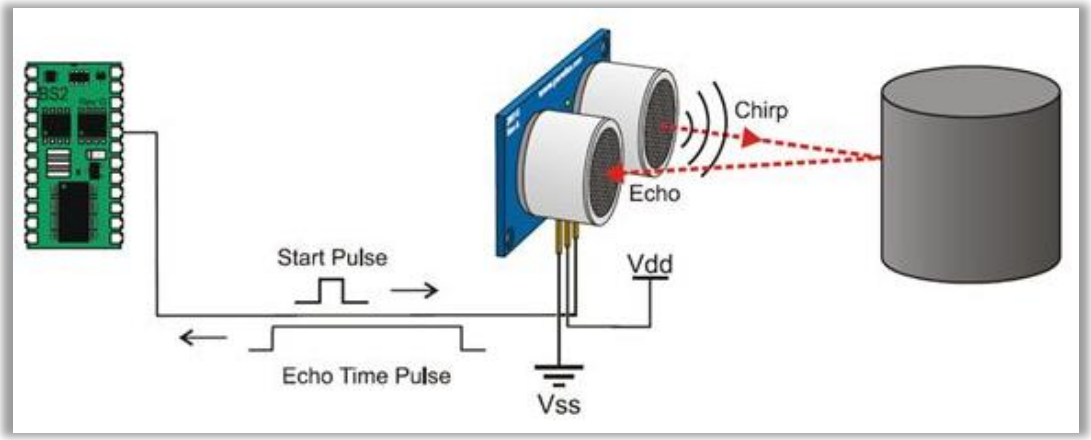


Figure 8 - How the HC-SR04 works

1.B.1.6) SERVO MOTOR SG90 X 2

These two servo motors will allow the microcontroller to scan the environment by rotating the ultrasonic sensor on a mount, just like radar. Two servos are being used in order to scan in both the X and Y axis.

Figure 9 - Servo SG90

1.B.1.7) ULTRASONIC SENSOR MOUNT

Since the ultrasonic sensor needs to rotate, it will have to be mounted on a structure that allows rotation, such as the one shown on the image below.

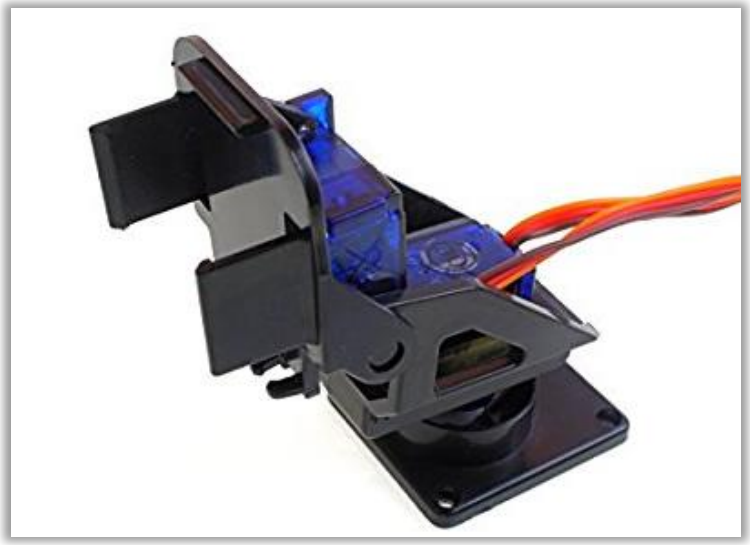


Figure 10 - Mount for the Ultrasonic Sensor

1.B.1.7) INFRARED RECEIVER VS1838B

The most basic way of wirelessly communicating with the car is by using a remote controller. This remote controller will use an infrared transmitter, therefore, for this reason, the IR receiver VS1838B will be installed on the car.

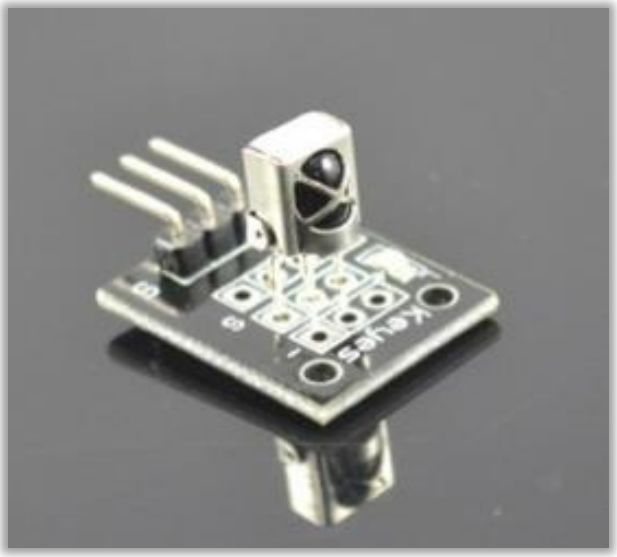


Figure 11 - IR Receiver VS1838B

1.B.1.8) INFRARED TRACKER SENSOR

This sensor will allow the car to follow a black line on the floor (or vice-versa). The way it works is each IR transmitter keeps outputting an optical IR wave and the receiver keeps trying to receive the echo of the same wave. When the signal is reflected back off a surface, the green led turns on, signalling the microcontroller that an object is present (in this case, the floor). If no signal is echoed back then an object with a dark colour (which absorbs the emitted IR light) was detected, thus, allowing the car to follow it.

Figure 12 - IR Tracker Sensor

1.B.1.9) DIGITAL HALL EFFECT SENSOR KY-003

This component simply detects if a magnet is present or not. Its output is digital. 0v output means no magnet is present and 5v if a magnet is nearby.

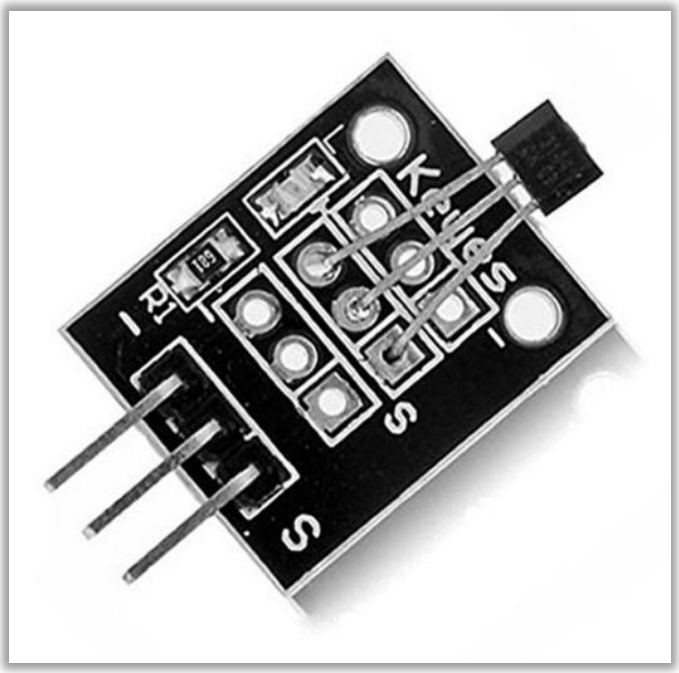


Figure 13 - Hall Effect Sensor KY-003

1.B.1.10) BLUETOOTH MODULE HC-05

This module will be used to allow medium range communication between Smartphone <-> Microcontroller.

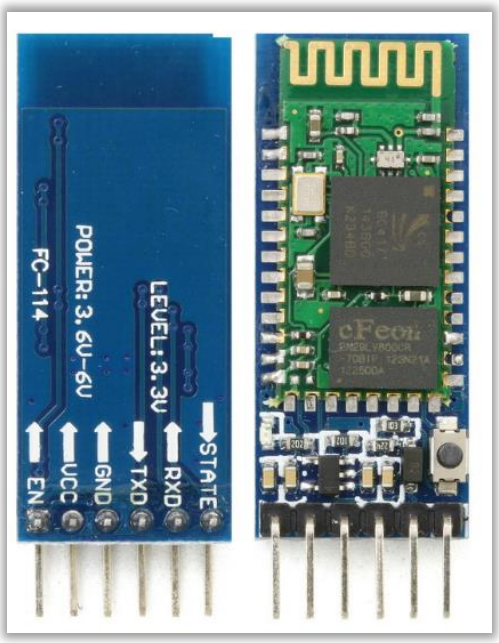


Figure 14 – Bluetooth Module HC-05

1.B.1.11) WI-FI MODULE ESP8266

Used for high range communication between Router <-> Microcontroller. This allows the microcontroller to be accessible anywhere on the internet.

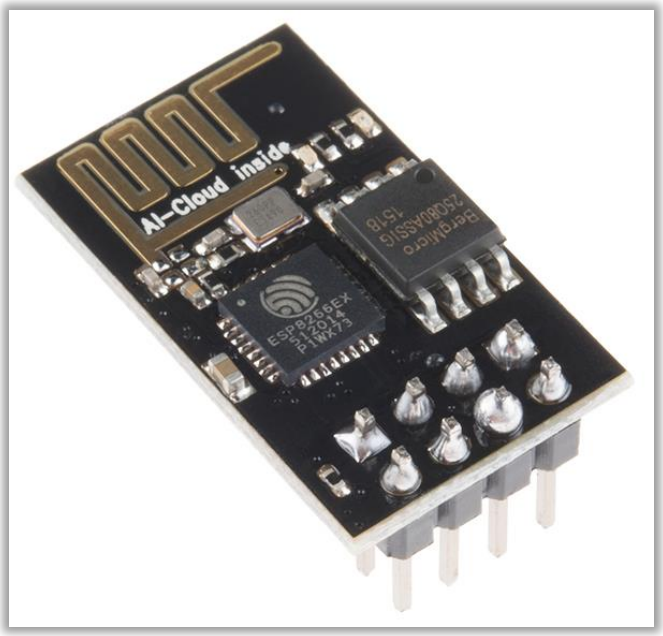


Figure 15 – Wi-Fi Module ESP8266

1.B.1.12) GPS MODULE U-BLOX NEO-6M

Used for locating the car in the physical world. It can be useful for generating calculations and to improve the algorithm of the car.

Figure 16 - GPS Module U-blox NEO-6M

1.B.1.13) LITHIUM BATTERY

Finally, a Lithium Battery to power the entire car.

Its specs are: 30C rating, output 11.1V, max output current 66 A and 2200mAh of autonomy.

Figure 17 - Lithium Battery - 30C 11.1V 66A 2200mAh

The hardware requirements (mostly peripherals) that belong to the Microcontroller RX63N and YRDKRX63N Demonstration board are:

1.B.2.1) GPIOS

The RX63N has 11 available 8-bit GPIO ports that range from port 0 to 9, and A to J.

b7	b6	b5	b4	b3	b2	b1	b0
Value after reset:							
0	0	0	0	0	0	0	0

Each port is composed of 8 pins and each has 3 main registers assigned to it. The registers are:

1) PDR – Port Direction Register – which sets the ports as either output (1) or input (0)
2) PODR – Port Output Data Register – which actually drives the physical wires to low impedance and output a certain voltage (5v - binary 1, 0v – binary 0) with a certain limited measure of frequency
3) PIDR – Port Input Data Register, which does the inverse of the PODR, by putting the tristate NMOS transistor gates in high impedance and allowing the microcontroller to read the digital voltages.

Other relevant registers:

1) **PMR** – Port Mode Register
2) **ODR** – Open Drain Control Register
3) **PCR** – Pull-Up Resistor Control Register
4) **DSCR** – Drive Capacity Control Register
5) **PSRx** – Port Switching Register A/B

These GPIO pins can be accessed by using the following C header declaration found in iodefine.h:

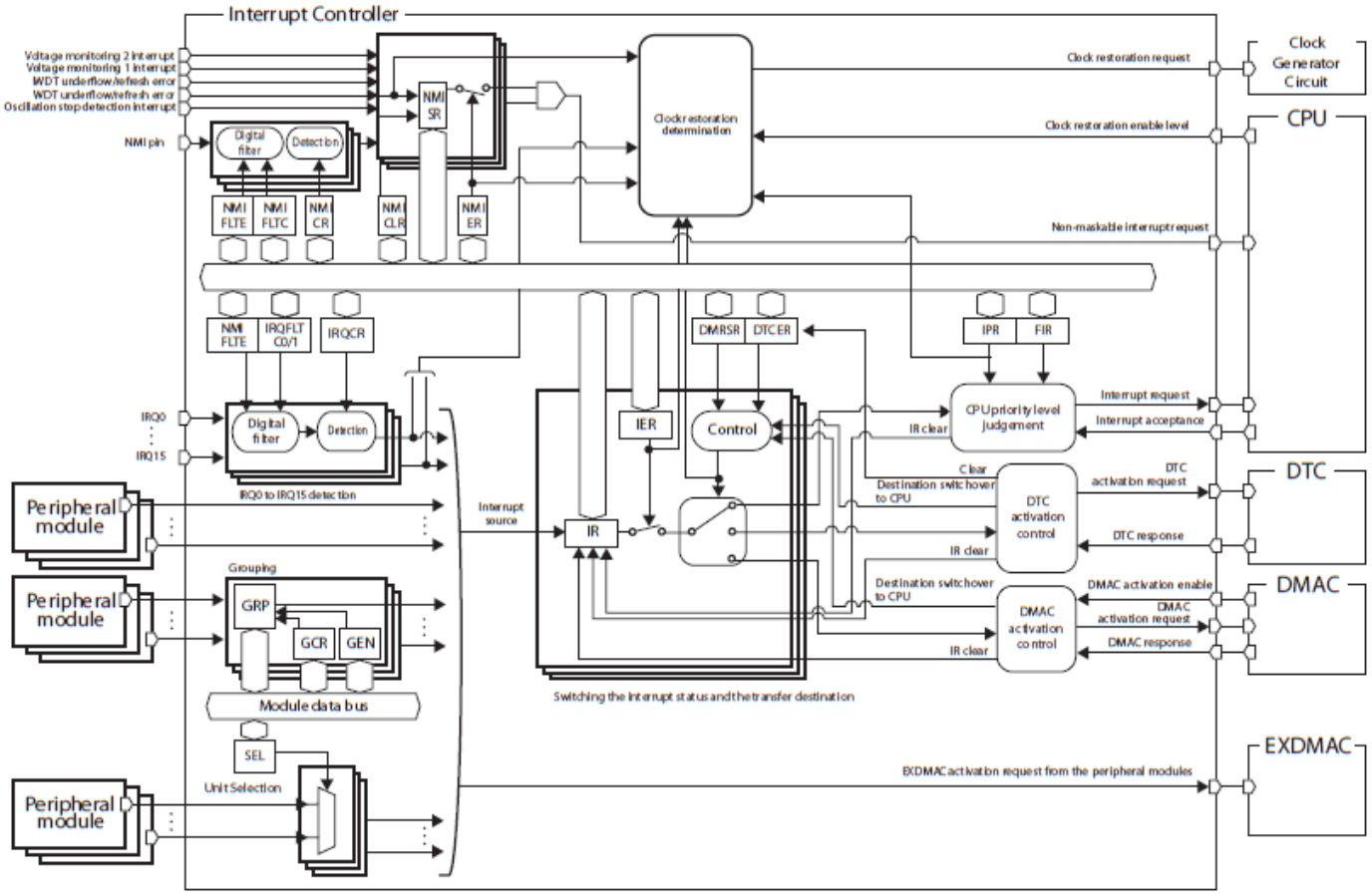
```
1   struct st_portn /* n = 0-9 and A-J */
2   {
3           union {
4                   unsigned char BYTE;
5                   struct {
6                           unsigned char B7:1;
7                           unsigned char B6:1;
8                           unsigned char B5:1;
9                           unsigned char B4:1;
10                          unsigned char B3:1;
11                          unsigned char B2:1;
12                          unsigned char B1:1;
13                          unsigned char B0:1;
14                  } BIT;
15          } PDR;
16          char wk0[31];
17          union {
18                  unsigned char BYTE;
19                  struct {
20                          unsigned char B7:1;
21                          unsigned char B6:1;
22                          unsigned char B5:1;
23                          unsigned char B4:1;
24                          unsigned char B3:1;
25                          unsigned char B2:1;
26                          unsigned char B1:1;
27                          unsigned char B0:1;
28                  } BIT;
29          } PODR;
30          char wk1[31];
31          union {
32                  unsigned char BYTE;
33                  struct {
34                          unsigned char B7:1;
35                          unsigned char B6:1;
36                          unsigned char B5:1;
37                          unsigned char B4:1;
38                          unsigned char B3:1;
39                          unsigned char B2:1;
40                          unsigned char B1:1;
41                          unsigned char B0:1;
42                  } BIT;
43          } PIDR;
44  }
```

The GPIO pins will be mainly used for controlling the H-Bridge and for other generalised cases. The other uses include turning LEDs on and reading basic sensors such as the hall effect sensor.

1.B.2.2) INTERRUPTS

On the RX63N, the interrupts are all managed by the ICU – Interrupt Control Unit.

This unit receives multiple signals from multiple external sources, such as an infrared sensor, ultrasonic sensor, or even push button, and redirects the signal to the CPU indicating the presence of an interrupt from one of the peripherals.



* Figure taken from RX63N Introduction book page 358 by Renesas

Figure 18 - ICU Diagram

The ICU is highly configurable and supports the following features:

ITEM		DESCRIPTION
Interrupts	Peripheral function interrupts	■ Interrupts from peripheral modules ■ Number of sources: 146 ■ Interrupt detection: Edge detection/level detection Edge detection or level detection is determined for each source of connected peripheral modules.
	External pin interrupts	■ Interrupts from pins IRQ15 to IRQ0 ■ Number of sources: 16 ■ Interrupt detection: Low level, falling edge, rising edge, both rising and falling edges One of these detection methods can be set for each source.
	Software interrupt	■ Interrupt generated by writing to a register ■ One interrupt source
	Interrupt priority	Specified by registers.
	Fast interrupt function	Faster interrupt processing of the CPU can be set only for a single interrupt source.
	DTC/DMACA control	■ The DTC and DMACA can be activated by interrupt sources. ■ Number of DTC activating sources: 102 (85 peripheral function interrupts + 16 external pin interrupts + 1 software interrupt) ■ Number of DMACA activating sources: 45(41 peripheral function interrupts + 4 external pin interrupts)
Non-maskable interrupts	NMI pin interrupts	■ Interrupt from the NMI pin ■ Interrupt detection: Falling edge/rising edge
	Power-voltage falling detection	Interrupt during power-voltage fall detection
	Oscillation stop detection	Interrupt during oscillation stop detection
Return from power-down modes		■ Sleep mode: Return is initiated by non-maskable interrupts or any other interrupt source. ■ All-module clock stop mode: Return is initiated by non-maskable interrupts, IRQ15 to IRQ0 interrupts, WDT interrupts, timer interrupts, USB resume interrupts, or RTC alarm interrupts. ■ Software standby mode: Return is initiated by non-maskable interrupts, IRQ15 to IRQ0 interrupts USB resume interrupts, or RTC alarm interrupts.

* Figure taken from RX63N Introduction book page 359 by Renesas

Figure 19 - ICU Specifications

As is the case with GPIOs, the Interrupts also have a range of registers that can be used.

The most common 8-bit register is the IR flag, which alerts the CPU when the ICU has an interrupt that needs to be executed.

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b0	IR	Interrupt status flag	0: No interrupt request is generated	R/W*
b7 to b1	—	Reserved	These bits are read as 0. The write value should always be 0.	R/W
Note: *For edge-detected sources, only 0 can be written to this bit, which clears the flag, and writing 1 to the bit is prohibited. Writing is not possible if the source is level-detected.				

* Figure taken from RX63N Introduction book page 360 by Renesas

Figure 20 - Interrupts - IR Flag

In this case, only the first bit can be used.

Another important register is the IERm (the m ranges from 2h to 1Fh).

Address: 0008 7202h to 0008 721Fh

	b7	b6	b5	b4	b3	b2	b1	b0
	IEN7	IEN6	IEN5	IEN4	IEN3	IEN2	IEN1	IEN0
Value after reset:	0	0	0	0	0	0	0	0

* Figure taken from RX63N Introduction book page 360 by Renesas

Figure 21 - Interrupts - IERm Register

Setting each bit to binary 1 will allow the correspondent interrupt to be executed by the CPU.

Other relevant registers:

1) **IPRm** (m = 0h..18h) – Interrupt Priority Register
2) **FIR** – Fast Interrupt Register
3) **SWINTR** – Software Interrupt Activation Register
4) **IRQCi** (i = 0 to 15) – IRQ Control Register
5) **NMIER** – Non-Maskable Interrupt Enable Register
6) **NMISR** – Non-Maskable Interrupt Status Register
7) **NMICLR** – Non-Maskable Interrupt Status Clear Register
8) **NMICR** – NMI Pin Interrupt Control Register.

The interrupts will be mainly used for the Infrared Line Tracker, Ultrasonic Sensor and Push Buttons. It could also replace the hall effect sensor being read using GPIOs.

1.B.2.3) TIMERS

There are three different types of timing mechanisms used in the RX63N. The normal highly configurable **Timer**, the equally configurable **Compare Match Timer** and the **Real Time Clock** (RTC), which allows the microcontroller to know the real time without having to synchronise on every start up with an external communication link.

In respect to the normal timers, the microcontroller has 4 channels available to be used, each being 8 bits wide.

These timers can be configured to work in 3 distinct modes: timer mode, pulse output mode and event counter mode.

In timer mode, the CPU will only be executing the interrupt handlers at a given period / frequency. It can also generate baud rates for the serial communication peripherals.

In pulse output mode, the CPU will simply output a PWM wave with a given frequency and duty cycle.

Finally, in event counter mode the timers can be cascaded to form a 16-bit timer out of two 8 bit timers. This is achieved when one timer overflows causing another to trigger/increment its counting register.

The specifications described by Renesas are:

ITEM	DESCRIPTION
Count clock	■ Frequency dividing clock: PCLK/1, PCLK/2, PCLK/8, PCLK/32, PCLK/64, PCLK/1024, PCLK/8192 ■ External clock
Number of channels	(8 bits × 2 channels) × 2 units
Compare match	■ 8-bit mode (compare match A, compare match B) ■ 16-bit mode (compare match A, compare match B)
Counter clear	Selected by compare match A or B, or an external reset signal.
Timer output	Output pulses with a desired duty cycle or PWM output
Cascading of two channels	■ 16-bit count mode 16-bit timer using TMR0 for the upper 8 bits and TMR1 for the lower 8 bits (TMR2 for the upper 8 bits and TMR3 for the lower 8 bits) ■ Compare match count mode TMR1 can be used to count TMR0 compare matches (TMR3 can be used to count TMR2 compare matches).
Interrupt sources	Compare match A, compare match B, and overflow
DTC activation	DTC can be activated by compare match A interrupts or compare match B interrupts.
Generation of trigger to start A/D converter conversion	Compare match A of TMR0 and TMR2*1
Capable of generating baud rate clock for SCI	Generates baud rate clock for SCI.*2
Low power consumption facilities	Each unit can be placed in a module-stop state.

Note 1. For details, see section 41, 12-Bit A/D Converter (S12ADa), and section 42, 10-Bit A/D Converter (ADb).
Note 2. For details, see section 35, Serial Communications Interface (SCIc, SCId).

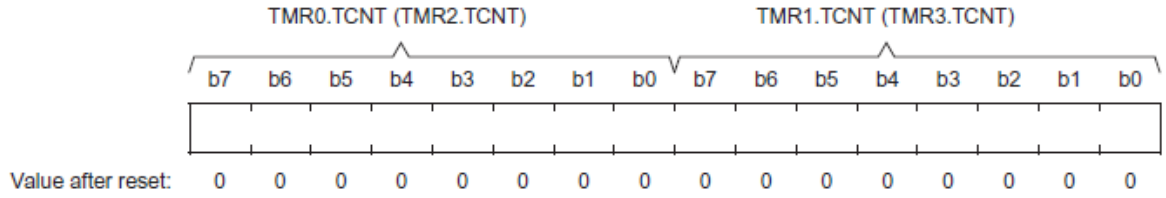
* Figure taken from RX63N Introduction book page 301 by Renesas

Figure 22 - Timer Specifications

The timers are by far the internal component with the most variety of settings to choose from.

For this reason, only the most important registers will be shown here.

The most basic timer register is the TCNT – Timer Counter.



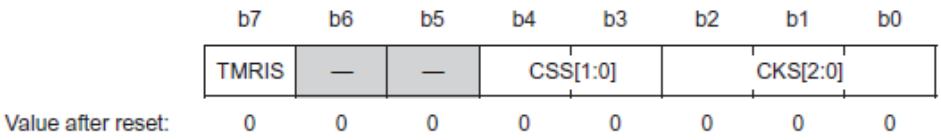
* Figure taken from RX63N Introduction book page 303 by Renesas

Figure 23 - Timers - TCNT Register

This register holds the current counting value of the timer, from value 0 to 255 decimal, and it increments/decrements its value at a given frequency.

Also, as can be seen on the previous figure, by using Timer 0's TCNT together with Timer 1's TCNT registers together, the timer mechanism is able to count up to 16 bits instead of 8.

Another critical register is the TCCR – Timer Counter Control Register.



* Figure taken from RX63N Introduction book page 305 by Renesas

Figure 24 - Timers - TCCR Register

This register is responsible for selecting the source of the clock (external/internal), the divided frequency generated from the original clock and the trigger edge of this clock (negative or positive edge).

BIT	SYMBOL	BIT NAME	DESCRIPTION	R/W
b2 to b0	CKS[2:0]	Clock Select*	See table below.	R/W
b4, b3	CSS[1:0]	Clock Source Select	See table below.	R/W
b6, b5	—	Reserved	These bits are always read as 0. The write value should always be 0.	R/W
b7	TMRIS	Timer Reset Detection Condition Select	0: Cleared at rising edge of the external reset	R/W
			1: Cleared when the external reset is high	
Note: * To use an external clock, set the Pn.PDR.Bi bit for the corresponding pin to "0" and the PORTn.PMR.Bi bit to "1". For details, see [1] section 21, I/O Ports.				

* Figure taken from RX63N Introduction book page 305 by Renesas

Figure 25 - Timers - TCCR Register Table 1

CHANNEL	TCCR REGISTER CSS[1:0] b4	b3	CKS[2:0] b2	b1	b0	DESCRIPTION
TMR0 (TMR2)	0	0	—	0	0	Clock input prohibited
					1	Uses external clock. Counts at rising edge*1.
				1	0	Uses external clock. Counts at falling edge*1.
					1	Uses external clock. Counts at both rising and falling edges*1.
	0	1	0	0	0	Uses frequency dividing clock. Counts at PCLK.
					1	Uses frequency dividing clock. Counts at PCLK/2.
				1	0	Uses frequency dividing clock. Counts at PCLK/8.
					1	Uses frequency dividing clock. Counts at PCLK/32.
			1	0	0	Uses frequency dividing clock. Counts at PCLK/64.
					1	Uses frequency dividing clock. Counts at PCLK/1024.
				1	0	Uses frequency dividing clock. Counts at PCLK/8192.
					1	Clock input prohibited
	1	0	—	—	—	Setting prohibited
	1	1	—	—	—	Counts at TMR1.TCNT (TMR3.TCNT) overflow signal*2
TMR1 (TMR3)	0	0	—	0	0	Clock input prohibited
					1	Uses external clock. Counts at rising edge*1.
				1	0	Uses external clock. Counts at falling edge*1.
					1	Uses external clock. Counts at both rising and falling edges*1.
	0	1	0	0	0	Uses frequency dividing clock. Counts at PCLK.
					1	Uses frequency dividing clock. Counts at PCLK/2.
				1	0	Uses frequency dividing clock. Counts at PCLK/8.
					1	Uses frequency dividing clock. Counts at PCLK/32.
			1	0	0	Uses frequency dividing clock. Counts at PCLK/64.
					1	Uses frequency dividing clock. Counts at PCLK/1024.
				1	0	Uses frequency dividing clock. Counts at PCLK/8192.
					1	Clock input prohibited
	1	0	—	—	—	Setting prohibited
	1	1	—	—	—	Counts at TMR0.TCNT (TMR2.TCNT) compare match A*2

Notes: 1. To use an external reset, set the PORTn.PDR.Bi bit for the corresponding pin to "0" and the PORTn.PMR.Bi bit to "1". For details, see [1] section 21, I/O Ports.
2. If the clock input of TMR0 (TMR2) is the overflow signal of the TMR1.TCNT (TMR3.TCNT) counter and that of TMR1 (TMR3) is the compare match signal of the TMR0.TCNT (TMR2.TCNT) counter, no incrementing clock is generated. Do not use this setting.

* Figure taken from RX63N Introduction book page 306 by Renesas

Figure 26 - Timers - TCCR Register Table 2

Other relevant registers:

1.1) TCORA/B – Time Constant Register A / B

1.2) TCR – Timer Control Register

1.3) TMDR – Timer Mode Register

1.4) TIORH/TIORL/TIOR – Timer IO Control Register

1.5) TGRx (x = A to D) – Timer General Register

1.6) TIER – Timer Interrupt Enable Register

1.7) TCSR – Timer Control Status Register

1.8) TSR – Timer Status Register

1.9) TSTR – Timer Start Register

1.10) TSYR – Timer Synchronous Register

1.11) NFCR – Noise Filter Control Register

Compare Match Registers:

2.1) CMSTR0/1 – Compare Match Timer Start Register 0 / 1

2.2) CMCR – Compare Match Timer Control Register

2.3) CMCNT – Compare Match Counter

2.4) CMCOR – Compare Match Constant Register

Real time Clock Registers:

3.1) R64CNT – 64 Hz Counter

3.2) RSECCNT – Second Counter

3.3) RMINCNT – Minute Counter

3.4) RHRCNT – Hour Counter

3.5) RWKCNT – Day-of-Week Counter

3.6) RDAYCNT – Date Counter

3.7) RMONCNT – Month Counter

3.8) RYRCNT – Year Counter

3.9) RCRn – RTC Control Register (n = 1 to 4)

3.10) RTCCRy – Time Capture Control Register y (y = 0 to 2)

For this assignment, the timers will be used for RTOS task switching, time management and high performance polling.

1.B.2.4) ADC CONVERTERS

Some of the ADC registers that can be used:

1) **ADDRn** (n = 0 to 20) – A/D Data Registers. These registers simply store the 16-bit value representation of the analogue signal, or in other words, the result of the ADC conversion.

b7	b6	b5	b4	b3	b2	b1	b0

Value after reset:

0	0	0	0	0	0	0	0

2) **ADCSR** – A/D Control Register

	b7	b6	b5	b4	b3	b2	b1	b0
	ADST	ADCS	—	ADIE	CKS[1:0]		TRGE	EXTRG
Value after reset:	0	0	0	0	0	0	0	0

* Figure taken from RX63N Introduction book page 168 by Renesas

Figure 27 - ADC - Control Register

3) **ADSANx** (x = 0 or 1) – A/D Channel Select Registers

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	ANS0[15:0]															
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

* Figure taken from RX63N Introduction book page 169 by Renesas

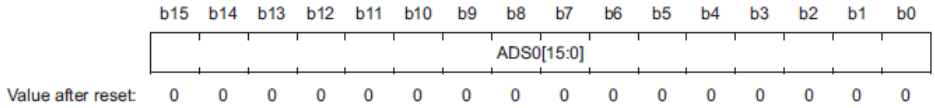
Figure 28 - ADC - Channel Select Registers (1)

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	—	—	—	—	—	—	—	—	—	—	—	ANS1[4:0]				
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

* Figure taken from RX63N Introduction book page 170 by Renesas

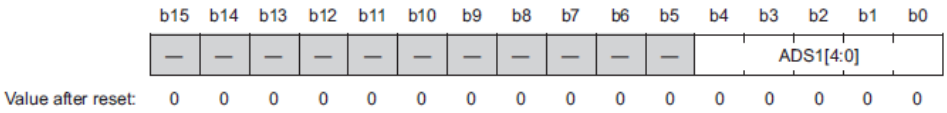
Figure 29 - ADC - Channel Select Registers (2)

4) ADADSx (x = 0 or 1) – A/D Converted Value Addition Mode Select Registers



* Figure taken from RX63N Introduction book page 170 by Renesas

Figure 30 - ADC - ADADSx Registers (1)



* Figure taken from RX63N Introduction book page 170 by Renesas

Figure 31 - ADC - ADADSx Registers (2)

5) ADADC – A/D Converted Value Addition Count Select Register

6) ADCER – A/D Control Extended Register

7) ADSTRGR – A/D Start Trigger Select Register

8) ADEXICR – A/D Converted Extended Input Control Register

9) ADTSDR – A/D Temperature Sensor Data Register

10) ADOCDR – A/D Internal Reference Voltage Data Register

Finally, it's important to note that the ADC converter is not being used for this project. This is because the Hall Effect sensor is a digital component and does not need to be ADC converted. GPIOs and Interrupts are perfectly capable of reading this sensor.

This subsection about ADCs was added purely for reference purposes and because the car might use ADC in the future.

1.B.2.5) SERIAL COMMUNICATION PERIPHERALS

Three serial communication mechanisms are being used on the RX63N microcontroller. The **UART** – Universal Asynchronous Receiver/Transmitter interface (renamed to SCI by Renesas), **I^2C** (or IIC/RIIC) – Inter Integrated Circuit and **SPI** – Serial Peripheral Interface (or RSPI).

	SERIAL COMMUNICATION INTERFACE	RENESAS SERIAL PERIPHERAL INTERFACE	RENESAS INTER INTEGRATED CHIP BUS
Modes	USART/Smart Card interface	Four wire/Three wire mode	I^2C mode or SM bus mode
Max transfer speed	1.92 Mbps with PCLK @ 50 MHz	Up to 25 Mbps	Up to 1 Mbps
Max. number of devices/slaves connected per bus	1	8 SPI integrated channels for slave selection	127
Number of wires required for communication (not including ground)	2 to 3	3 to 4	2
Max. length of wires [1], page 9.	40 meters @ 1 Mbps	10 feet with clock speed of few kHz	10 meters @ 100 kbps
Number of bits per unit transfer (excluding start and stop)	7 or 8	Up to 32 bits	8
Slave selection method	Addressed during communication when in multi-processor communication mode, or else not available.	Using slave select line	Through addressing

* Figure taken from RX63N Introduction book page 197 by Renesas

Figure 32 - Serial Comms - Specifications

1.B.2.5.1) UART

Some of the registers used by UART are:

1) SMR - Serial Mode Register, where the UART protocol parameters are configured

	b7	b6	b5	b4	b3	b2	b1	b0
	CM	CHR	PE	PM	STOP	MP	CKS[1:0]	
Value after reset:	0	0	0	0	0	0	0	0

Bit	Symbol	Bit Name	Description	R/W
b1, b0	CKS[1:0]	Clock Select	b1 b0 0 0: PCLK clock (n = 0)*1 0 1: PCLK/4 clock (n = 1)*1 1 0: PCLK/16 clock (n = 2)*1 1 1: PCLK/64 clock (n = 3)*1	R/W*4
b2	MP	Multi-Processor Mode	(Valid only in asynchronous mode) 0: Multi-processor communications function is disabled 1: Multi-processor communications function is enabled	R/W*4
b3	STOP	Stop Bit Length	(Valid only in asynchronous mode) 0: 1 stop bit 1: 2 stop bits	R/W*4
b4	PM	Parity Mode	(Valid only when the PE bit is 1 in asynchronous mode) 0: Selects even parity 1: Selects odd parity	R/W*4
b5	PE	Parity Enable	(Valid only in asynchronous mode) • When transmitting 0: Parity bit addition is not performed 1: The parity bit is added • When receiving 0: Parity bit checking is not performed 1: The parity bit is checked	R/W*4
b6	CHR	Character Length	(Valid only in asynchronous mode) 0: Selects 8 bits as the data length*2 1: Selects 7 bits as the data length*3	R/W*4
b7	CM	Communications Mode	0: Asynchronous mode 1: Clock synchronous mode	R/W*4

* Figure taken from RX63N User's Manual book page 1327 by Renesas

Figure 33 - Serial Comms - UART - SMR Register

2) SCR – Serial Control Register

	b7	b6	b5	b4	b3	b2	b1	b0
	TIE	RIE	TE	RE	MPIE	TEIE	CKE[1:0]	
Value after reset:	0	0	0	0	0	0	0	0

Bit	Symbol	Bit Name	Description	R/W
b1, b0	CKE[1:0]	Clock Enable	• For SCI0 to SCI4, SCI7 to SCI11 (Asynchronous mode) b1 b0 0 0: On-chip baud rate generator The SCKn pin functions as I/O port. 0 1: On-chip baud rate generator The clock with the same frequency as the bit rate is output from the SCKn pin. 1 x: External clock The clock with a frequency 16 times the bit rate should be input from the SCKn pin. Input a clock signal with a frequency eight times the bit rate when the SEMR.ABCS bit is 1. (Clock synchronous mode) b1 b0 0 x: Internal clock The SCKn pin functions as the clock output pin. 1 x: External clock The SCKn pin functions as the clock input pin.	R/W*1
b1, b0	CKE[1:0]	Clock Enable	• For SCI5, SCI6 and SCI12 (Asynchronous mode) b1 b0 0 0: On-chip baud rate generator The SCKn pin functions as I/O port. 0 1: On-chip baud rate generator The clock with the same frequency as the bit rate is output from the SCKn pin. 1 x: External clock or TMR clock • When an external clock is used, the clock with a frequency 16 times the bit rate should be input from the SCKn pin. Input a clock signal with a frequency eight times the bit rate when the SEMR.ABCS bit is 1. • The TMR clock can be used. (Clock synchronous mode) b1 b0 0 x: Internal clock The SCKn pin functions as the clock output pin. 1 x: External clock The SCKn pin functions as the clock input pin.	R/W*1
b2	TEIE	Transmit End Interrupt Enable	0: A TEI interrupt request is disabled 1: A TEI interrupt request is enabled	R/W

* Figure taken from RX63N User's Manual book page 1330 by Renesas

Figure 34 - Serial Comms - UART - SCR Register

Other relevant registers:

3) SSR – Serial Status Register

4) SCMR – Serial Control Mode Register

5) BRR – Bit Rate Register

6) SEMR – Serial Extended Mode Register

7) SNFR – Noise Filter Setting Register

1.B.2.5.2) I²C

In respect to the I²C protocol, there is a very wide range of registers that can be used:

1) **ICMRn** (n = 1 to 3) – I²C Mode Register <n>
2) **ICCRn** (n = 1 to 2) - I²C Bus Control Register <n>
3) **ICFER** - I²C Bus Function Enable Register
4) **ICSER** - I²C Bus Status Enable Register
5) **ICIER** - I²C Bus Interrupt Enable Register
6) **ICSRn** (n = 1 to 2) - I²C Bus Status Register <n>
7) **SARLy** (y = 0 to 2) – Slave Address Register Ly
8) **SARUy** (y = 0 to 2) – Slave Address Register Uy
9) **ICBRL/H** - I²C Bus Bit Rate Low/High-Level Register
10) **ICDRT** - I²C Bus Transmit Data Register
11) **ICDRS** - I²C Bus Shift Register
12) **TMOCNT** – Timeout Internal Counter

This is the only description provided for I²C on this report. The reason is due to the fact that I²C has too many registers and is not a fundamental component to the smart autonomous car system.

1.B.2.5.3) SPI

The last internal component of the microcontroller is the SPI interface, which is the fastest short-range communication protocol present on the RX63N.

For this protocol, only two registers will be described, since there are 20 separate registers for the RSPI alone.

1) SPCR – RSPI Control Register. Allows the configuration of a wide range of settings for the SPI protocol. There is also a similar register named SPCR 2, which has other settings to configure.

	b7	b6	b5	b4	b3	b2	b1	b0
	SPRIE	SPE	SPTIE	SPEIE	MSTR	MODFEN	TXMD	SPMS
Value after reset:	0	0	0	0	0	0	0	0

Bit	Symbol	Bit Name	Description	R/W
b0	SPMS	RSPI Mode Select	0: SPI operation (four-wire method) 1: Clock synchronous operation (three-wire method)	R/W
b1	TXMD	Communications Operating Mode Select	0: Full-duplex synchronous serial communications 1: Serial communications consisting of only transmit operations	R/W
b2	MODFEN	Mode Fault Error Detection Enable	0: Disables the detection of mode fault error 1: Enables the detection of mode fault error	R/W
b3	MSTR	RSPI Master/Slave Mode Select	0: Slave mode 1: Master mode	R/W
b4	SPEIE	RSPI Error Interrupt Enable	0: Disables the generation of RSPI error interrupt requests 1: Enables the generation of RSPI error interrupt requests	R/W
b5	SPTIE	RSPI Transmit Interrupt Enable	0: Disables the generation of RSPI transmit interrupt requests 1: Enables the generation of RSPI transmit interrupt requests	R/W
b6	SPE	RSPI Function Enable	0: Disables the RSPI function 1: Enables the RSPI function	R/W
b7	SPRIE	RSPI Receive Interrupt Enable	0: Disables the generation of RSPI receive interrupt requests 1: Enables the generation of RSPI receive interrupt requests	R/W

* Figure taken from RX63N User's Manual book page 1568 by Renesas

Figure 35 – Serial Comms – RSPI - SPCR

2) SPDR – RSPI Data Register.

	b31	b30	b29	b28	b27	b26	b25	b24	b23	b22	b21	b20	b19	b18	b17	b16
	SPD31	SPD30	SPD29	SPD28	SPD27	SPD26	SPD25	SPD24	SPD23	SPD22	SPD21	SPD20	SPD19	SPD18	SPD17	SPD16
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

	b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
	SPD15	SPD14	SPD13	SPD12	SPD11	SPD10	SPD9	SPD8	SPD7	SPD6	SPD5	SPD4	SPD3	SPD2	SPD1	SPD0
Value after reset:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

* Figure taken from RX63N User's Manual book page 1573 by Renesas

Figure 36 - Serial Comms – RSPI - SPDR

The RX63N's internal UART hardware is not being used for this project, due to pin shortage. Instead, UART is being generated out of the GPIO pins. There are very strong reasons for this. They are:

1- Pin and channel count
2- Software flexibility (any GPIO pin can be RX/TX pin)
3- The code can be easily ported to another platform, since the transmission is purely virtual (executed by the CPU using a C algorithm)

The I^2C protocol is being used for the accelerometer and temperature sensor.

Finally, the SPI protocol is being used on the LCD screen.

The hardware that is being used on the YRDKRX63N board that is external to the microcontroller is listed below.

1.B.3.1) SD CARD

Will be used for storing event log files.

Figure 37 – SD Card

1.B.3.2) LCD SCREEN

Will be used to display user messages.

Figure 38 - LCD Screen

1.B.3.3) SPEAKER

Will be used to play tunes and / or play status / alarm sounds for the robot car.

Figure 39 - Speaker

1.B.3.4) MICROPHONE

Could be used to control the car without a wireless communication link. Speech recognition algorithms could be implemented in order to process the incoming data from this sensor and use that to control the car instead.

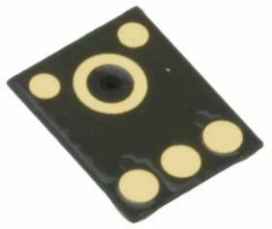


Figure 40 - Microphone

1.B.3.5) ACCELEROMETER (ADXL345)

Will be used to calculate the speed of the car and to determine if the car is being lifted off the ground by someone.

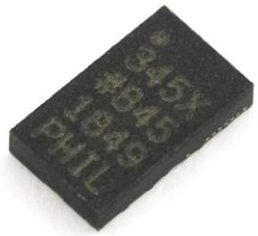


Figure 41 – Accelerometer

1.B.3.6) PUSH BUTTONS

Will be used to create simple user interaction together with the LCD screen.

Figure 42 - Push button

1.B.3.7) TEMPERATURE SENSOR (ADT75ARZ / ADT7410TRZ)

Will be used to monitor the type of environment the car is present on. For instance, in cold countries, if the temperature is below a threshold, then it would be easy to determine if the car is inside or outside a building.

More uses could be applied to this sensor with future developments.

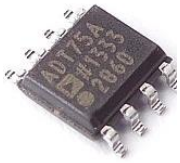


Figure 42 - Temperature Sensor 1

Figure 43 - Temperature Sensor 2

1.B.4.1) MISCELLANEOUS COMPONENTS

There are a few missing hardware components from the list that were explicitly added due to their basic nature:

1- Passive components, such as:
   a. Resistors (pull-ups and pull-downs)
   b. Capacitors
2- The PCB itself (fiberglass)
3- PCB headers
4- PCB connectors
5- Wires
6- Screws
7- Bolts
8- Spacers
9- Solder
10- Hot glue
11- PVC tape
12- Cable ties

1.C) SOFTWARE REQUIREMENTS

After listing the required hardware on the previous subtask, it is in this section where the software for the previous hardware will be described. More specifically, the software requirements for the internal components of the RX63N, such as the GPIOs, Interrupts, Timers, UART, I2C and SPI.

This description will follow a specific format. Each component will be explained with some initialization code and a very simple usage of this same component.

1.C.1) GPIOS

1.C.1.1) INITIALIZATION CODE

```
1   void init_gpios(void)
2   {
3         PORTD.PDR.BIT.B0  = 1; // Set pin PD0 as output (connected to an LED)
4         PORTD.PODR.BIT.B0 = 0; // Output binary 0 out of this pin
5         PORT4.PDR.BIT.B1  = 0; // Set pin P41 as input  (connected to a push button)
6   }
```

1.C.1.2) SAMPLE CODE

```
1   void main(void)
2   {
3         // Initialize two GPIO pins
4         init_gpios();
5
6         // Super loop
7         while(true) {
8               // Read button
9               unsigned char push_button = PORT4.PIDR.BIT.B1;
10
11              // If button was pressed (the act of pressing pulls the pin's voltage down)
12              if(push_button == false)
13                    PORTD.PODR.BIT.B0 = 1; // Output binary 1 out of this pin
14              else // If not pressed
15                    PORTD.PODR.BIT.B0 = 0; // Output binary 0 out of this pin
16        }
17  }
```

1.C.2) INTERRUPTS

1.C.2.1) INITIALIZATION CODE

```
1   void init_interrupts(void)
2   {
3          // Set the output LED pin:
4          PORTD.PDR.BIT.B0       = 1;    // Set pin PD0 as output (connected to an LED)
5          PORTD.PODR.BIT.B0      = 0;    // Output binary 0 out of this pin
6
7          // Set the switch pin to trigger the interrupt # 9:
8          PORT4.PDR.BIT.B1       = 0;    // Set this pin as input (connected to a switch)
9          MPC.P41PFS.BYTE        = 0x40; // Set the pin to Interrupt mode (as opposed to GPIO/Analog mode)
10         ICU.IRQCR[9].BIT.IRQMD = 0x01; // Set the interrupt to trigger on falling edge
11         IPR(ICU, IRQ9)         = 0x03; // Set the interrupt priority to 3
12         IR (ICU, IRQ9)         = 0;    // Clear pending interrupts
13         IEN(ICU, IRQ9)         = 1;    // And finally, enable the interrupt
14  }
```

1.C.2.2) SAMPLE CODE

```
1   bool led_toggle = false;
2
3   // Declare this interrupt handler above the init_interrupts function
4
5   #pragma interrupt(sw2_isr (vect = VECT(ICU, IRQ9)))
6   void sw2_isr(void)
7   {
8          if(led_toggle == false)
9                 led_toggle = true;
10         else
11                led_toggle = false;
12  }
13
14  void main(void)
15  {
16         init_interrupts();
17         while(true) {
18                // Cast the led_toggle value into unsigned char
19                unsigned char toggle_value = (unsigned char)led_toggle;
20                // Output whatever the toggle value is out through the LED pin
21                PORTD.PODR.BIT.B0 = toggle_value;
22         }
23  }
```

1.C.3) TIMERS

1.C.3.1) INITIALIZATION CODE

```
1   void init_timer(void) {
2           // Set Timer 0 as a PWM wave generator
3           MSTP(TMR0)         = 0; // Enable Timer 0 unit
4           TMR0.TCCR.BIT.CSS  = 1; // Use the internal clock
5           TMR0.TCCR.BIT.CKS  = 0; // Uses the original clock frequency without division
6           TMR0.TCR.BIT.CCLR  = 1; // This timer counts up until its counter value matches the register TCORA
7           TMR0.TCSR.BIT.OSA  = 2; // Outputs 5v (high) out of the TMR0 pin whenever the timer matches TCORA
8           TMR0.TCSR.BIT.OSB  = 1; // Outputs 0v (low)  out of the TMR0 pin whenever the timer matches TCORB
9           TMR0.TCORA         = 0; // Set the PWM frequency (0 by default)
10          TMR0.TCORB         = 0; // Set the Duty cycle    (0 by default)
11  }
```

1.C.3.2) SAMPLE CODE

```
1   void set_timer_pwm(unsigned char pwm_frequency, unsigned char duty_cycle)
2   {
3           TMR0.TCORA = 255 - pwm_frequency;
4           TMR0.TCORB = 255 - duty_cycle;
5           // Inverted the value because the higher the
6           // counter value is the slower the trigger occurs
7   }
8
9   void main(void)
10  {
11          unsigned char frequency  = 255;
12          unsigned char duty_cycle = 0;
13
14          // Initialize the timer
15          init_timer();
16
17          // Set an initial frequency and duty cycle (the values range from 0 to 255)
18          set_timer_pwm(frequency, 0);
19
20          while(true)
21          {
22                  // Delay for a while
23                  for(unsigned int i = 0; i < 0x1000; i++);
24
25                  // Set new Duty Cycle
26                  set_timer_pwm(frequency, duty_cycle);
27
28                  // Increment it
29                  duty_cycle++;
30
31                  // Check if the duty cycle is out of bounds
32                  if(duty_cycle >= frequency)
33                          duty_cycle = 0;
34          }
35  }
```

1.C.4) UART

1.C.4.1) INITIALIZATION CODE

```
1   void init_uart(void)
2   {
3           // Activate SCI channel 2
4           SYSTEM.MSTPCRA.BIT.ACSE = 0;
5           MSTP(SCI2)              = 0;
6
7           // Configure the pins TXD2 and RXD2
8           SCI2.SCR.BYTE    = 0x00; // Clear the entire SCR register
9           MPC.P50PFS.BYTE  = 0x4A; // Configure pin to use the UART peripheral instead of GPIO/SPI/I2C
10          MPC.P52PFS.BYTE  = 0x4A; // Configure pin to use the UART peripheral instead of GPIO/SPI/I2C
11          PORT5.PDR.BIT.B0 = 1;    // Set P50 as output (TXD2)
12          PORT5.PDR.BIT.B2 = 0;    // Set P52 as input  (RXD2)
13          PORT5.PMR.BIT.B0 = 1;    // Set pin to peripheral mode
14          PORT5.PMR.BIT.B2 = 1;    // Set pin to peripheral mode
15
16          // Configure SCI channel 2
17          SCI2.SMR.BYTE      = 0x00; // Configure parity, data width, stop bits, etc
18          SCI2.SCMR.BIT.SMIF = 0;    // Disable smart card mode (already disabled by default but it's good practice to reset anyways)
19          SCI2.BRR           = 12;   // Set the baud rate value
20          SCI2.SCR.BIT.RIE   = 1;    // Enable the reception (RX) interrupt
21          SCI2.SCR.BIT.TIE   = 1;    // Enable the transmission (TX) interrupt
22          SCI2.SCR.BIT.TEIE  = 1;    // Enable transmission (TX) interrupt which triggers whenever a full byte is sent
23          IR(SCI2, RXI2)     = 0;    // Enable RX interrupt (on the ICU and not on the SCI unit)
24          IR(SCI2, TXI2)     = 0;    // Enable TX interrupt (on the ICU and not on the SCI unit)
25          SCI2.SCR.BYTE     |= 0x30; // Enable both RX and TX and start receiving / transmitting
26  }
27
```

1.C.4.2) SAMPLE CODE

```
1   void sci2_transmit(unsigned char byte)
2   {
3         // Wait for the current data to be transmitted before re-transmitting another byte
4         while(IEN(SCI2, TXI2) == 0); // Be careful. This blocks the main thread
5         IEN(SCI2, TXI2) = 0;         // Clear the TX interrupt bit
6         SCI2.TDR = byte;             // Set the TX data register to the byte we want to transmit
7   }
8
9   unsigned char sci2_receive(void)
10  {
11        unsigned char byte = 0xFF;   // Set the byte to a default invalid value
12        while(IEN(SCI2, RXI2) != 0); // Wait for the RX buffer to be full
13        byte = SCI2.RDR;             // Fetch the received byte from the RX data register into memory
14        IEN(SCI2, RXI2) = 1;         // Re-enable RX interrupts
15        return byte;                 // Return received byte
16  }
17
18  void main(void)
19  {
20        // Initialize UART
21        init_uart();
22
23        while(true) {
24              // Wait and receive any incoming byte
25              unsigned char byte = sci2_receive();
26              // Echo back the received byte
27              sci2_transmit(byte);
28        }
29  }
```

1.C.5) I²C

1.C.5.1) INITIALIZATION CODE

```
1   void init_i2c(unsigned char this_devices_address)
2   {
3          SYSTEM.MSTPCRB.BIT.MSTPB21 = 0; // Enable the I2C / RIIC unit on the MSTP
4          RIIC0.ICCR1.BIT.ICE        = 0; // Do not drive the SCL and SDA wires for now
5          RIIC0.ICCR1.BIT.IICRST     = 1; // Reset the internal states and controls of the RIIC0 module
6          RIIC0.ICCR1.BIT.IICRST          = 0; // This flag needs to be cleared 'manually' in order for it to start working again after resetting
7
8          RIIC0.SARU0.BIT.FS    = 0;  // Set device address to 7 bits of width
9          RIIC0.SARL0.BYTE       = this_devices_address; // Set the address of the current device, which would probably be the master
10         RIIC0.ICMR1.BIT.CKS   = 7;  // Select the internal clock source PCLK divided by 128
11         RIIC0.ICBRH.BIT.BRH   = 28; // Set the bit counter for the high level of the serial clock cycle
12         RIIC0.ICBRL.BIT.BRL   = 29; // Set the bit counter for the low level of the serial clock cycle
13
14         RIIC0.ICMR3.BIT.ACKWP = 1;  // Disable the Write Protection for the ACK bit
15
16         RIIC0.ICIER.BIT.RIE   = 1;  // Enable the RX interrupt for when the buffer is full
17         RIIC0.ICIER.BIT.TIE   = 1;  // Enable the TX interrupt for when the buffer is full
18         RIIC0.ICIER.BIT.TEIE  = 0;  // Enable the TX interrupt for when the data has finished transmitting
19
20         RIIC0.ICIER.BIT.NAKIE = 1;  // Enable  the 'NACK  bit received' interrupt
21         RIIC0.ICIER.BIT.SPIE  = 1;  // Enable  the 'STOP  bit received' interrupt
22         RIIC0.ICIER.BIT.STIE  = 0;  // Disable the 'START bit received' interrupt
23
24         RIIC0.ICIER.BIT.ALIE  = 0;  // Disable the 'Arbitration lost' interrupt
25         RIIC0.ICIER.BIT.TMOIE = 0;  // Disable the 'Timeout interrupt' request
26
27         PORT1.PMR.BIT.B3    = 1;    // Set port P13 as a peripheral pin
28         PORT1.PMR.BIT.B2    = 1;    // Set port P12 as a peripheral pin
28         RIIC0.ICCR1.BIT.ICE = 1;    // Start up the I2C protocol
30  }
31
32  void uninit_i2c(void)
34  {
35         // Just disable the RIIC0 module on the MSTP
36         SYSTEM.MSTPCRB.BIT.MSTPB21 = 1;
37  }
```

1.C.5.2) SAMPLE CODE

```
1   unsigned char i2c_send_start(void)
2   {
3           // First see if the RIIC0 module is enabled/working
4           if(RIIC0.ICCR1.BIT.ICE) {
5                   // If it's enabled, wait until the bus is available
6                   while(RIIC0.ICCR2.BIT.BBSY);
7
8                   // Once it's available, 'lock' the bus by requesting a transmission using a START condition
9                   RIIC0.ICCR2.BIT.ST = 1;
10
11                  // Wait until the START condition is detected
12                  while(!(RIIC0.ICCR2.BIT.BBSY && RIIC0.ICSR2.BIT.START));
13
14                  // Disable the interrupt bit
15                  RIIC0.ICSR2.BIT.START = 0;
16
17                  // Return success code
18                  return 1;
19          } else {
20                  // Else, if it's not working, just return 0 which means an error code
21                  return 0;
22          }
23  }
24
25  unsigned char i2c_send_stop(void)
26  {
27          // First see if the RIIC0 module is enabled/working
28          if(RIIC0.ICCR1.BIT.ICE) {
29                  // If it's enabled, wait until the bus is available
30                  while(RIIC0.ICCR2.BIT.BBSY)
31                          RIIC0.ICCR2.BIT.SP = 1; // Keep setting the STOP condition bit while the bus is busy
32
33                  // Return success code
34                  return 1;
35          } else {
36                  // Else, if it's not working, just return 0 which means an error code
37                  return 0;
38          }
39  }
40
41  unsigned char i2c_write_byte_to_register(unsigned char slave_address, unsigned register_address, unsigned char byte)
42  {
43          // Transmit and wait the slave address
44          RIIC0.ICDRT = slave_address & 0xFE;
45          while(!RIIC0.ICSR2.BIT.TDRE);
46
47          // Transmit and wait the slave register's address
48          RIIC0.ICDRT = register_address;
49          while(!RIIC0.ICSR2.BIT.TEND) {
50                  // If we detect a NACK condition, bail out
51                  if(RIIC0.ICSR2.BIT.NACKF) {
52                          // Oops, we've received a NACK condition. Transmission must be aborted
53                          RIIC0.ICSR2.BIT.NACKF = 0;
54                          return 0;
55                  }
56          }
57
58          // Wait until the register address data has been fully transmitted
59          while(!RIIC0.ICSR2.BIT.TDRE);
60
61          // Now write the byte into the TX buffer
62          RIIC0.ICDRT = byte;
63          while(!RIIC0.ICSR2.BIT.TEND) {
64                  // If we detect a NACK condition, bail out
65                  if(RIIC0.ICSR2.BIT.NACKF) {
66                          // Oops, we've received a NACK condition. Transmission must be aborted
67                          RIIC0.ICSR2.BIT.NACKF = 0;
```

```
68                          return 0;
69                  }
70          }
71
72          // And wait for the byte to be transmitted
73          while(RIIC0.ICSR2.BIT.TDRE);
74
75          // Now write the byte into the TX buffer
76          while(!RIIC0.ICSR2.BIT.TEND) {
77                  // If we detect a NACK condition, bail out
78                  if(RIIC0.ICSR2.BIT.NACKF) {
79                          // Oops, we've received a NACK condition. Transmission must be aborted
80                          RIIC0.ICSR2.BIT.NACKF = 0;
81                          return 0;
82                  }
83          }
84
85          return 1;
86  }
87
88  unsigned char i2c_write_byte(unsigned char slave_address, unsigned char byte)
89  {
90          // Transmit and wait the slave address
91          RIIC0.ICDRT = slave_address & (0xFE);
92          while(!RIIC0.ICSR2.BIT.TDRE);
93
94          // Write the byte into the TX buffer
95          RIIC0.ICDRT = byte;
96          while(!RIIC0.ICSR2.BIT.TEND) {
97                  // If we detect a NACK condition, bail out
98                  if(RIIC0.ICSR2.BIT.NACKF) {
99                          // Oops, we've received a NACK condition. Transmission must be aborted
100                         RIIC0.ICSR2.BIT.NACKF = 0;
101                         return 0;
102                 }
103         }
104
105         // And wait for the byte to be transmitted
106         while(!RIIC0.ICSR2.BIT.TDRE);
107
108         while(!RIIC0.ICSR2.BIT.TEND) {
109                 // If we detect a NACK condition, bail out
110                 if(RIIC0.ICSR2.BIT.NACKF) {
111                         // Oops, we've received a NACK condition. Transmission must be aborted
112                         RIIC0.ICSR2.BIT.NACKF = 0;
113                         return 0;
114                 }
115         }
116         return 1;
117 }
118
119 unsigned char i2c_read_byte(unsigned char slave_address, unsigned char register_address)
120 {
121         unsigned char byte = 0xFF; // Set invalid default value
122         i2c_write_byte(slave_address, register_address); // Send to the slave address the register from which we want to
123 read from
124         i2c_send_stop(); // Request stop condition
125         RIIC0.ICSR2.BIT.STOP = 0; // Reset the stop flag
126
127         i2c_send_start(); // Request start condition
128         while(!RIIC0.ICSR2.BIT.TDRE); // And wait for the TX buffer to empty out
129         RIIC0.ICDRT = slave_address | (0x01); // Send the slave address in order to select the device
130         while(!RIIC0.ICSR2.BIT.RDRF); // Wait until the slave device responds with data, thus, making the buffer full
131
132         // If there was no NACK condition, then:
133         if(RIIC0.ICSR2.BIT.NACKF == 0) {
134                 RIIC0.ICMR3.BIT.WAIT  = 1; // Set the wait bit to 1
135                 RIIC0.ICMR3.BIT.ACKBT = 1; // Set ack bit to 1
136                 byte = RIIC0.ICDRR; // Read dummy byte
137                 while(!RIIC0.ICSR2.BIT.RDRF); // Wait again until the slave responds with data
138                 RIIC0.ICSR2.BIT.STOP = 0; // Reset the STOP bit
```

```
139                     RIIC0.ICCR2.BIT.SP = 1; // And request for the STOP condition to be generated
140                     byte = RIIC0.ICDRR; // Now we can finally fetch the received data from the slave device
141                     while(!RIIC0.ICSR2.BIT.STOP); // And wait until the stop condition has been transmitted
142                     return byte; // Return the byte to the caller function
143             }
144             else {
145                     // Else, just return whatever the default byte value was (an invalid value)
146                     return byte;
147             }
148     }
149
150     #define MASTER_ADDR                 0x10
151     #define ACCELEROMETER_SLAVE_ADDR    0x3A
152     #define ACCELEROMETER_REGISTER_ADDR 0x2D
153
154     void main(void)
155     {
156             // Initialize I2C protocol:
157             init_i2c(MASTER_ADDR);
158
159             // Write byte 0 to register 0x3A of the accelerometer:
160             i2c_send_start();
161             i2c_write_byte_to_register(ACCELEROMETER_SLAVE_ADDR, ACCELEROMETER_REGISTER_ADDR, 0x00);
162             i2c_send_stop();
163
164             // Read register # 0 of the accelerometer:
165             i2c_send_start();
166             unsigned char data = i2c_read_byte(ACCELEROMETER_SLAVE_ADDR, 0x00);
167             i2c_send_stop();
168
169             // Output this message out of the Segger J-Link debugger console
170             printf("\nReceived data: %d", data);
171
172             uninit_i2c();
173
174             while(true);
175     }
```

1.C.6) SPI

1.C.6.1) INITIALIZATION CODE

```
1   void init_spi(void)
2   {
3          // Enable the SPI unit on the MSTP
4          MSTP(RSPI0)       = 0;
5          RSPI0.SPPCR.BYTE  = 0x00;   // Initialize the SPI Pin Control Register
6          RSPI0.SPBR        = 0;      // Set the bit rate to the maximum value possible
7          // Note: if RSPI0.SPBR = 0, then bit rate = 25Mbps assuming PCLK = 50MHz
8          RSPI0.SPDCR.BYTE  = 0x20;   // 16-Bit data width, 1 frame, 1 chip select
9          RSPI0.SPCR2.BYTE  = 0x00;   // Disable idle interrupt
10         RSPI0.SPCMD0.WORD = 0x0400; // Send data's MSB first, keep SSL low
11         IPR(RSPI0, SPRI0) = 3;      // Set shared IPL for RSPIO
12         IEN(RSPI0, SPRI0) = 0;      // Disable 'RX buffer full' interrupt
13         IEN(RSPI0, SPTI0) = 0;      // Disable 'TX Buffer full' interrupts
14         IR (RSPI0, SPRI0) = 0;      // Clear pending RX buffer full interrupts
15         MPC.PC5PFS.BYTE   = 0x0D;   // Configure pin PC5 to use the RSPI peripheral (SCK)
16         MPC.PC6PFS.BYTE   = 0x0D;   // Configure pin PC6 to use the RSPI peripheral (MOSI)
17         MPC.PC7PFS.BYTE   = 0x0D;   // Configure pin PC7 to use the RSPI peripheral (MISO)
18         PORTC.PMR.BYTE    = 0xE0;   // Set all 3 pins to peripheral mode
19         PORTC.PODR.BYTE   = 0x17;   // Output binary 1 to pins PC0, PC1, PC2 and PC4
20         PORTC.PDR.BYTE    = 0x7F;   // Set all Port C pins (except PC7, which is MISO, since it's an input pin) to output mode
21         RSPI0.SPCR.BYTE   = 0xE9;   // Enable RSPI
22         RSPI0.SSLP.BYTE   = 0x00;   // Set SSLs to active high
23  }
```

1.C.6.2) SAMPLE CODE

```
1   void spi_send_32bits(signed short low_word, signed short high_word)
2   {
3         PORTC.PODR.BIT.B4 = 0; // Pull the PC4 pin down (which is connected to the chip select of the SD-Card)
4         while(RSPI0.SPSR.BIT.IDLNF); // Wait until SPI becomes available
5         RSPI0.SPDR.WORD.L = low_word;  // Set the lower word of the register
6         RSPI0.SPDR.WORD.H = high_word; // And the higher word of the register. The SPI protocol will transmit these two word values
7         while(RSPI0.SPSR.BIT.IDLNF); // Wait until sends the data and is available again
8         (void)RSPI0.SPDR.WORD.L; // Read the lower word of the SPDR
9         (void)RSPI0.SPDR.WORD.H; // Read the higher word of the SPDR. This is only done so that the interrupt flag can be cleared
10        PORTC.PODR.BIT.B4 = 1; // Terminate transmission by pulling the chip select pin high again
11  }
12
13  void main(void)
14  {
15        // Initialize SPI protocol
16        init_spi();
17
18        while(true) {
19              // Keep sending the 32-bit value 0x00000055 to the SD Card module
20              spi_send_32bits(0x0000, 0x0055);
21        }
22  }
```

1.C.7) SOFTWARE TOOLS

In the context of embedded systems and project development, it is known that a company project is usually composed of many different stages.

The design of the product / hardware is one of those project stages, including the software that is executed by the microcontroller.

The only way of performing this hardware design is by using software **tools**, such as ECAD and IDEs, provided by companies such as Renesas and also by the University.

The software tools being used for the development of the Smart Autonomous Bot are:

1. **e²studio** – This tool allows the student to write code for the microcontroller board RX63N. This type of software is called an IDE – Integrated Development Environment.

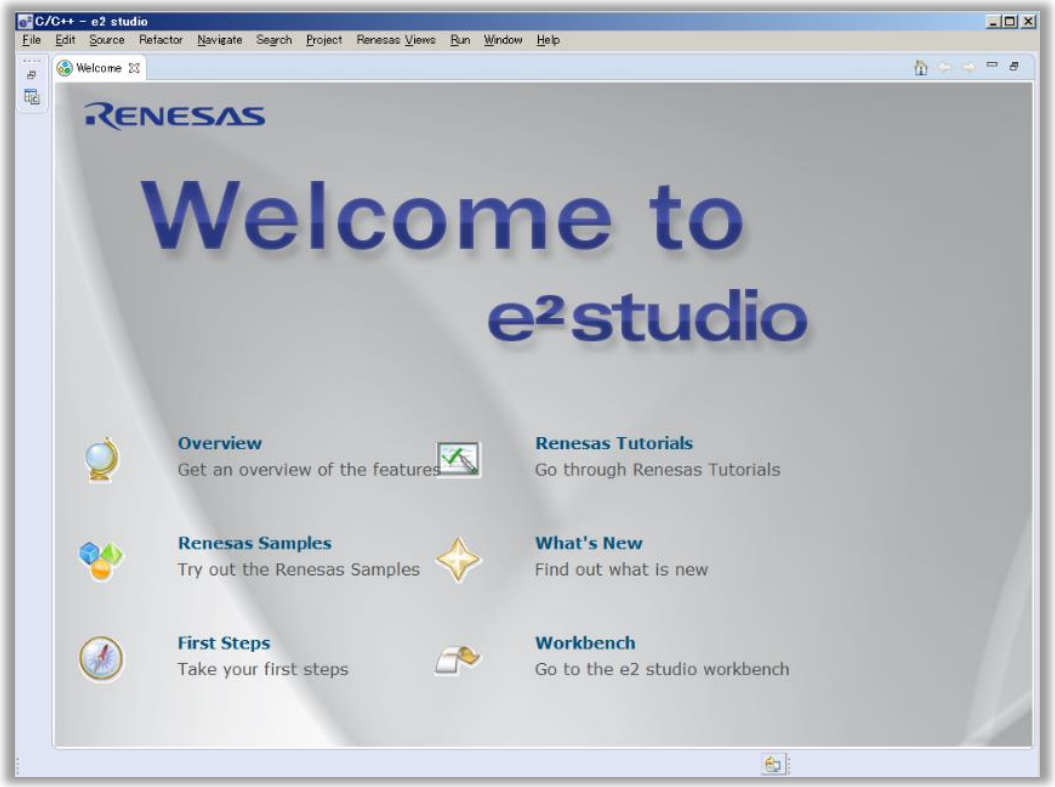


Figure 43 – e2studio welcome screen

2. **Proteus** – Allows the design of the schematic and PCB layout of the robot car. This software is often categorized as an ECAD (Electronic and Electrical Computer Aided Design) software tool rather than an IDE.

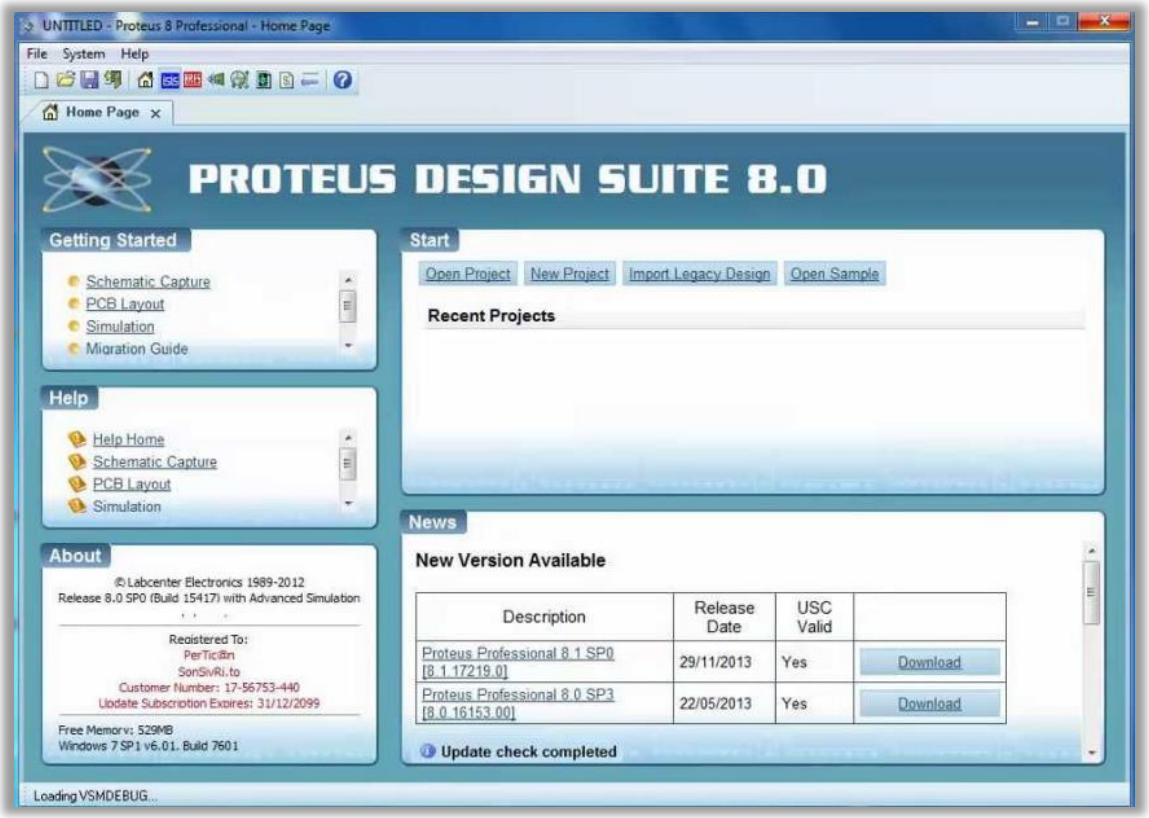


Figure 44 - Proteus welcome screen

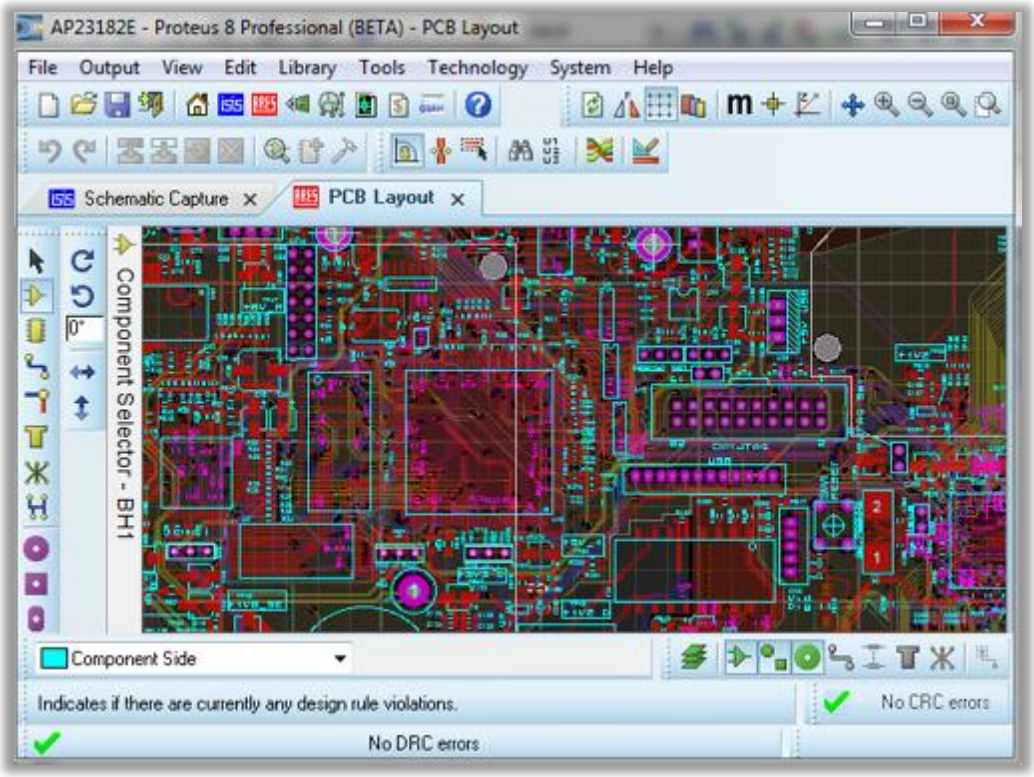


Figure 45 - Proteus Ares example screen

- **NOTE:** This image is for representation purposes only. It does not reflect the work being done on the smart car.

In respect to software requirements, the only topic that still needs to be covered is the actual C driver + application code that will be written on the microcontroller itself which will make up the actual Smart Car Algorithm.

This will be described in great detail on the Tasks 2.C, 3.B and 6 on this report.

TASK 2 – DESIGN

2.A) LAYOUT DIAGRAM + PCB

At this stage, the schematic has been successfully designed and laid out on the Proteus ECAD tool.

The full design is shown on the images below:

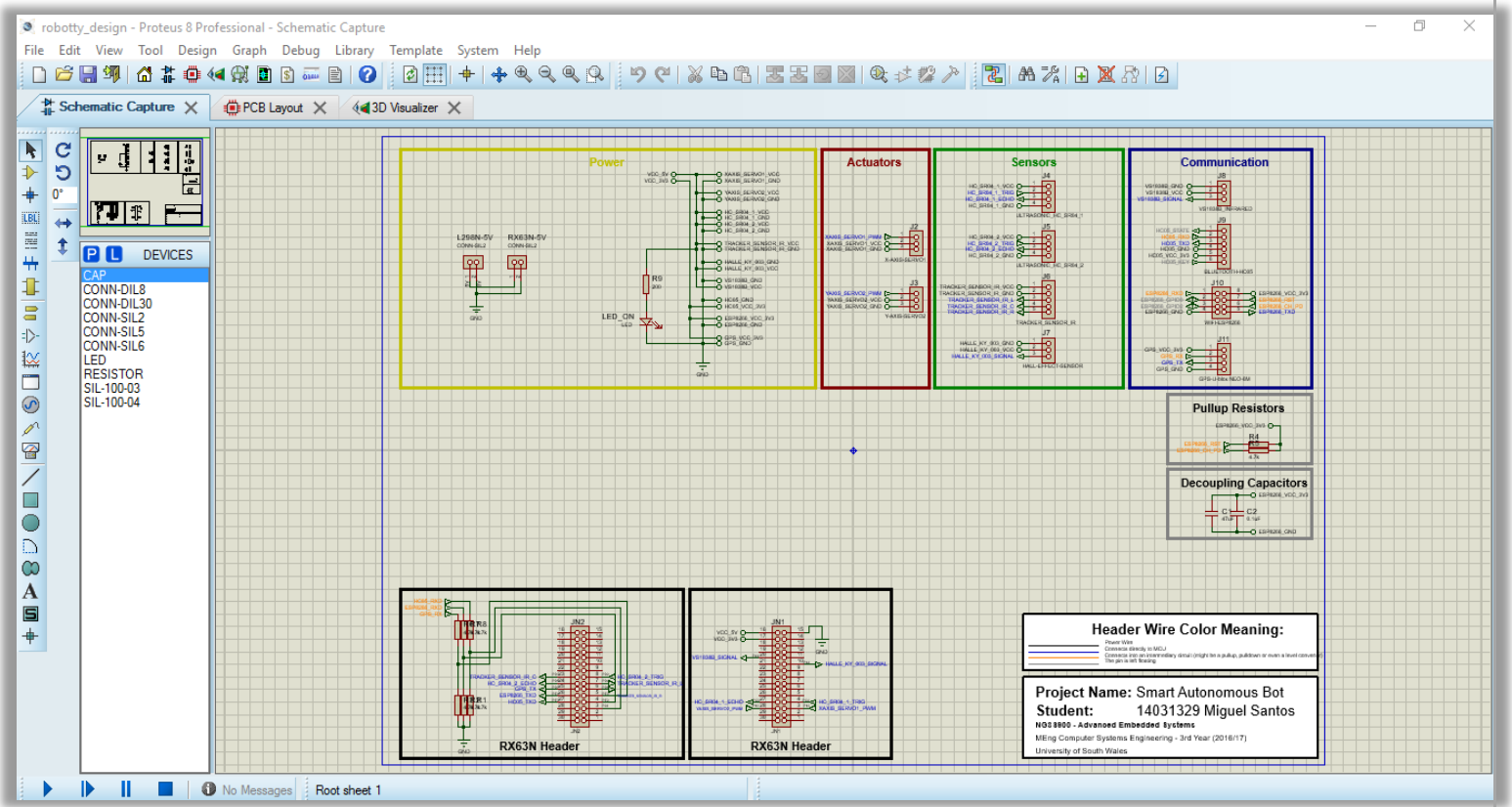


Figure 46 - Complete Proteus Circuit Schematic Capture

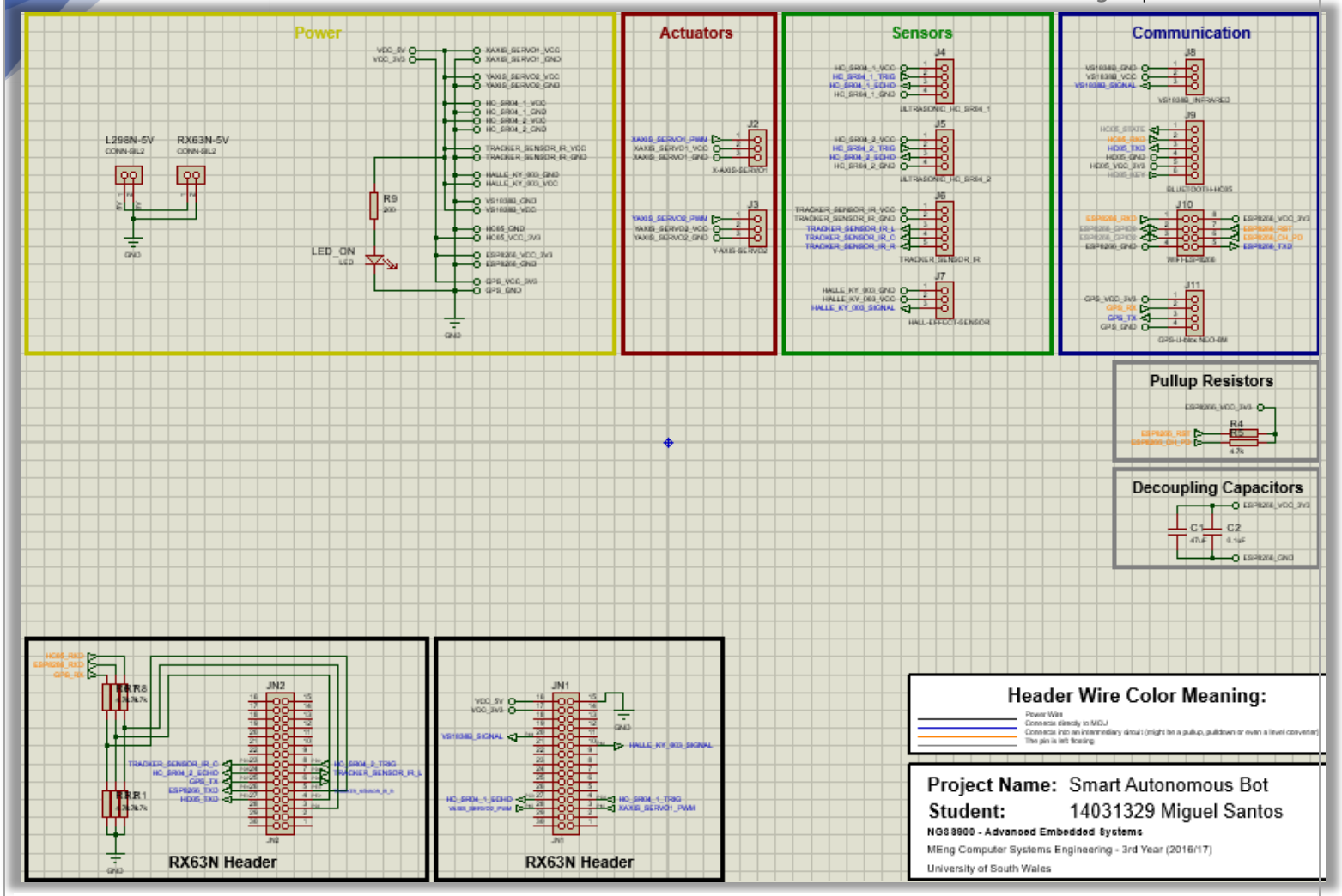


Figure 47 - Smart Bot Circuit Schematic

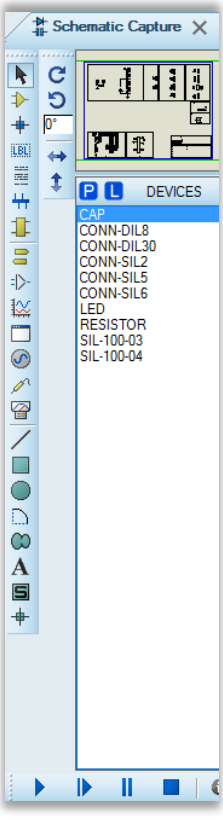


Figure 48 - Proteus Schematic Components

As can be seen, the schematic was designed in a very simplified way.

In the power 'block', two headers can be identified. These are the headers where the 5v output of the two H-Bridges will be connected to. The second header will connect into the RX63N's power header. The rest of the components will therefore use the 5V and 3.3V output voltages provided by the RX63N JN1 header on pins 1, 2 and 3.

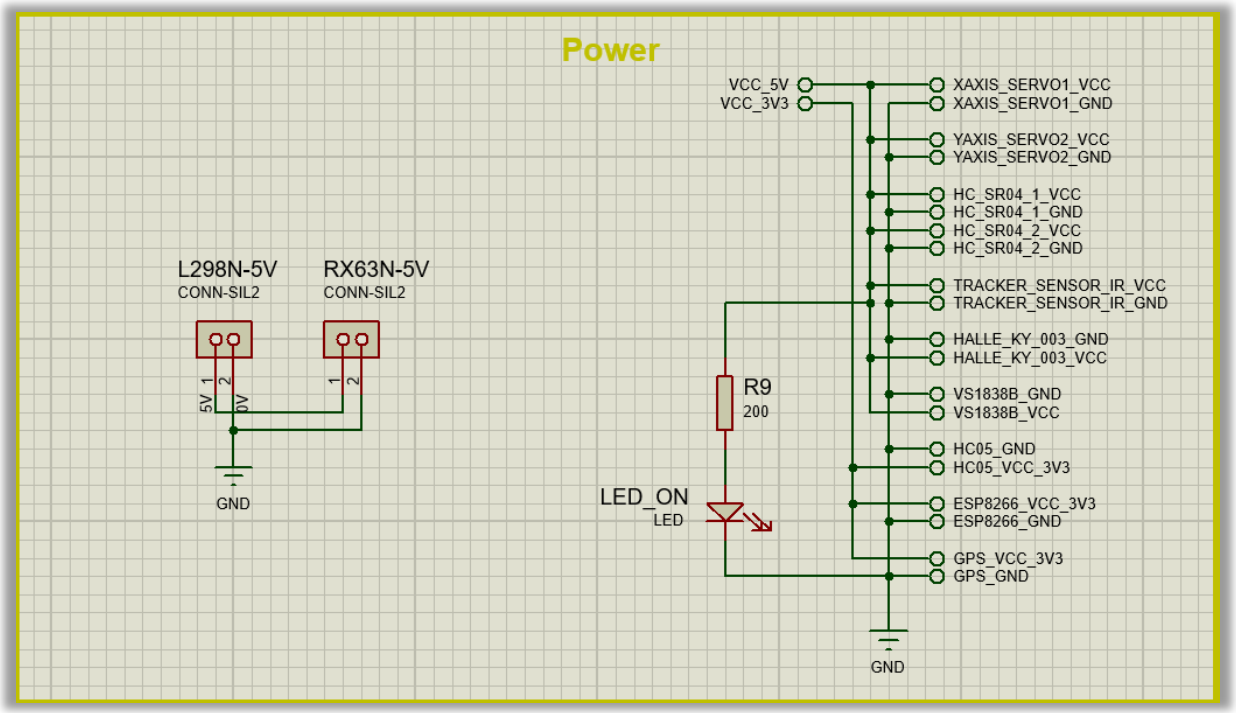


Figure 49 - Schematic - Power block

The following 3 images show the main components of the smart bot. The servos, the sensors and the communication modules are connected in these blocks.

Also note, the H-Bridges' inputs were **not** added to the schematic because the 4 inputs connect directly into the pins 1, 2, 3 and 4 of the header PMOD1 on the RX63N as this simplifies and improves the design.

This consideration will be reminded on the next section while showing the PCB and Bot-Chassis.

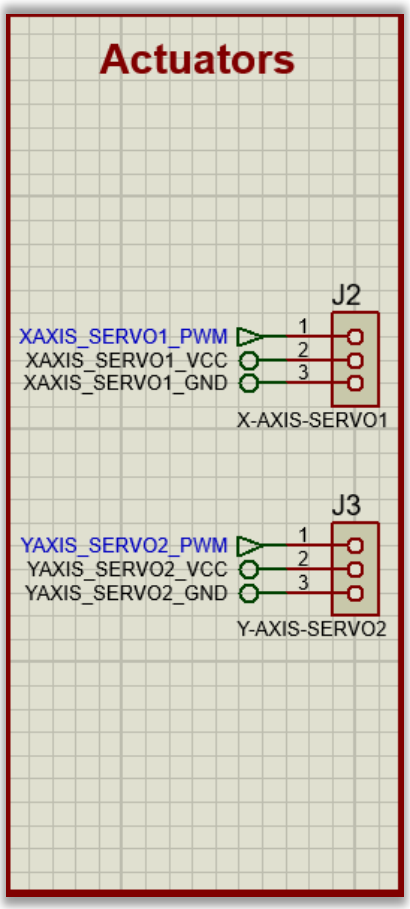


Figure 50 - Schematic - Actuators block

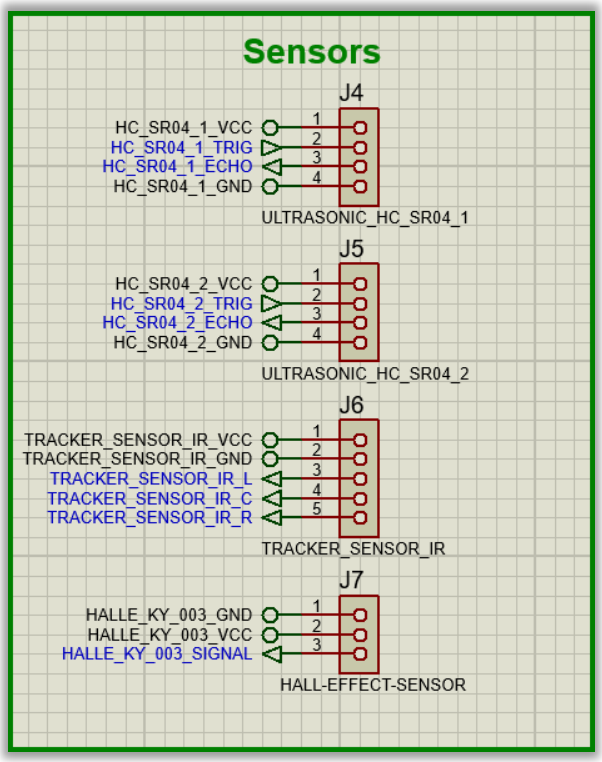


Figure 51 - Schematic - Sensors block

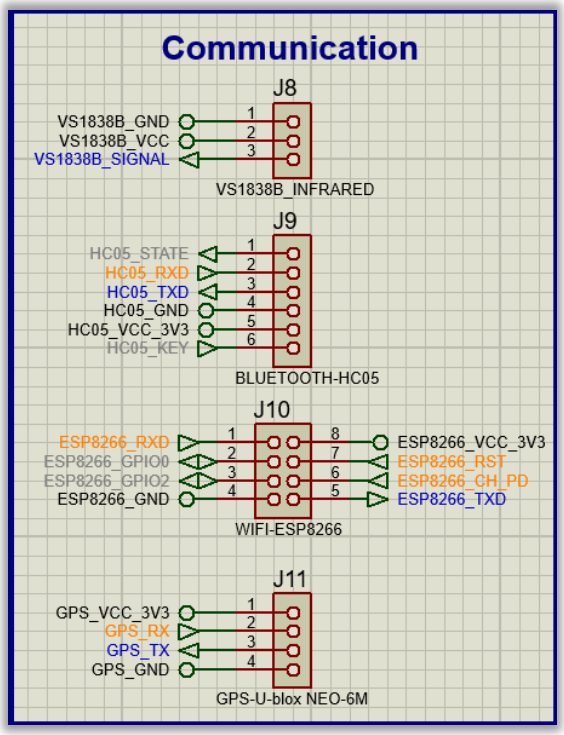


Figure 52 - Schematic - Communication block

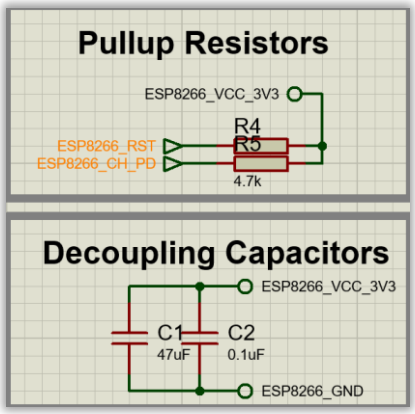


Figure 55 - Schematic - Misc block

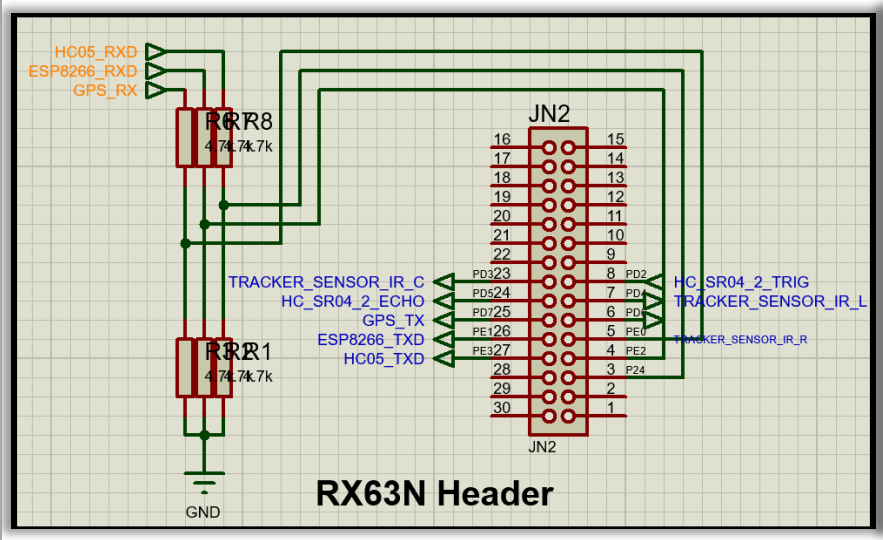


Figure 54 - Schematic - RX63N JN2 Header block

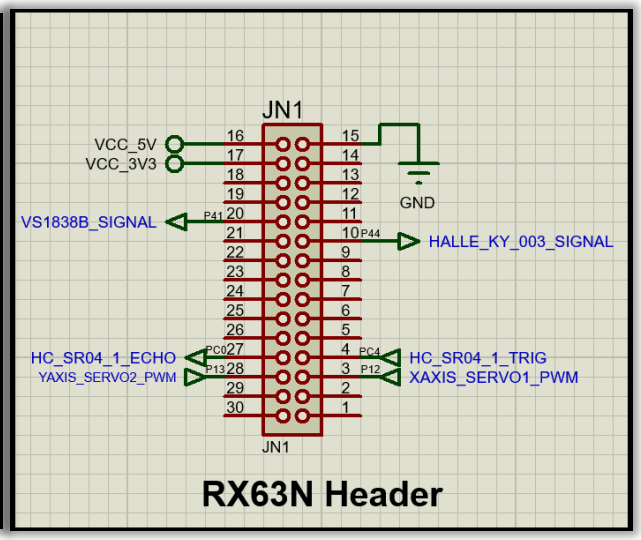


Figure 53 - Schematic - RX63N JN1 Header block

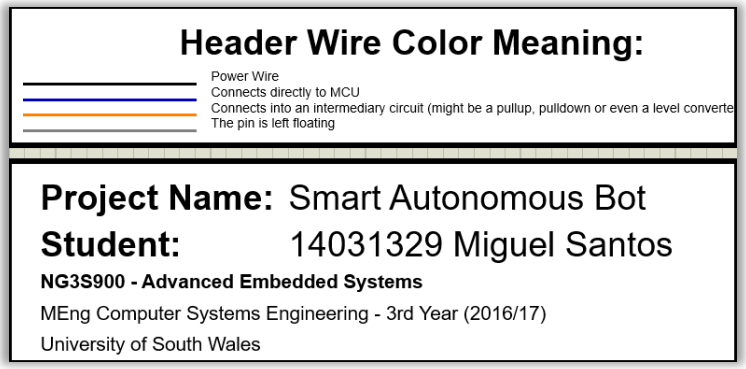


Figure 56 - Schematic title and information

Finally, after designing the schematic, the PCB was created from this same design.

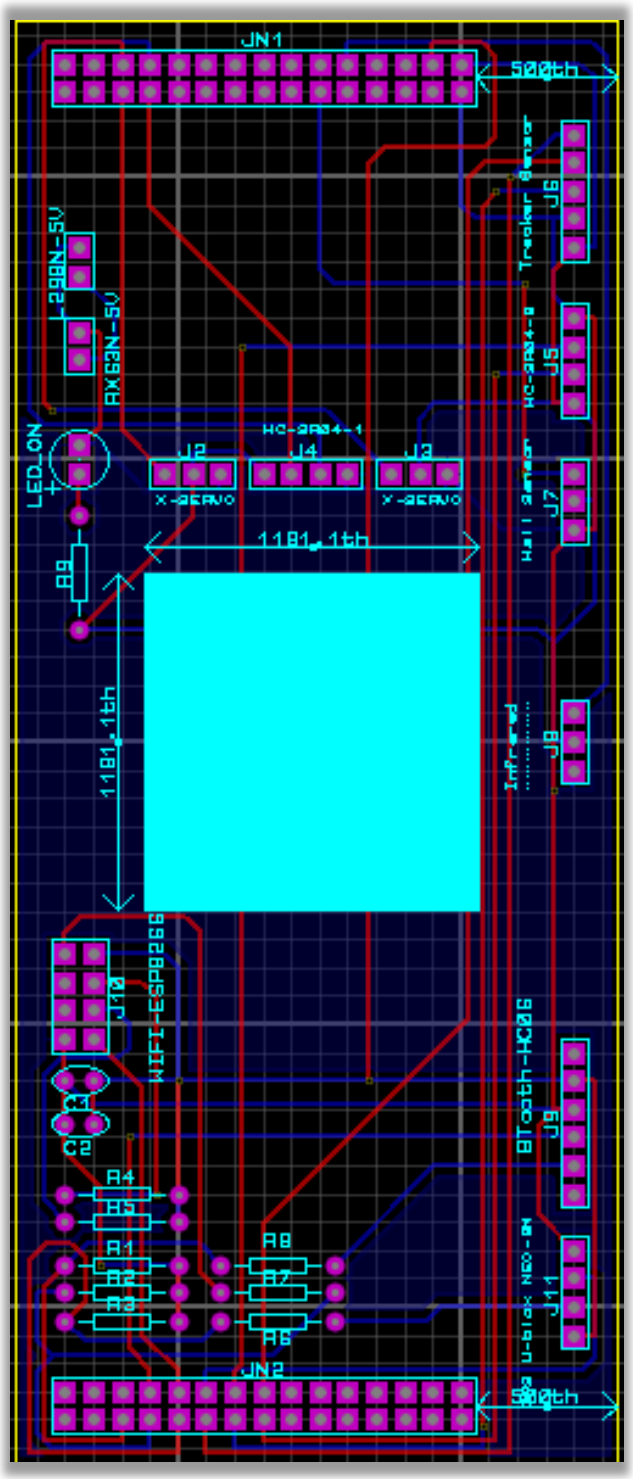


Figure 57 - PCB Layout (original display)

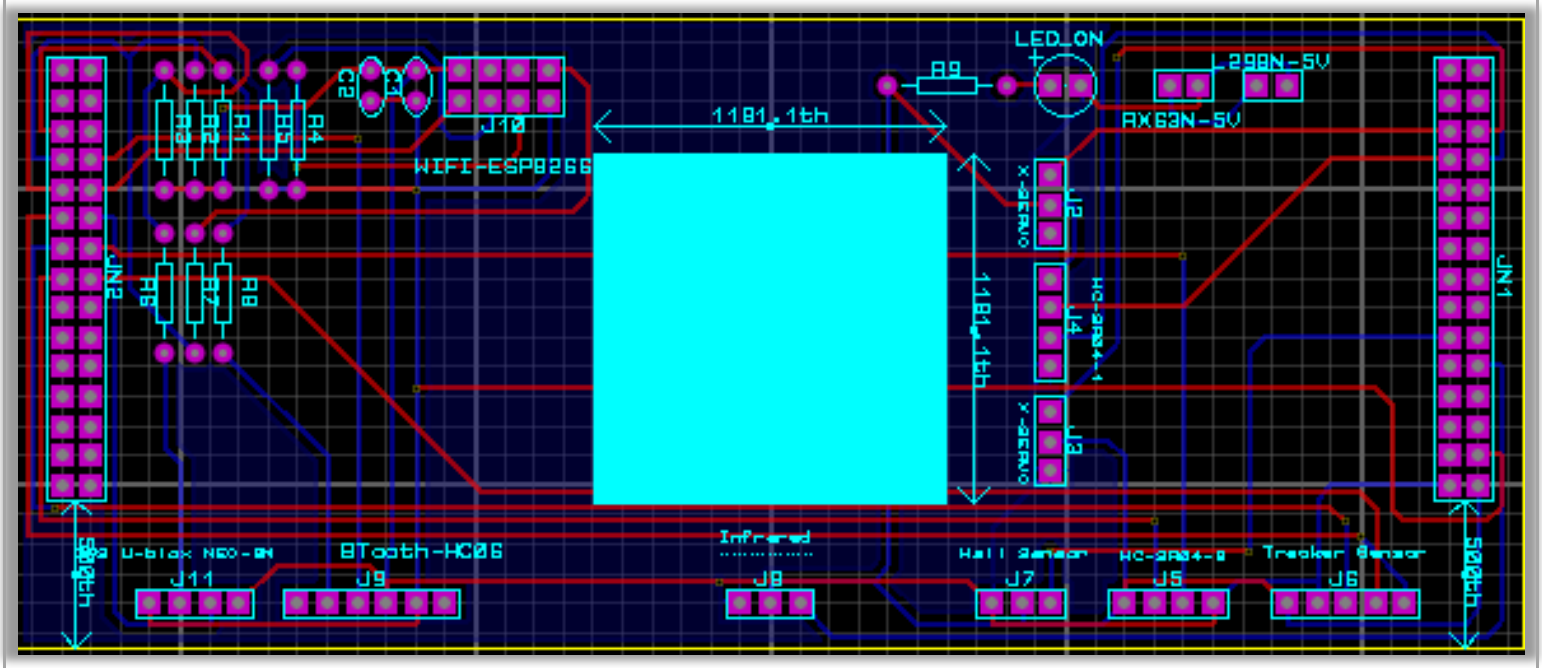


Figure 58 - PCB Layout (rotated 90º clockwise)

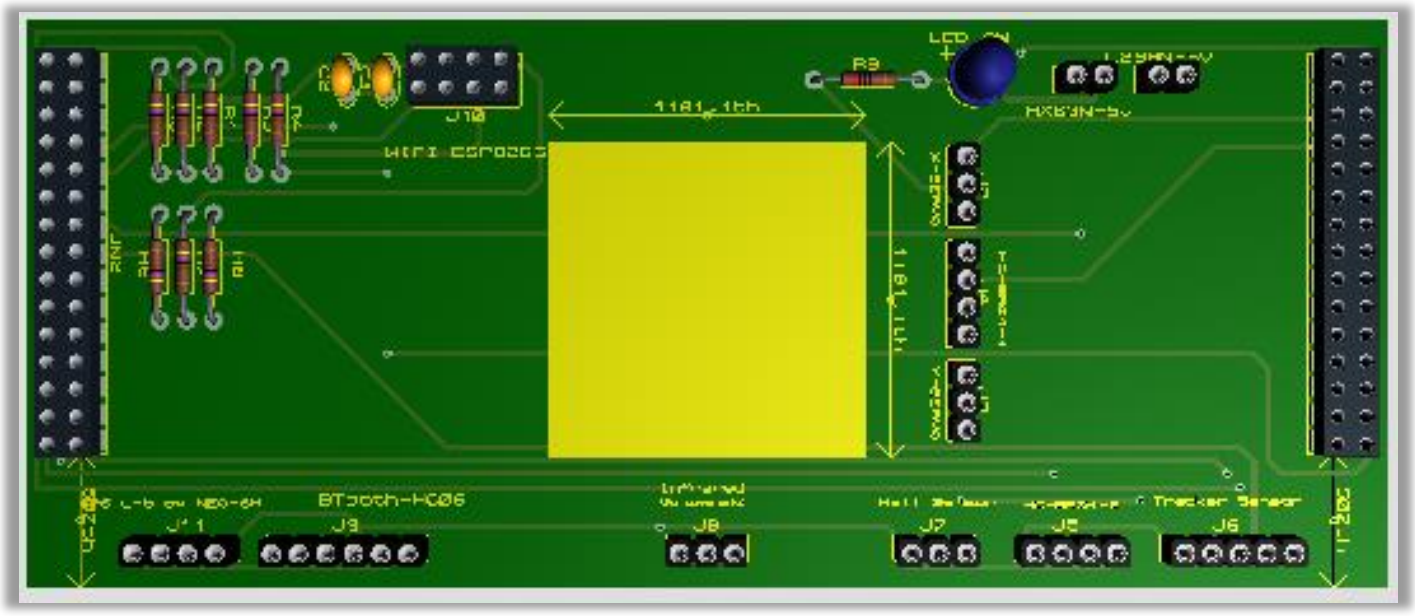


Figure 59 - PCB 3D View - Top

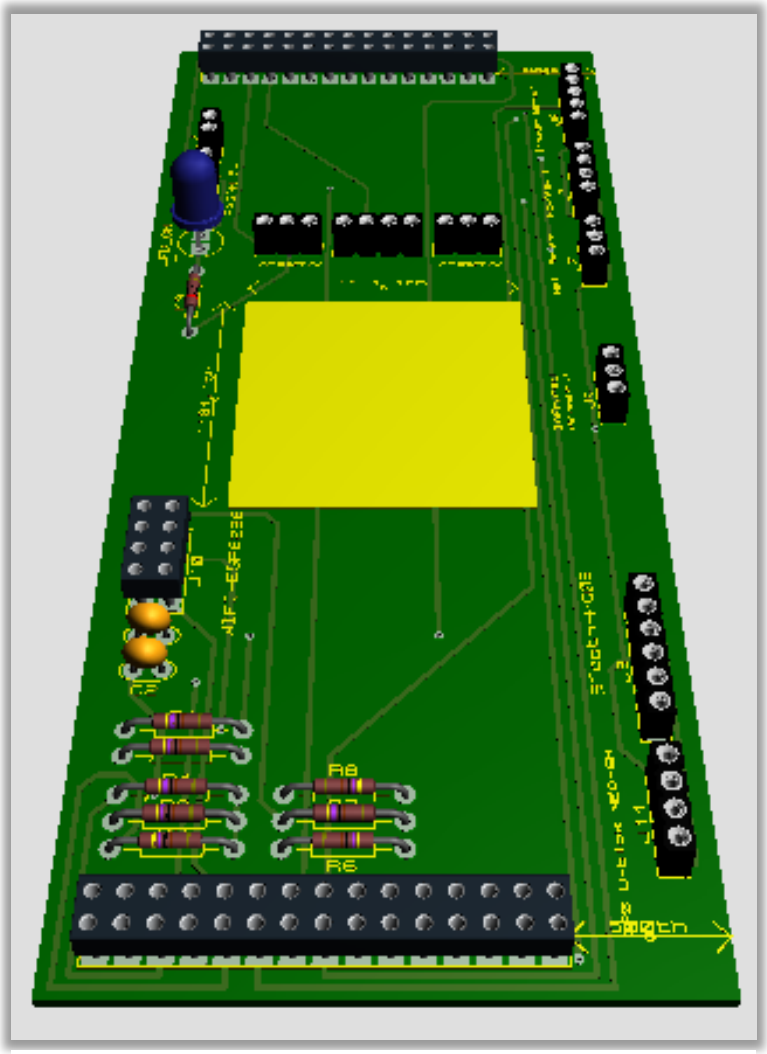


Figure 60 - PCB 3D View – Front

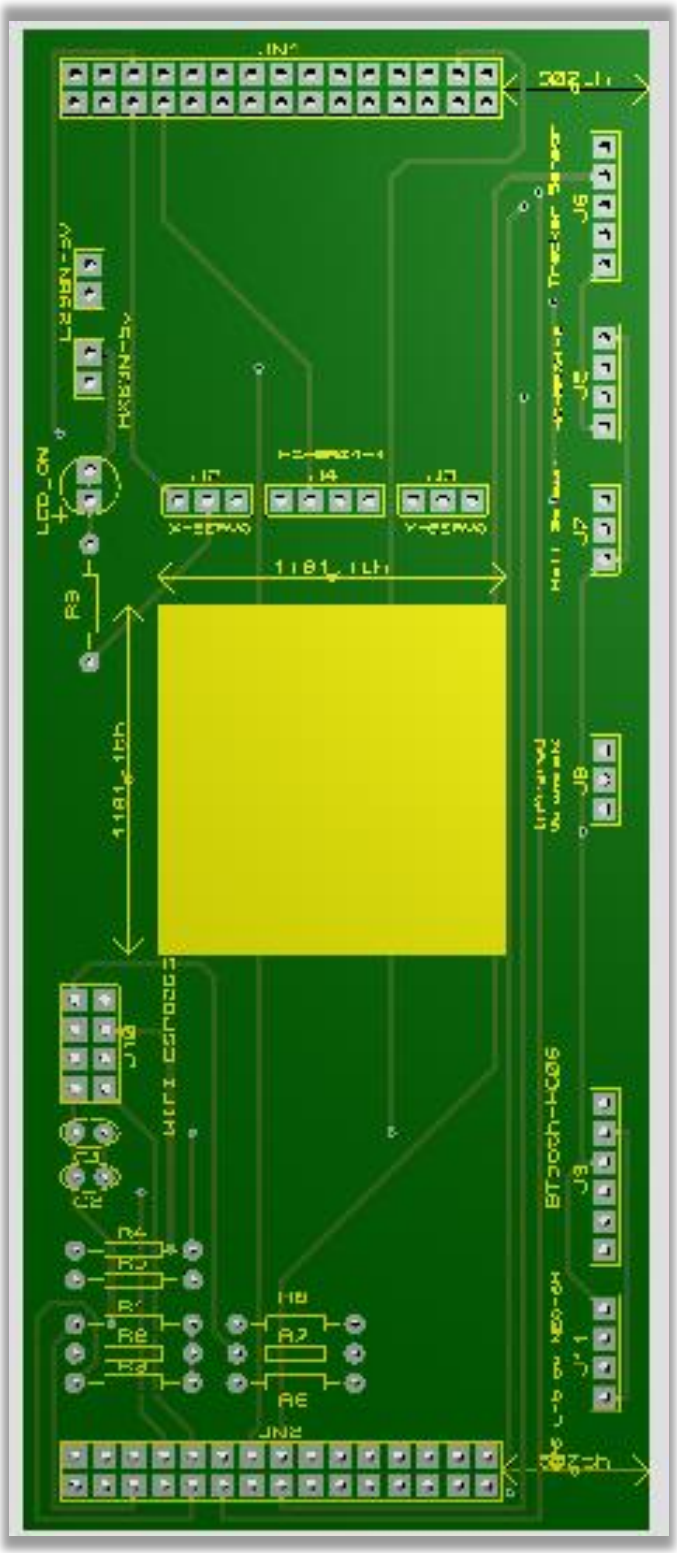


Figure 61 - PCB 3D View - Top no components

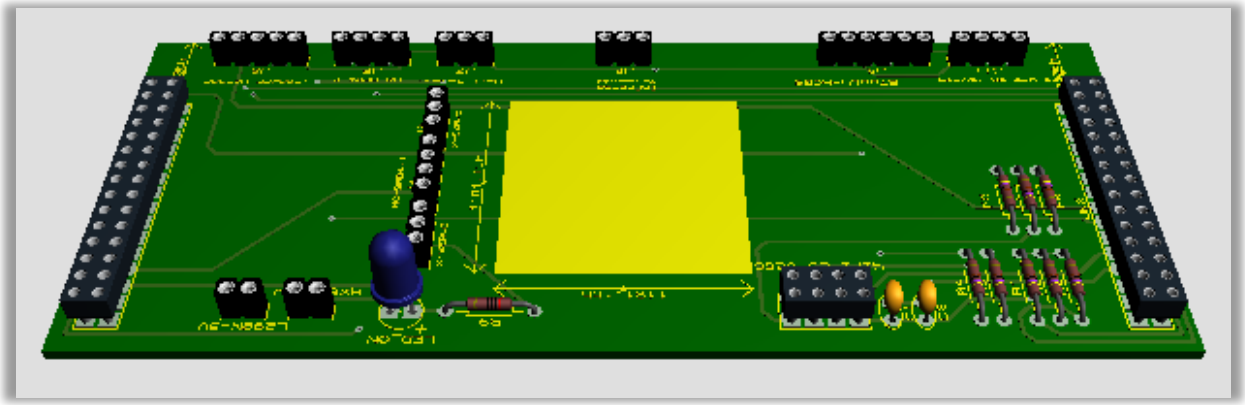


Figure 62 - PCB 3D View – Left

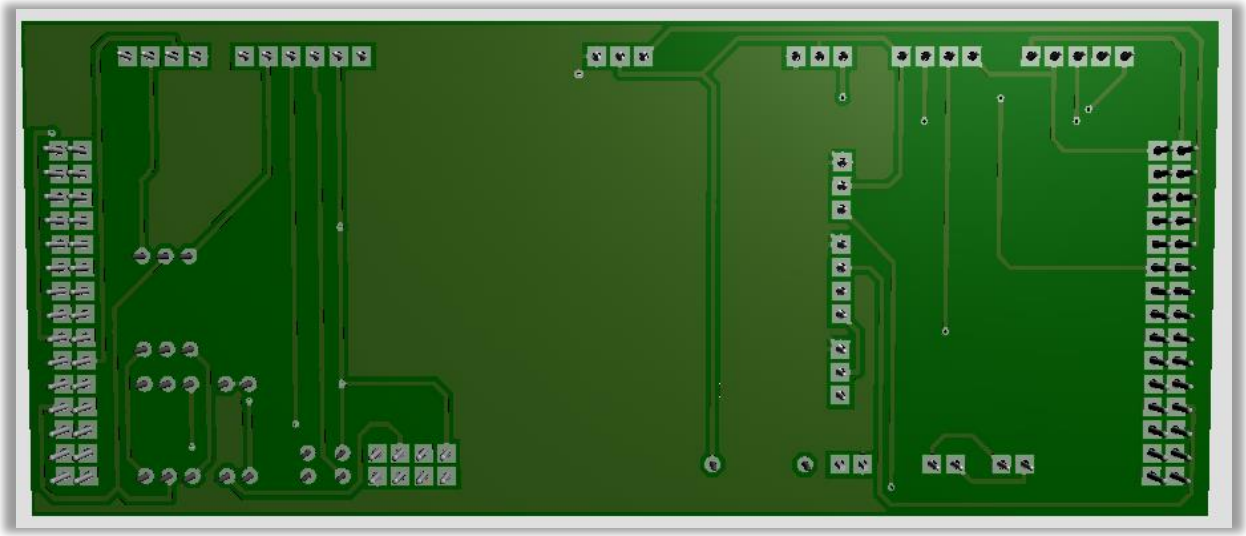


Figure 63 - PCB 3D View – Bottom

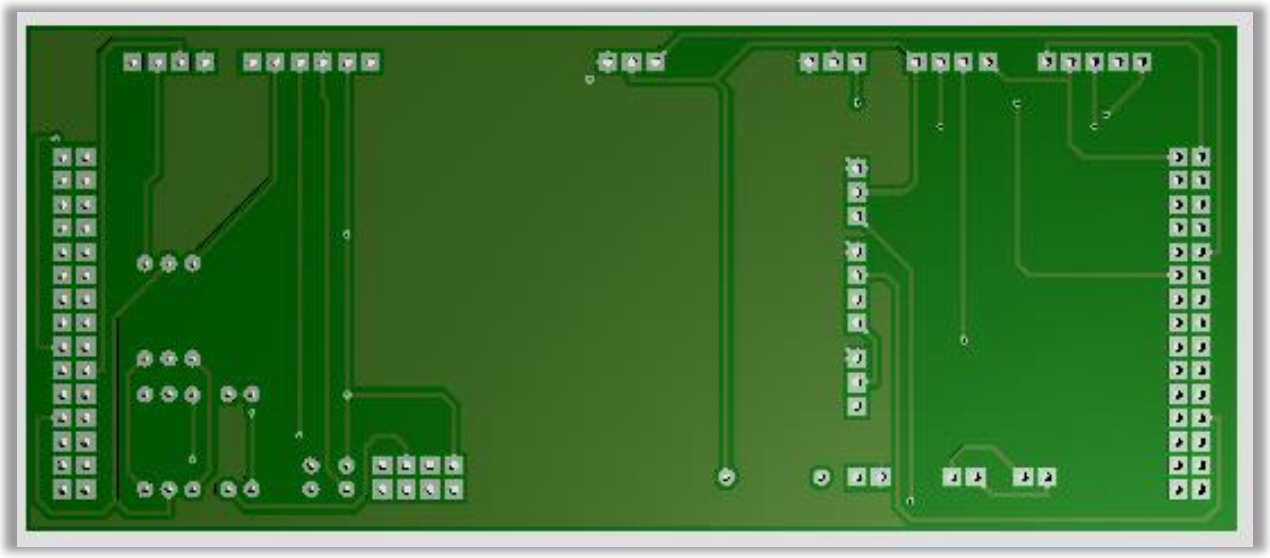


Figure 64 - PCB 3D View - Bottom no components

As can be seen, the board was designed to fit directly in the RX63N board. However, notice the width of the PCB. Its width does not match the RX63N board's width. This is because the Application Header was not being used, the PMOD1 only had 4 pins in use, and the LCD was being obstructed by the PCB.

The intended final result is illustrated on the following image:

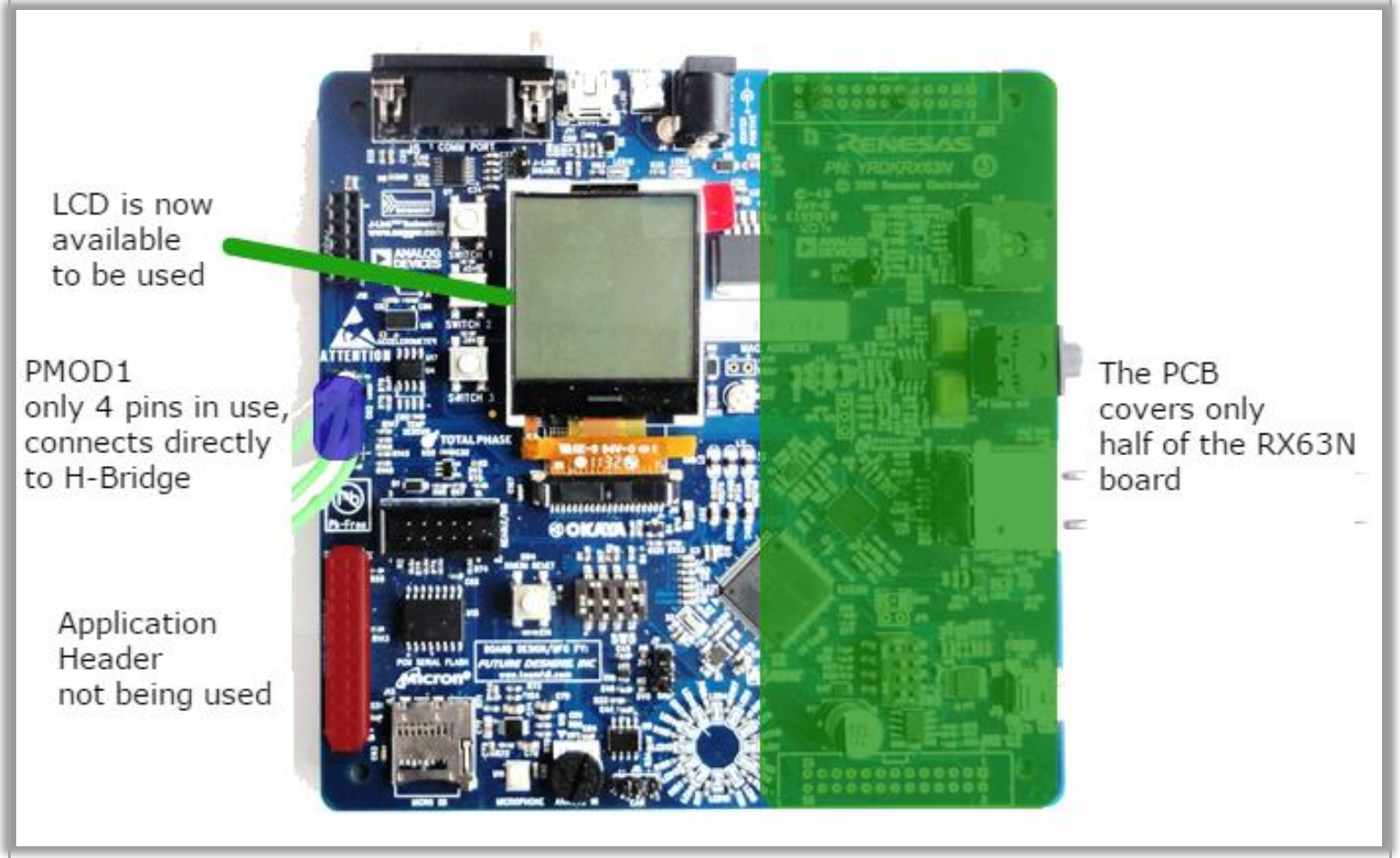


Figure 65 - PCB final result

2.B) BOT CHASSIS + BOM

As was previously shown on the task 1.B, the chassis of the car is made out of two plastic/acrylic structures, with 4 spacing screws on the four corners to add height to the car.

Figure 66 - Car Chassis

The RX63N and PCB should be connected to the car the following way.

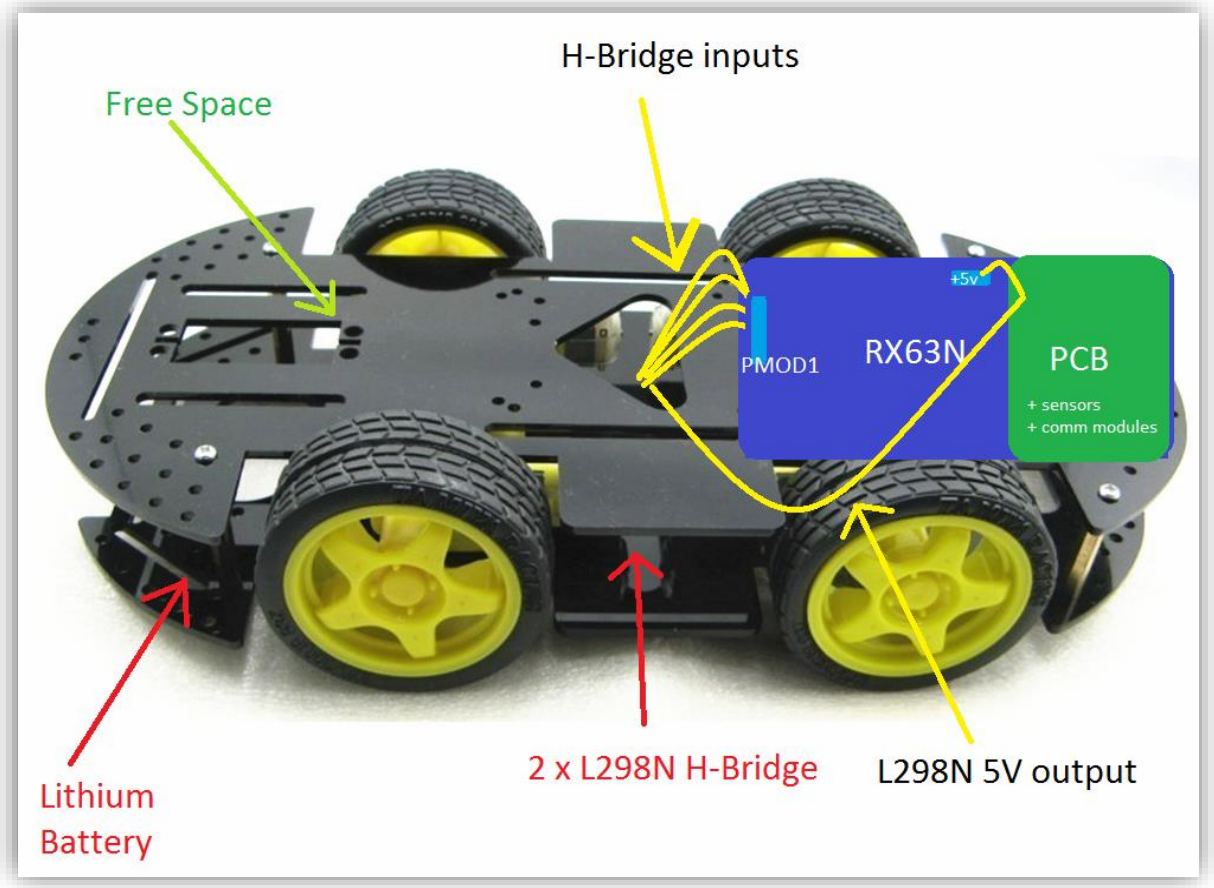


Figure 67 - Car Chassis with connections

As can be observed, the lithium battery powers the two L298N H-Bridges directly, the H-Bridges output a 5V into the RX63N using a voltage regulator and the RX63N then outputs again both 5v and 3.3v out through the JN1 header into the PCB.

Finally, the BOM for the entire car (excluding the RX63N board and other minor components that do not belong to the schematic) is shown below:

Bill Of Materials for Smart Autonomous Bot

Design Title	Smart Autonomous Bot
Author	14031329 Miguel Santos
Document Number	
Revision	V1
Design Created	14 january 2017
Design Last Modified	21 january 2017
Total Parts In Design	26

Category	Quantity	References	Value	Stock Code	Unit Cost
Capacitors	1	C1	47uF	526-1531	0,164
Capacitors	1	C2	0.1uF	830-9948	0,44
Resistors	8	R1-R8	4.7k	707-7726	0,017
Resistors	1	R9	200	148-332	0,135
Miscellaneous	1	J2	X-AXIS-SERVO1	132-0361	0,091
Miscellaneous	1	J3	Y-AXIS-SERVO2	132-0361	0,091
Miscellaneous	1	J4	ULTRASONIC_HC_SR04_1	828-2067	0,45
Miscellaneous	1	J5	ULTRASONIC_HC_SR04_2	681-2979	0,054
Miscellaneous	1	J6	TRACKER_SENSOR_IR	681-2985	0,097
Miscellaneous	1	J7	HALL-EFFECT-SENSOR	132-0361	0,091
Miscellaneous	1	J8	VS1838B_INFRARED	132-0361	0,091
Miscellaneous	1	J9	BLUETOOTH-HC05	828-2067	0,45
Miscellaneous	1	J10	WIFI-ESP8266	828-2067 x 2	0,45
Miscellaneous	1	J11	GPS-U-blox NEO-6M	828-2067	0,45
Miscellaneous	1	JN1	JN1	828-1541	1,34
Miscellaneous	1	JN2	JN2	828-1541	1,34
Miscellaneous	2	L298N-5V,RX63N-5V	CONN-SIL2	483-8461	0,102
Miscellaneous	1	LED_ON	LED	228-6004	0,136

Totals

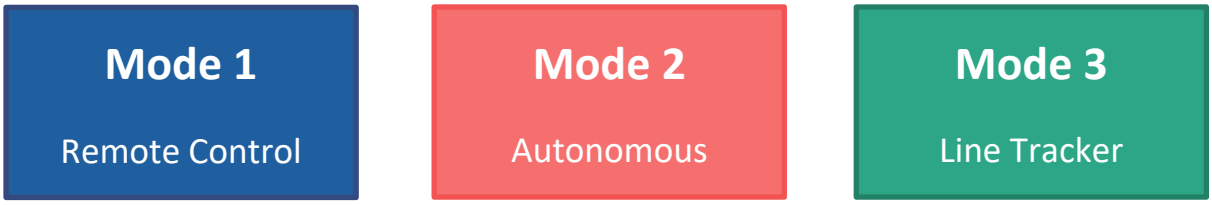
Category	Quantity	Unit Cost
Capacitors	2	£0,60
Resistors	9	£0,27
Miscellaneous	15	£5,34
Total	**26**	**£6,21**

Figure 68 - BOM list

2.C) WRITING THE ALGORITHM

The smart bot's top level application program is actually a collection of subprograms, that can be entered via user interaction and through other methods, which will be described on the next subsections.

The car is capable of performing 3 separate functions, or in other words, the car has 3 modes of function.



Each mode has its own algorithm, including the top level application, which is responsible of switching from one mode to another.

Please note that each sub-algorithm does not check in any of its stages if the user is trying to switch modes. The top level algorithm does this instead.

By doing this, it becomes extremely easy to introduce other sub-algorithms/programs into the 'super' application without having to worry about the other coexisting modes.

2.C.1) MODE 1 – REMOTE CONTROL ALGORITHM

This mode is the mode with the highest priority. Whenever the smart car starts up, it immediately enters this mode.

As the name of the mode describes, the remote control mode allows the user to control the car like a simple RC car through Bluetooth.

This control is being done using the Android app 'Arduino BT Joystick', which is a simple program for controlling robots via Bluetooth.

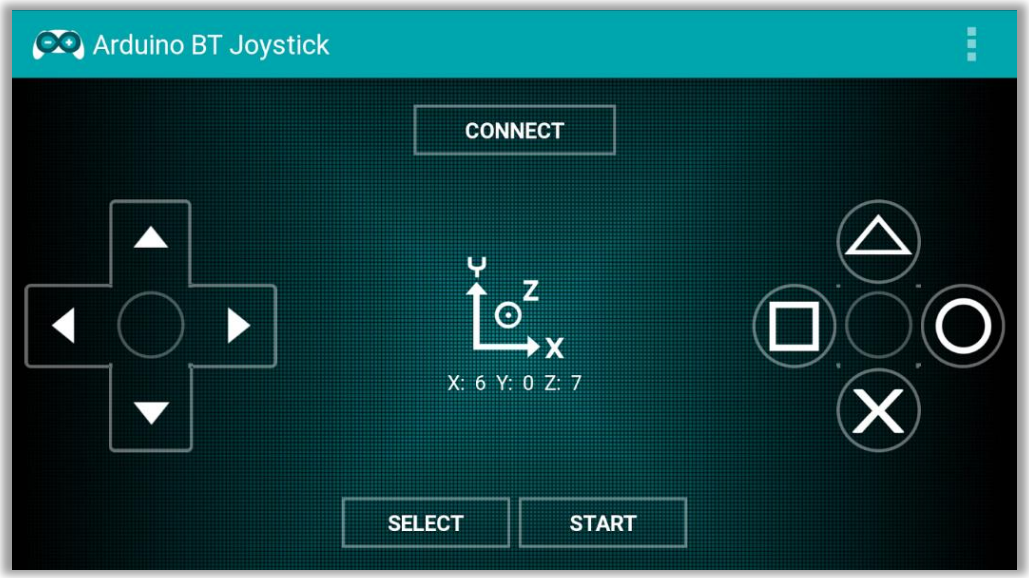
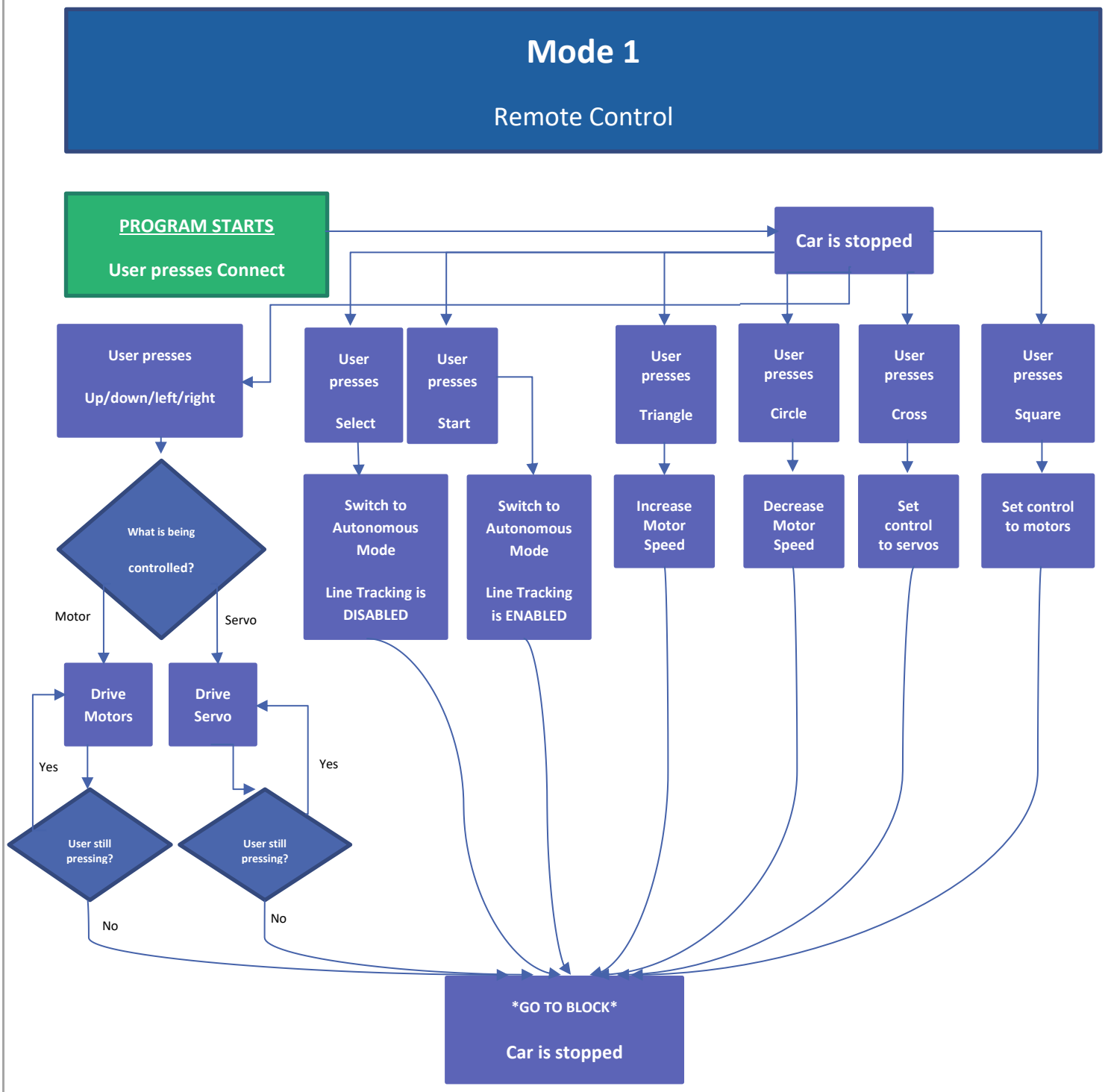


Figure 69 - Bluetooth Remote Controller App

Note: the following pages of this report, which include subtask 2.C.4), will mention 'user' and 'button' multiple times. The button in question is the virtual button of this Android application.

The following flowchart represents the state of the algorithm as the user presses the buttons on the Android Application.



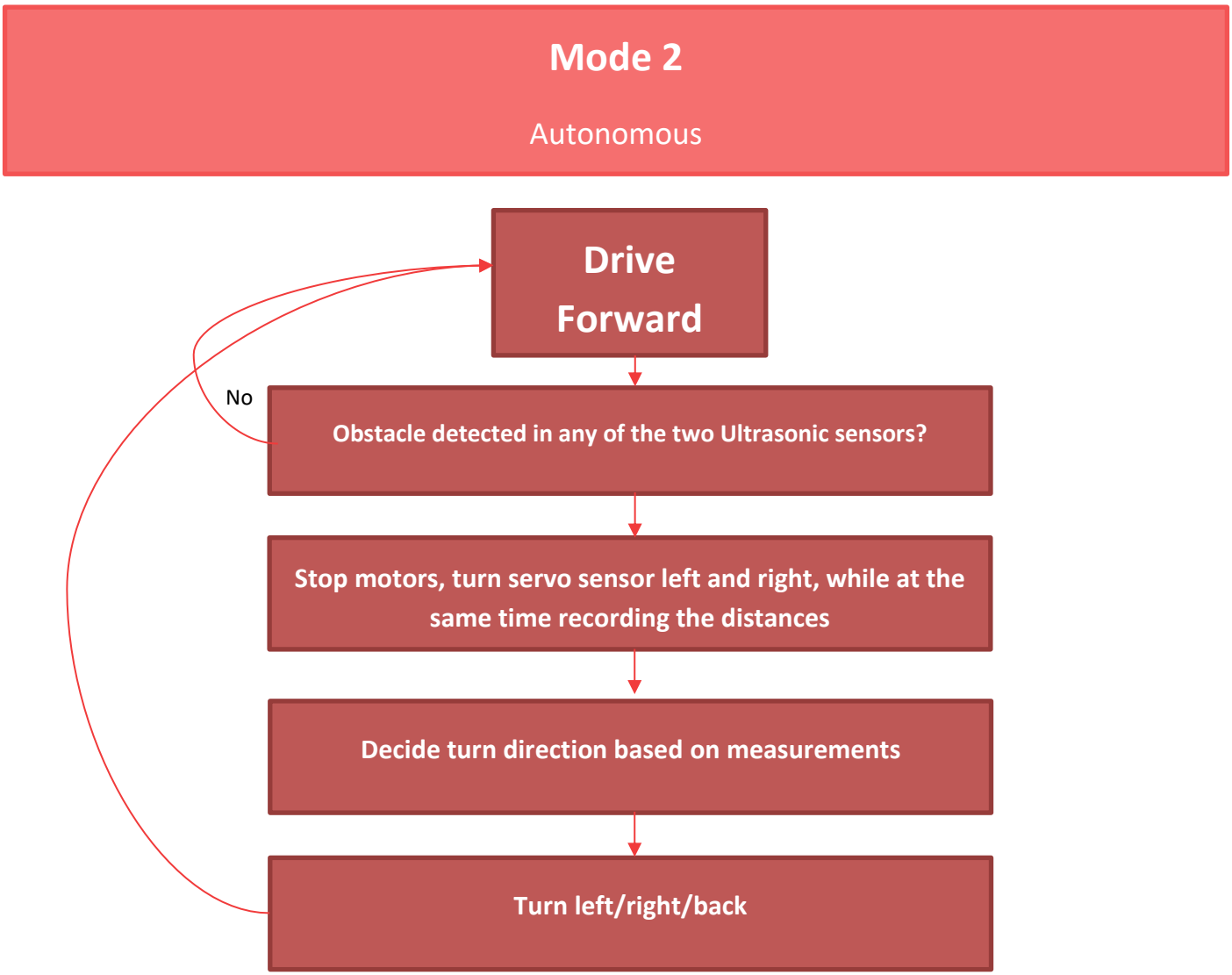
2.C.2) MODE 2 – AUTONOMOUS ALGORITHM

This mode is the mode with the lowest priority (in terms of system control overriding privileges).

What this means is, if the car is in autonomous mode and the user presses any button (other than select and start,) the car will switch immediately from autonomous mode to remote control mode. The reason the algorithm works this way is to allow the user to override the controls in case the car is auto-driving itself into a very dangerous place, like a hole or water for example.

Remember that the top level does this management, and not the 'autonomous sub-program'.

The algorithm for this mode is:

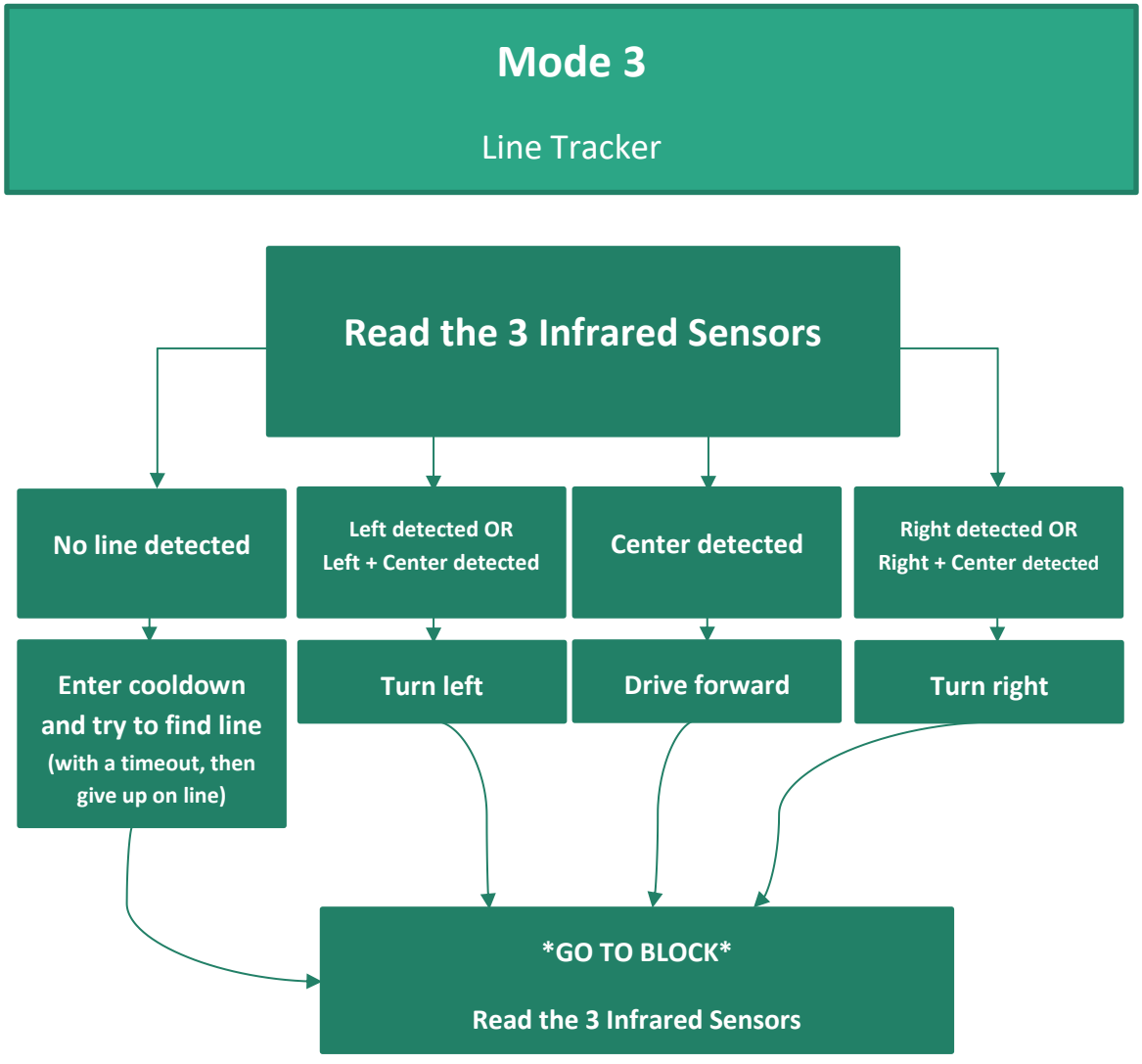


2.C.3) MODE 3 – LINE TRACKER ALGORITHM

In this mode, the car tries to stay on a black/white line, by using three infrared sensors on the bottom of the car chassis.

Its priority is medium, as it can override the autonomous mode (as soon as it finds a line while in autonomous mode,) but cannot override the remote control mode. Also, the remote control mode can override this mode by switching immediately from line tracker mode to remote mode, without transitioning to autonomous mode first.

Algorithm:



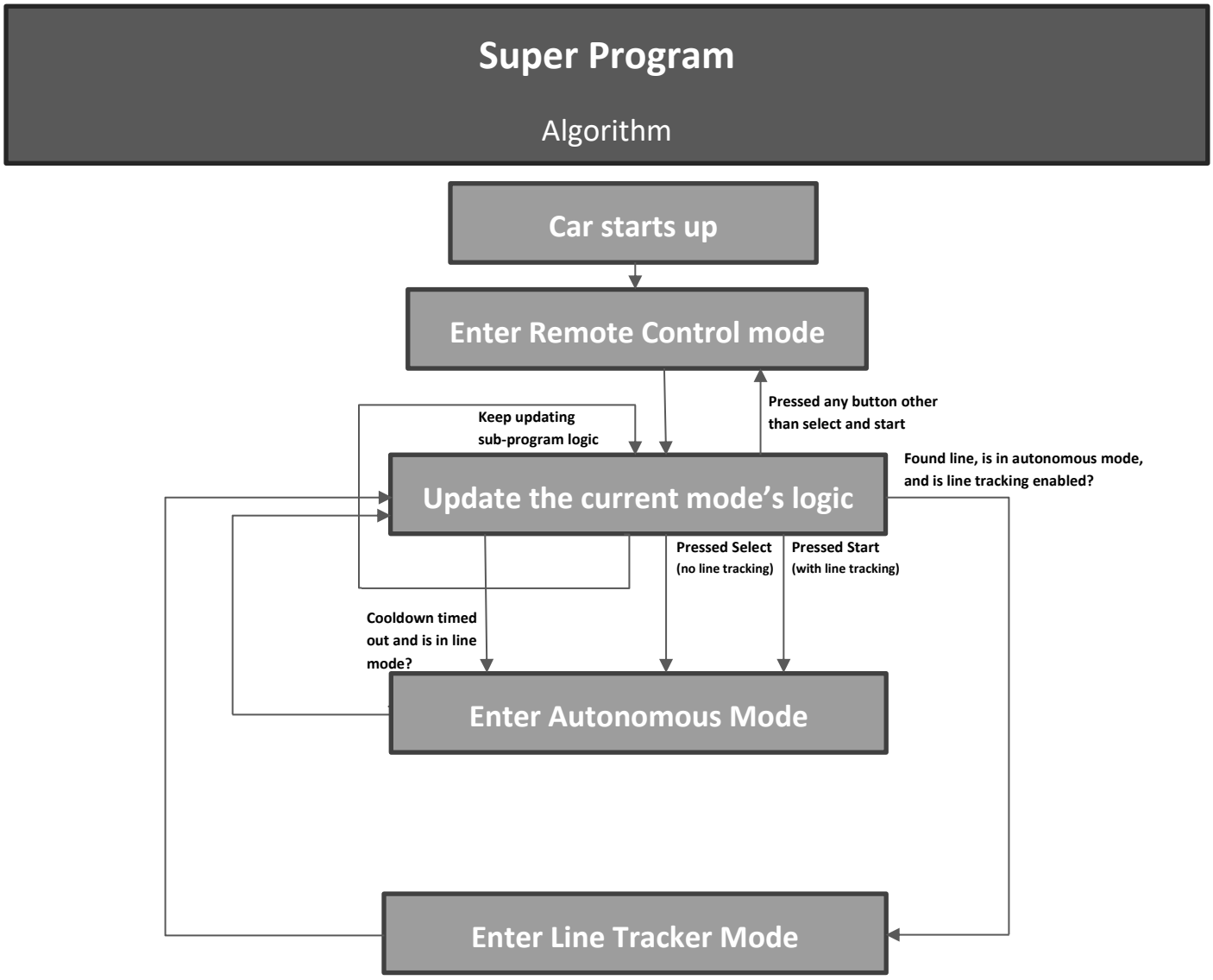
When the car loses track of the line, a counter starts ticking. While the counter is ticking, the car is turning to the previous known direction trying to re-enter the line. If the counter reaches the maximum value, the top-level algorithm will switch the car mode from line tracker back to autonomous.

If the car is in autonomous mode and it finds a line (it doesn't matter if it's the first or second time the car is finding the line while in this mode,) the top-level algorithm will switch the mode from autonomous to line-tracker mode.

Finally, there is an exception to the flowchart on the previous page. If the car detects a line on both the left and right sensors, it means the line has a reversed color and / or is malformed. The car treats this kind of line as an error and it tries to find a valid line by driving forward instead of trying to drive on the malformed line. If no line is found within a given time, the mode will be switched back to autonomous.

2.C.4) COMPLETE ALGORITHM FLOW

The top level of the program, which could be called super program (constituted of multiple sub-programs), manages the sub-programs and the modes the following way:



TASK 3 - IMPLEMENTATION

3.A) ASSEMBLING THE BOT

The following images show the process of mounting the chassis together with the RX63N board.

- **Step 1**: Organise the chassis components

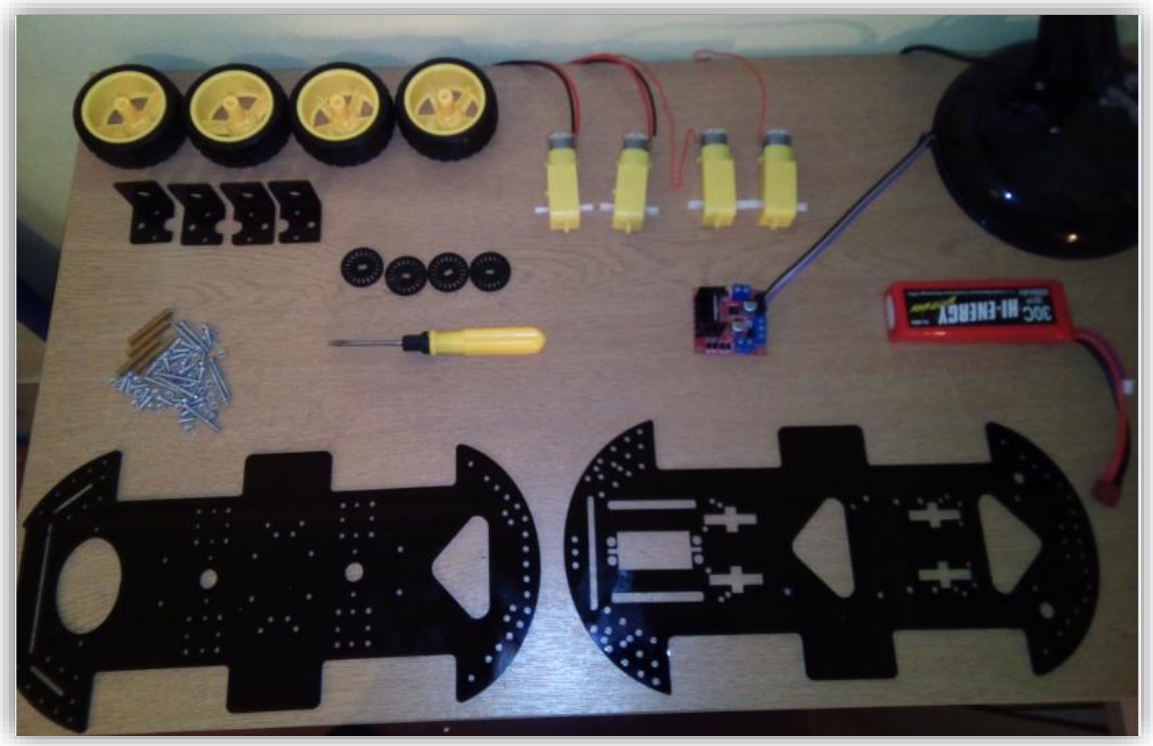


Figure 70- Mounting the chassis - chassis parts

- **Step 2**: Mount and screw the DC Motor holders on the lower side of the chassis

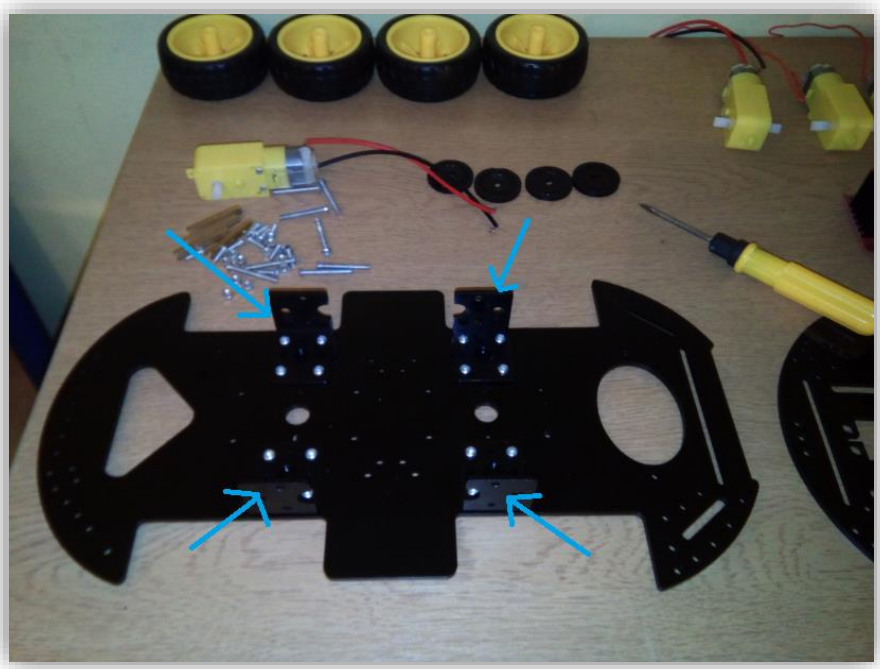


Figure 71 - Mounting the chassis – DC Motor Holders

- **Step 3**: Place and screw the motors on the holders.

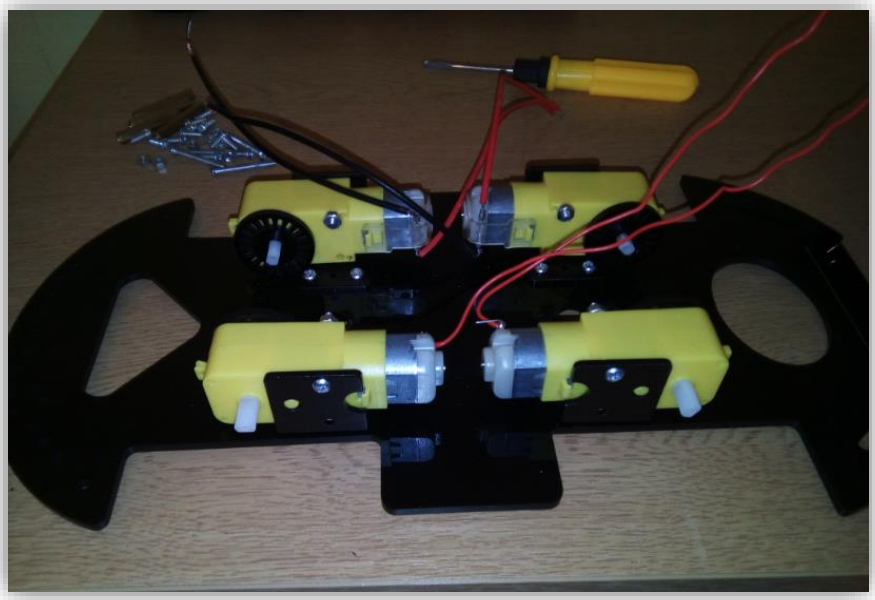


Figure 72 - Mounting the chassis - Screwing the motors

- **Step 4**: Place the wheels on the DC motors

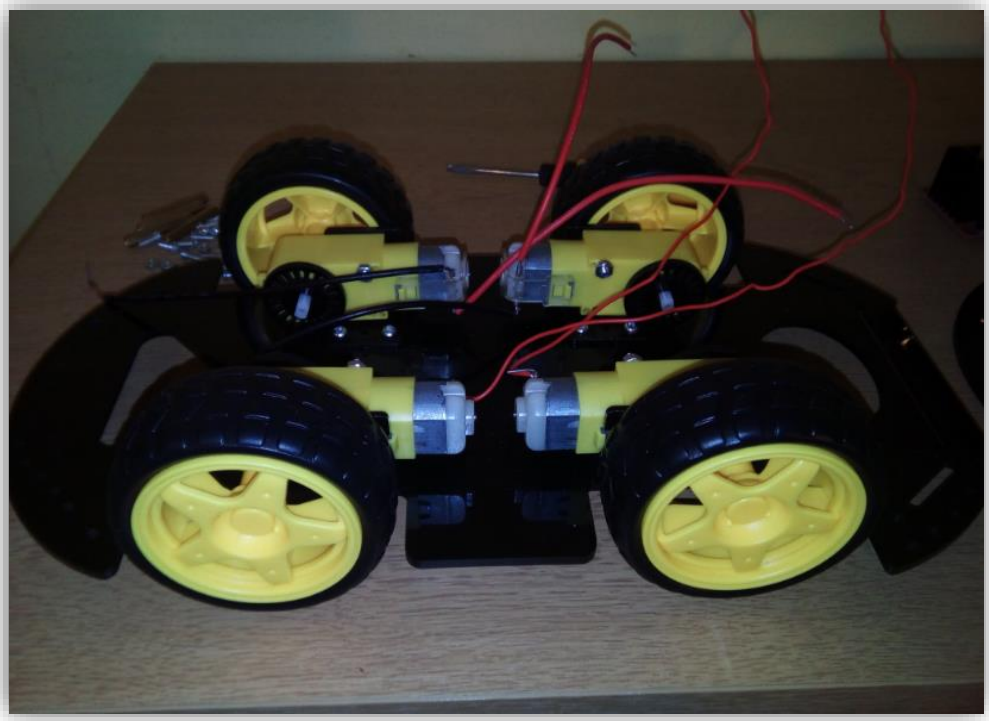


Figure 73 - Mounting the chassis - Placing the wheels

Important Note: the next step after placing the wheels would be to simply put the top part of the chassis, like so:

Figure 74 - Mounting the chassis - Top side detail

However, this setting does not reserve enough space for the RX63N board. Therefore, the car had to be flipped upside down.

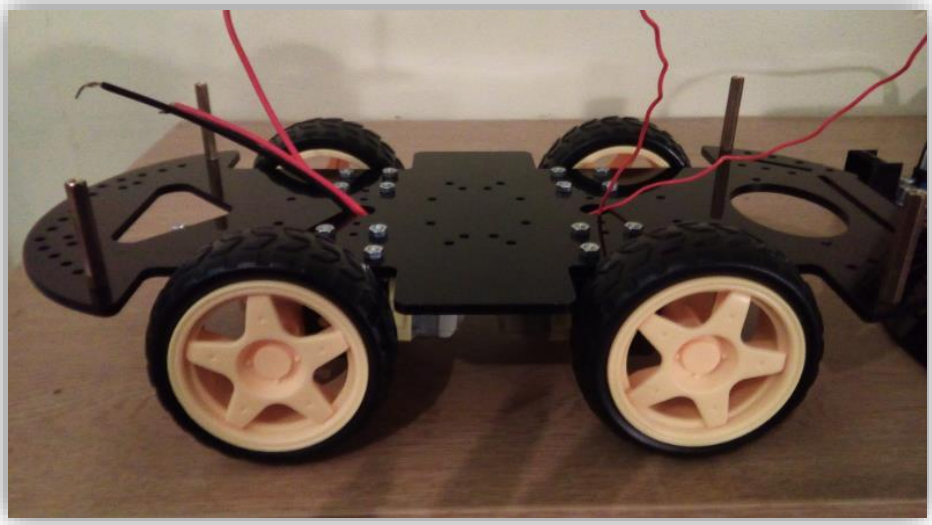


Figure 75 - Mounting the chassis - upside down car

This way, the top part of the chassis will give enough height for the RX63N board.

- **Step 5**: Place the Lithium battery and both H-Bridges (the photo currently only shows one h-bridge) on top of the car.

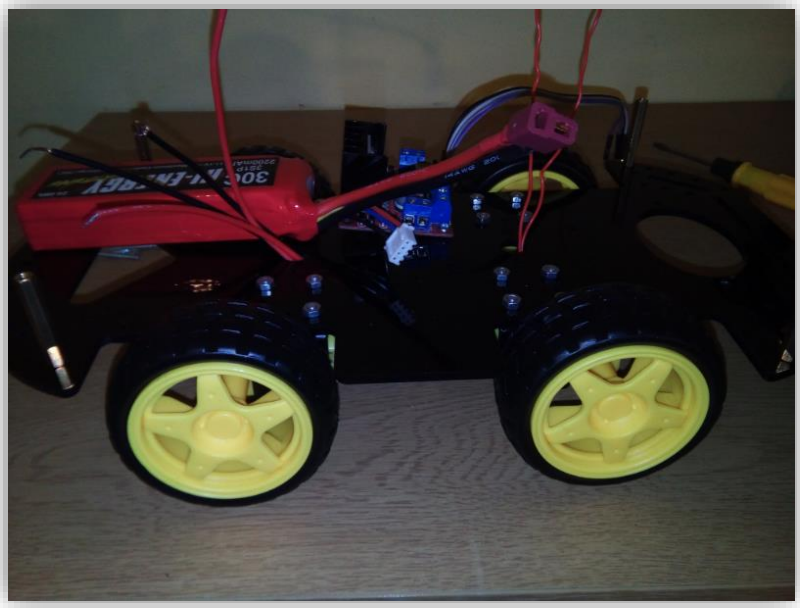


Figure 76 - Mounting the car - placing the battery and H-bridges

- **Step 6**: Screw the top side of the chassis on the four screws in each corner of the car and place the RX63N board on top of that.

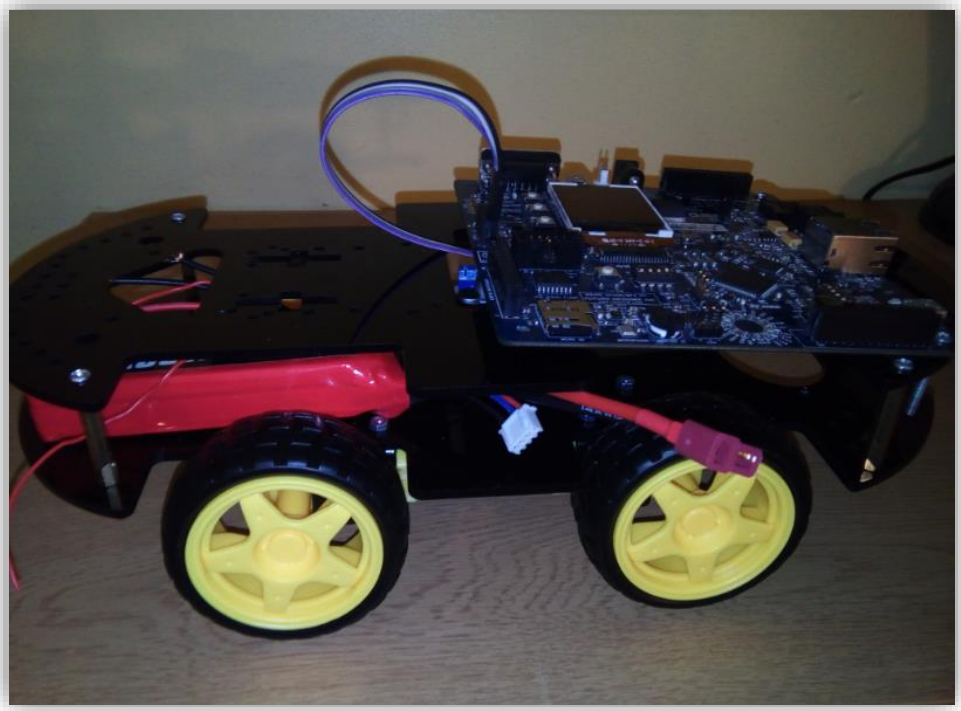


Figure 77 - Mounting the chassis - placing the RX63N board on the car

- **Step 7**: Finally, place all the sensors around the chassis and connect them to the RX63N board.

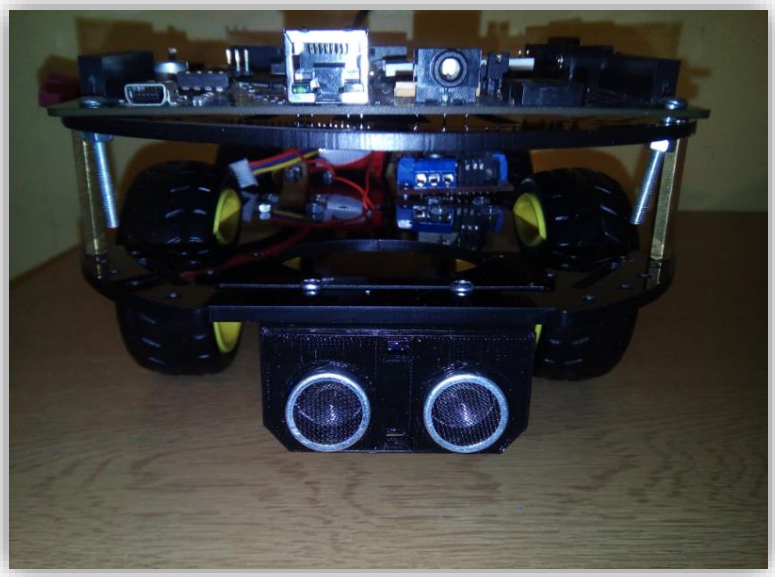


Figure 78 - Mounting the chassis - placing the sensors

After completely mounting the car, these are the expected results.

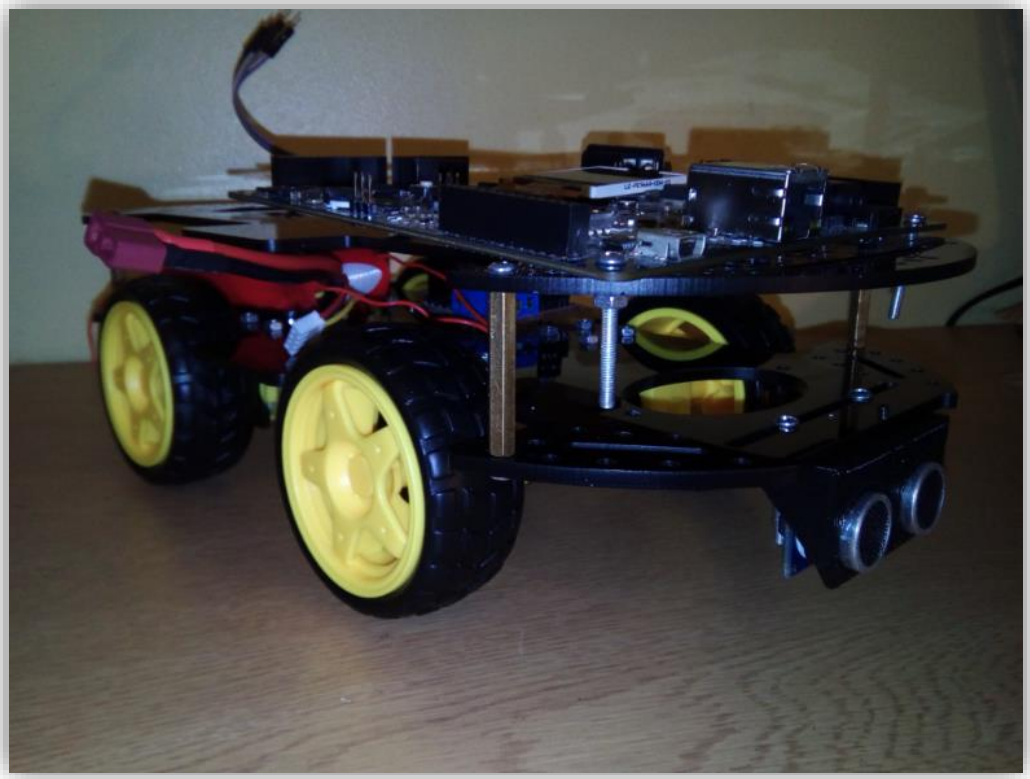


Figure 79 - Mounted chassis expected result

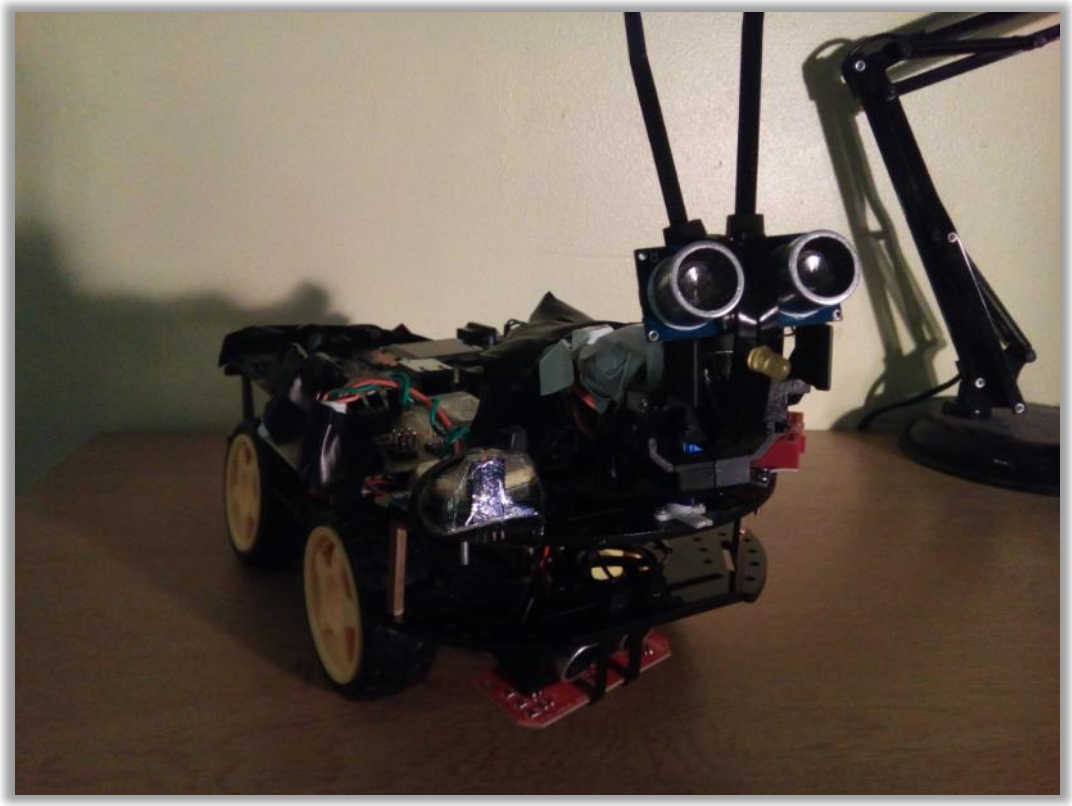


Figure 80 - Car Completed - photo 1

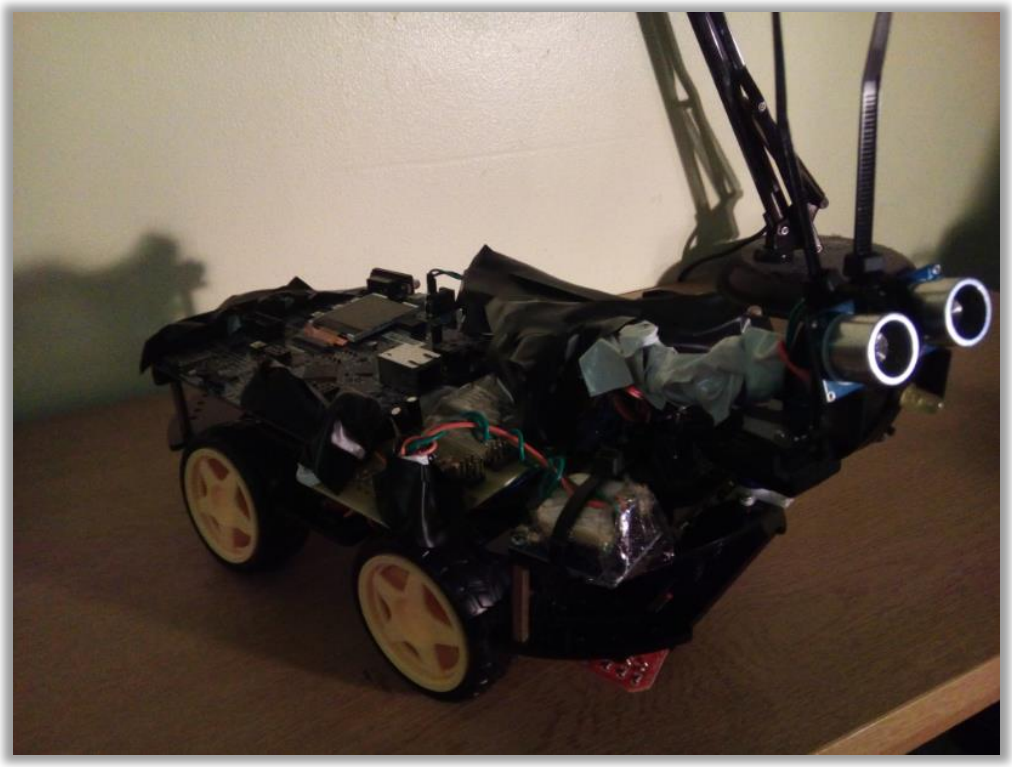


Figure 81 - Car Completed - photo 2

Figure 82 - Car Completed - photo 3

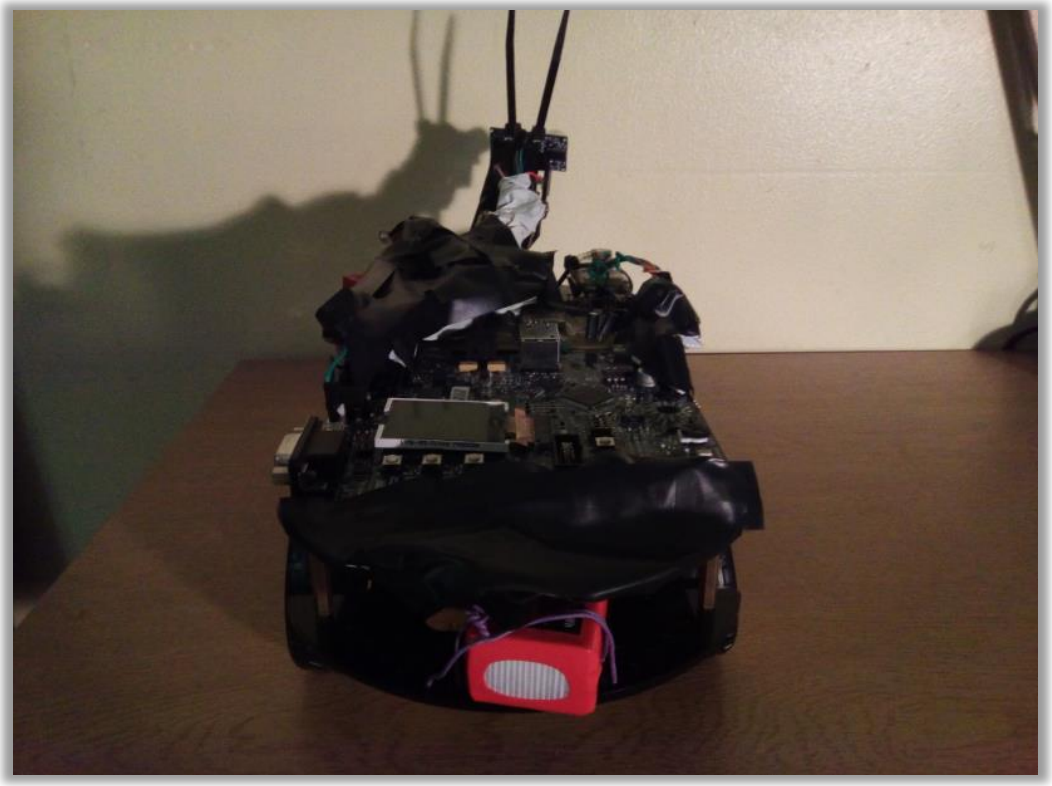


Figure 83 - Car Completed - photo 4

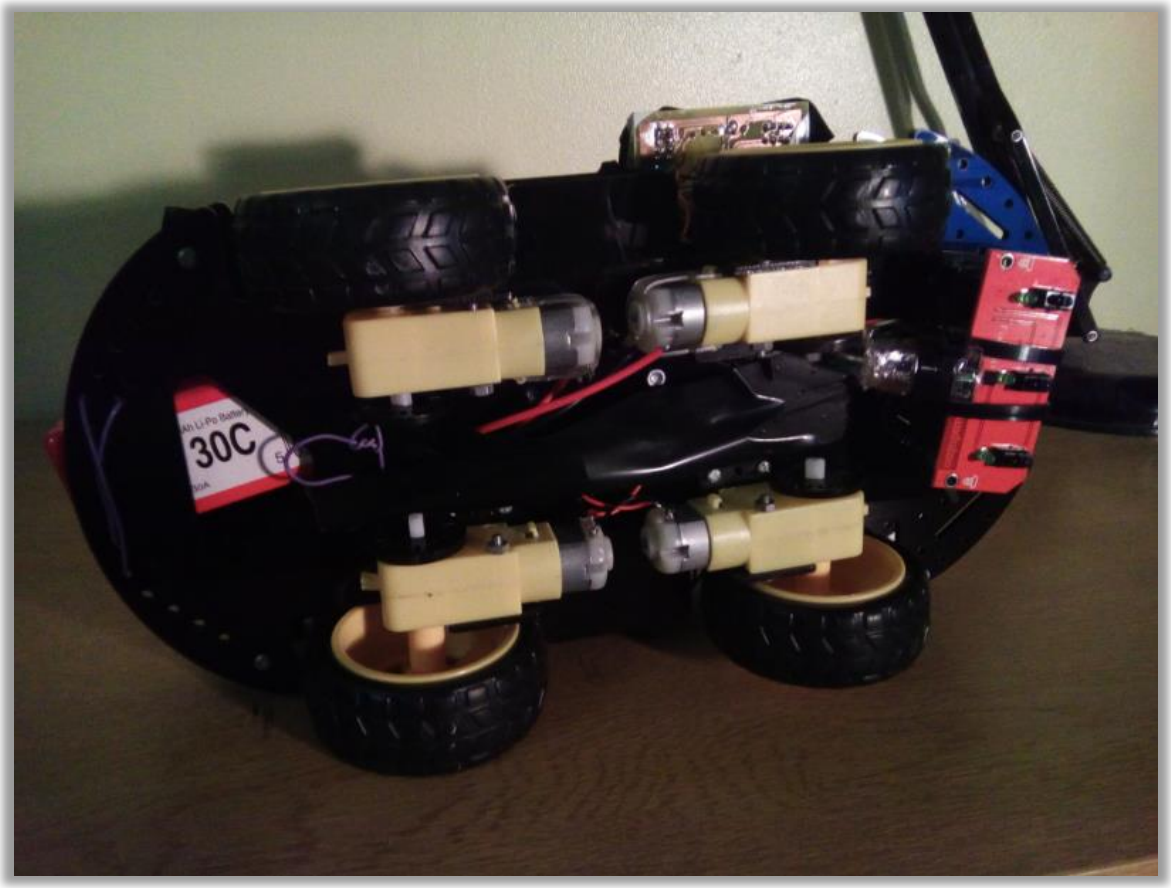


Figure 84 - Car Completed - photo 5

3.B) WRITING THE CODE

3.B.1) ULTRASONIC SENSOR CODE

```
#include <stdint.h>
#include <stdio.h>
#include "platform.h"
#include "r_gpio_rx_if.h"
#include "r_cmt_rx_if.h"
#include "lcd.h"

#define TRIG GPIO_PORT_D_PIN_5
#define ECHO GPIO_PORT_D_PIN_3

volatile bool enable_timer = false;
volatile uint32_t delta_time = 0;

void timer_cback(void * arg) {
        /* This timer will trigger every 1 uS. It will allow us to create delays */
        if(enable_timer) delta_time++;
        else delta_time = 0;
}

void delay_us(uint32_t timeout) {
        enable_timer = true;
        while(delta_time < timeout);
        enable_timer = false;
}

uint32_t pulseIn(gpio_port_pin_t read_pin, uint8_t trig_level, uint32_t timeout) {
        enable_timer = true;
        if(trig_level) {
                /* Wait for a logic 1 on the read pin: */
                while(!R_GPIO_PinRead(read_pin)) if(delta_time >= timeout && timeout > 0 && (int32_t)timeout != -1)
                        break;
                /* Wait for a logic 0 on the read pin: */
                while(R_GPIO_PinRead(read_pin))  if(delta_time >= timeout && timeout > 0 && (int32_t)timeout != -1)
                        break;
        } else {
                /* Wait for a logic 0 on the read pin: */
                while(R_GPIO_PinRead(read_pin))  if(delta_time >= timeout && timeout > 0 && (int32_t)timeout != -1)
                        break;
                /* Wait for a logic 1 on the read pin: */
                while(!R_GPIO_PinRead(read_pin)) if(delta_time >= timeout && timeout > 0 && (int32_t)timeout != -1)
                        break;
        }
        uint32_t ret = delta_time;
        enable_timer = false;
        return ret;
}

uint32_t hc_sr04_read(gpio_port_pin_t echo_pin) {
        #define MAX_DISTANCE 200

        /* Send a pulse to the trigger pin: */
        R_GPIO_PinWrite(TRIG, GPIO_LEVEL_HIGH);
        delay_us(10);
        R_GPIO_PinWrite(TRIG, GPIO_LEVEL_LOW);

        /* Now read the echo pulse: */
        uint32_t distance = (pulseIn(ECHO, 1, 0))/58.2-10;
```

```
58          return distance >= MAX_DISTANCE ? MAX_DISTANCE : distance;
59  }
60
61  void main(void) {
62          /* Initialize LCD: */
63          lcd_initialize();
64          lcd_clear();
65
66          /* Initialize GPIO pins: */
67          R_GPIO_PinDirectionSet(ECHO, GPIO_DIRECTION_INPUT);
68          R_GPIO_PinDirectionSet(TRIG, GPIO_DIRECTION_OUTPUT);
69          R_GPIO_PinWrite(TRIG, GPIO_LEVEL_LOW);
70
71          /* Initialize CMT timer: */
72          R_CMT_CreatePeriodic(1000000, timer_cback, (uint32_t*)&CMT0);
73
74          for(;;) {
75                  /* Read HC-SR04: */
76                  uint32_t distance = hc_sr04_read(ECHO);
77
78                  /* And display its value: */
79                  char buff[50];
80                  sprintf(buff, "-> %d cm   ", distance);
81                  lcd_display(LCD_LINE1, (const uint8_t*)buff);
82
83                  /* Wait 50 ms until the next reading: */
84                  delay_us(50000);
85          }
86  }
```

3.B.2) BLUETOOTH CODE

```
1   #include <stdio.h>
2   #include <string.h>
3   #include "suart/suart.h"
4   #include "lcd.h"
5
6   #define TX GPIO_PORT_2_PIN_4
7   #define RX GPIO_PORT_E_PIN_3
8
9   void rx_cback(uint8_t * bytes, uint32_t bufflen, uint8_t incoming_channel) {
10        /* Echo back the received buffer: */
11        suart_tx_buff(bytes, bufflen, incoming_channel);
12  }
13
14  void main(void) {
15        /* Initialize Soft UART: */
16        suart_init(rx_cback, 9600, 0, TX, RX);
17        /* And do nothing. Everything happens on the callback 'rx_cback' */
18        while(1);
19  }
```

Please note that the Bluetooth module uses the UART protocol. This protocol has been implemented using a library that was implemented by the student. This implementation is hidden, and no explanation will be given for the functions '`suart_init`' and '`suart_tx_buff`', as they can be found on the source code. The reason is that the library is very complex and it covers C features / algorithms / data structures that are not on the scope of this project.

The libraries were implemented because they were absolutely necessary for wireless communication to be possible within multiple channels (not just through Bluetooth).

3.B.3) H-BRIDGE TEST CODE (NOT OPTIMIZED)

```
1   #include "platform.h"
2   #include "r_gpio_rx_if.h"
3   #include "r_cmt_rx_if.h"
4
5   volatile uint32_t delta_timer = 0;
6   volatile bool enable_timer = false;
7
8   void timer_cback(void * args) {
9           if(enable_timer) delta_timer++;
10          else delta_timer = 0;
11  }
12
13  void delay_ms(uint32_t ms) {
14          enable_timer = true;
15          while(delta_timer < ms * 1000);
16          enable_timer = false;
17  }
18
19  enum MOTOR_TYPE {
20          MOTOR_LEFT,
21          MOTOR_RIGHT
22  };
23
24  enum MOTOR_DIRECTION {
25          MOTOR_STOP1,
26          MOTOR_FORW,
27          MOTOR_BACK,
28          MOTOR_STOP2
29  };
30
31  void motor_ctrl(char whichmotor, char dir) {
32          switch(whichmotor) {
33          case 0:
34                  if(!dir) {
35                          R_GPIO_PinWrite(GPIO_PORT_E_PIN_4, 0);
36                          R_GPIO_PinWrite(GPIO_PORT_E_PIN_6, 0);
37                  } else if(dir == 1) {
38                          R_GPIO_PinWrite(GPIO_PORT_E_PIN_4, 1);
39                          R_GPIO_PinWrite(GPIO_PORT_E_PIN_6, 0);
40                  } else if(dir == 2) {
41                          R_GPIO_PinWrite(GPIO_PORT_E_PIN_4, 0);
42                          R_GPIO_PinWrite(GPIO_PORT_E_PIN_6, 1);
43                  } else {
44                          R_GPIO_PinWrite(GPIO_PORT_E_PIN_4, 1);
45                          R_GPIO_PinWrite(GPIO_PORT_E_PIN_6, 1);
46                  }
47                  break;
48          case 1:
49                  if(!dir) {
50                          R_GPIO_PinWrite(GPIO_PORT_E_PIN_7, 0);
51                          R_GPIO_PinWrite(GPIO_PORT_E_PIN_5, 0);
52                  } else if(dir == 1) {
53                          R_GPIO_PinWrite(GPIO_PORT_E_PIN_7, 1);
54                          R_GPIO_PinWrite(GPIO_PORT_E_PIN_5, 0);
55                  } else if(dir == 2) {
56                          R_GPIO_PinWrite(GPIO_PORT_E_PIN_7, 0);
57                          R_GPIO_PinWrite(GPIO_PORT_E_PIN_5, 1);
58                  } else {
59                          R_GPIO_PinWrite(GPIO_PORT_E_PIN_7, 1);
60                          R_GPIO_PinWrite(GPIO_PORT_E_PIN_5, 1);
61                  }
62                  break;
63          }
64  }
65
66
67
```

```
68  void main(void) {
69          /* Initialize CMT for delays: */
70          R_CMT_CreatePeriodic(1000000, timer_cback, (uint32_t*)&CMT0);
71
72          /* Initialize GPIO: */
73          R_GPIO_PinDirectionSet(GPIO_PORT_E_PIN_4, GPIO_DIRECTION_OUTPUT);
74          R_GPIO_PinDirectionSet(GPIO_PORT_E_PIN_6, GPIO_DIRECTION_OUTPUT);
75          R_GPIO_PinDirectionSet(GPIO_PORT_E_PIN_7, GPIO_DIRECTION_OUTPUT);
76          R_GPIO_PinDirectionSet(GPIO_PORT_E_PIN_5, GPIO_DIRECTION_OUTPUT);
77
78          /* Stop motors: */
79          motor_ctrl(MOTOR_LEFT, MOTOR_STOP1);
80          motor_ctrl(MOTOR_RIGHT, MOTOR_STOP1);
81
82          for(;;) {
83                  /* Control left motor every 1 second: */
84                  delay_ms(1000);
85                  motor_ctrl(MOTOR_LEFT, MOTOR_FORW);
86                  delay_ms(1000);
87                  motor_ctrl(MOTOR_LEFT, MOTOR_STOP1);
88                  delay_ms(1000);
89                  motor_ctrl(MOTOR_LEFT, MOTOR_BACK);
90                  delay_ms(1000);
91                  motor_ctrl(MOTOR_LEFT, MOTOR_STOP1);
92
93                  /* Control right motor every 1 second: */
94                  delay_ms(1000);
95                  motor_ctrl(MOTOR_RIGHT, MOTOR_FORW);
96                  delay_ms(1000);
97                  motor_ctrl(MOTOR_RIGHT, MOTOR_STOP1);
98                  delay_ms(1000);
99                  motor_ctrl(MOTOR_RIGHT, MOTOR_BACK);
100                 delay_ms(1000);
101                 motor_ctrl(MOTOR_RIGHT, MOTOR_STOP1);
102         }
103
104         for(;;);
105 }
```

3.B.4) RTC CODE

```
#include <stdio.h>
#include "platform.h"
#include "r_rtc_rx_if.h"
#include "lcd.h"

#define UPDATE_TIME 0

void rtc_callback(void * args) {
        if(*(rtc_cb_evt_t*)args == RTC_EVT_PERIODIC) {
                /* Read and Display Time: */
                tm_t time;
                if(R_RTC_Read(&time, NULL) == RTC_SUCCESS) {
                        char msg1[] = "Time: ";
                        char msg2[20];

                        sprintf(msg2, "%.2d:%.2d:%.2d",time.tm_hour,time.tm_min, time.tm_sec);
                        printf("%s%s\n", msg1,msg2);

                        lcd_display(LCD_LINE1, msg1);
                        lcd_display(LCD_LINE2, msg2);
                }
        }
}

void main(void) {
        /* Initialize LCD: */
        lcd_initialize();
        lcd_clear();

        /* Initialize RTC: */
#if !UPDATE_TIME
        rtc_init();
#endif

        rtc_init_t rtc_init_cfg = {
                .p_callback        = rtc_callback,
                .output_freq       = RTC_OUTPUT_OFF,
                .periodic_freq     = RTC_PERIODIC_2_HZ,
                .periodic_priority = 7,
                .set_time          = UPDATE_TIME
        };

        tm_t init_time = {
                00,  /* Seconds (0-59) */
                04,  /* Minute (0-59) */
                18,  /* Hour (0-23) */
                16,  /* Day of the month (1-31) */
                0,   /* Month (0-11, 0=January) */
                117, /* Year since 1900 */
                1,   /* Day of the week (0-6, 0=Sunday) */
                16,  /* Day of the year (0-365) */
                0    /* Daylight Savings enabled (>0), disabled (=0), or unknown (<0)*/
        };
        printf("Initializing RTC: %d\n", R_RTC_Open(&rtc_init_cfg, &init_time));
        for(;;);
}
```

3.B.5) SD CARD CODE

Will be used for storing log files.

```
#include <stdio.h>
#include "platform.h"
#include "sdcard_inc.h"

void main(void) {
        /* Initialize SD Card slot: */
        if(!R_SdCardSlotInit()) {
                printf("ERROR: No SDCard found on the slot. The program will continue when the SDCard is present ...\n");
                while(!R_SdCardSlotInit());
        }

        printf("SD Card found!\n");

        /* Initialize the rest and mount the SD Card. */
        R_MmcDiskInit();
        R_SdCardDiskMount(SDCARD_LUN);

        DIR dir;
        FILINFO finfo;
        FRESULT res;

        /* Open root directory: */
        res = f_opendir(&dir, "1:");

        /* And navigate through it: */
        while(res == FR_OK) {
                while ((f_readdir(&dir, &finfo) == FR_OK) && finfo.fname[0])  {
                        /* Is entry a directory or a file: */
                        if (finfo.fattrib & AM_DIR) { /* Entry is a directory */
                                printf("\nDirectory: %s", finfo.fname);
                        } else { /* Entry is a file. */
                                printf("\nFile: %s", finfo.fname);
                        }
                }
        }

        for(;;);
}

```

RELEVANT NOTE ABOUT THE CODE

This code has been implemented using a linear approach without RTOS, and only the most basic components were implemented.

The rest of the code that is not found in this section (e.g. line tracker, hall effect sensor, gps, etc) has been implemented purely in RTOS (see Task 6).

3.C) TEST INCLUDING DESIGNING A TESTING STRATEGY

At this stage, all of the components have been tested and are functional. Some of the test results of the core components that contribute the most to the car can be found on the table below.

Test #	Test Point	Type of test	Result
1	Ultrasonic Sensor	Range	According to measurements, the range is most accurate when the sensor is perpendicular to the wall. Also, the sensor succesfully measures distances between 2cm and 400cm.
		Blind spots	The sides and back of the car are the most vulnerable spots. In certain occasions, the car gets stuck on an object which has a shape that does not allow both sensors to get a reading, forcing the user to take control either via Remote Control mode or physically picking up the car.
		Innacuracies and 'false alarms'	The sensor seems to get invalid readings if it tries to detect soft objects, such as cloth or simply the carpet on the floor. This could be caused due to the fact that cloth absorbs pressure waves / sound. This results in the car colliding against the object without being able to detect it.
2	IR Line Tracker	Innacurate detection	Innacuracies have been found while trying to track a line on a surface that has a bright color but not completely white. On certain occasions, the sensor wouldn't detect a reflection off the bright (color bright) surface, for example on a light brown table. It would only detect if the distance between the table and the sensor was greatly reduced.
		Line thickness	The car seems to be very capable of following lines of any thickness. There is an exception, which is when the white background is not thick enough (which surrounds the black line). This means when the car turns to one of the directions when it finds the end of the black line, if the white line is not thick enough, the car might find another line and / or the floor itself.

3	Motor	Minimum RPM required	Due to the weight, momentum and wheel grip/friction of the car, the minimum required duty cycle value for the PWM is 14.50% (frequency is 490 Hz).
		Durability	The gears inside the motors seem to wear down quite quickly if the car is being used at the maximum speed. Because of this, the allowed speed has been capped to a safe level on the software. It is also believed that the weight of the car contributes to this problem.
4	Servo	Power consumption	The two servos tend to consume too much current. For this reason, it is advised not to control the servos directly through the Remote Control mode.
		Maximum angle	The servos seem to have a lower angle of rotation than expected. Instead of 180º, they rotate approximately 150º, which isn't critical.
5	Bluetooth	Data transmission rate	The Bluetooth module has been heavily tested continuously since the beginning of the project, and the results show that this module is extremely responsive and more than perfect for this application.

Finally, the strategy adopted for the software development of this project follows a very simple algorithm of trial and error. Develop, test, fix, test, add/commit code, and develop again. The strategy is best described by analysing the diagram below.

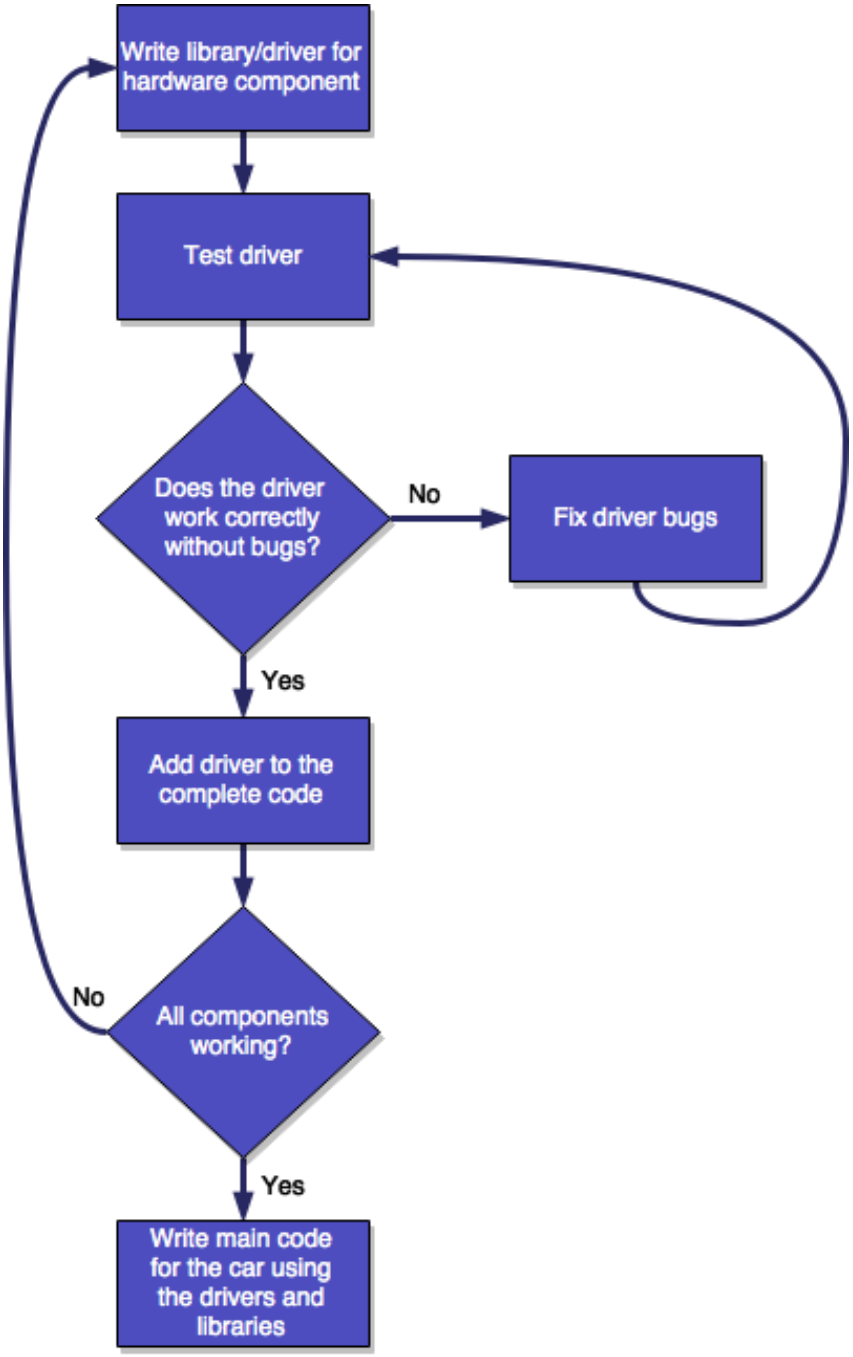


Figure 85 - Testing strategy algorithm

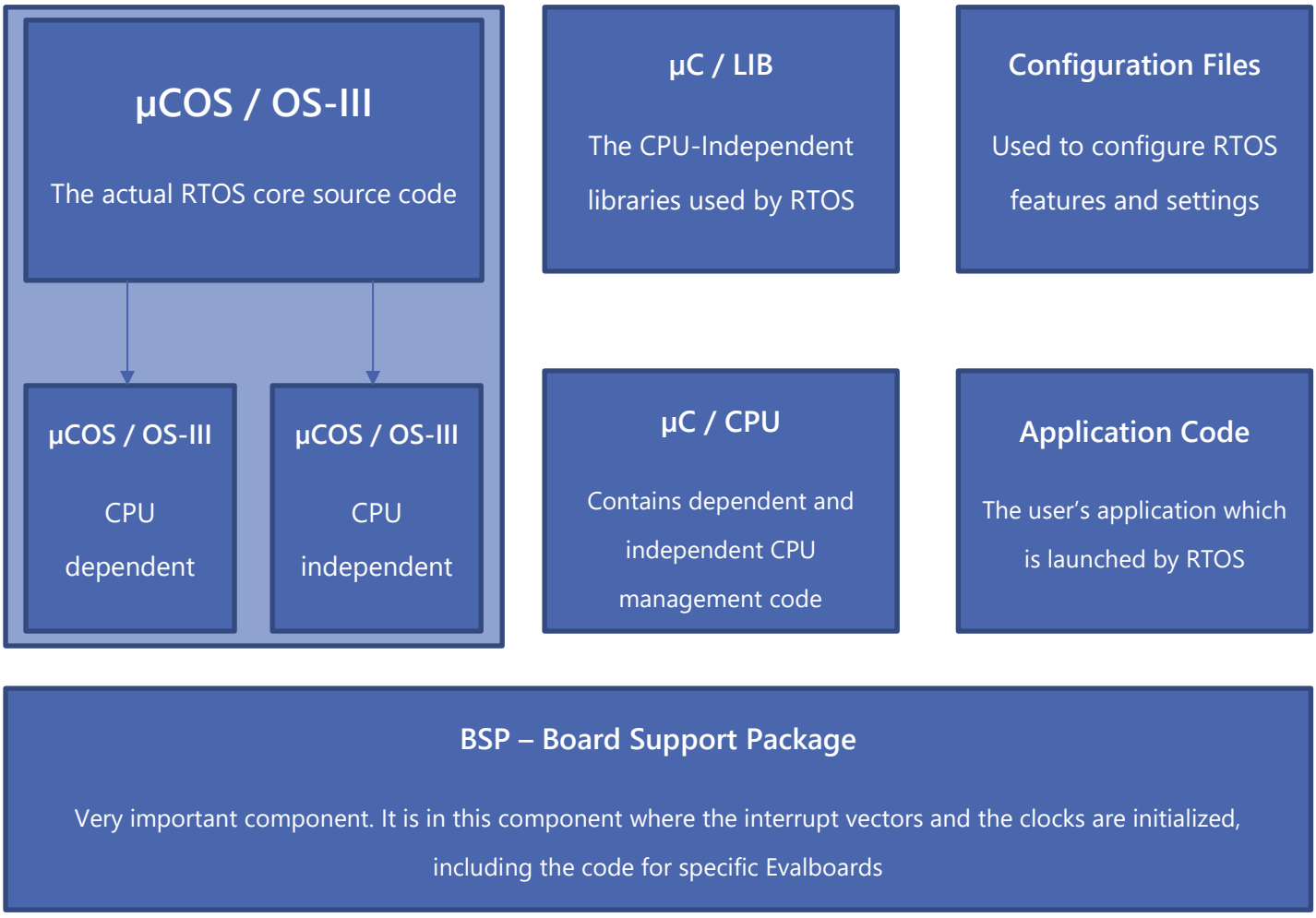
The last step, which includes the main code of the program, also expands downwards and follows the exact same algorithm as the libraries and drivers.

TASK 4 – BACKGROUND RESEARCH & DESIGN WITH RTOS µCOSIII

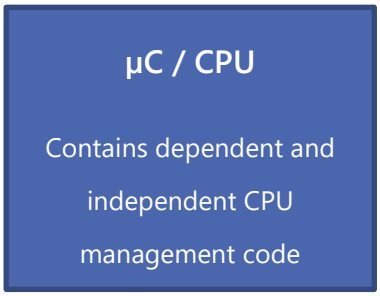
4.A) PORTING FEATURES

In this section the RTOS (Real Time Operating Sistem) components will be identified and organised with a list and a diagram.

First, the RTOS is architecturally designed with modularity in mind.



4.A.1) CPU COMPONENTS



This component contains the following C source files:

1) **cpu.h** – Contains Renesas' RX63N specific code. It defines memory stack configurations and other memory-related settings.
2) **cpu_def.h** – Declares CPU-Independent macros related to stack growth direction and position
3) **cpu_c.c**
4) **cpu_a.asm**
5) **cpu_core.c** – Initializes and manages critical CPU features, which were declared and defined in cpu.h. These features include: unrecoverable software interrupt exception and timestamp timers.
6) **cpu_core.h** – This file contains cpu error codes, function prototypes and other cpu-related macros.

4.A.2) LIBRARIES COMPONENTS

µC / LIB

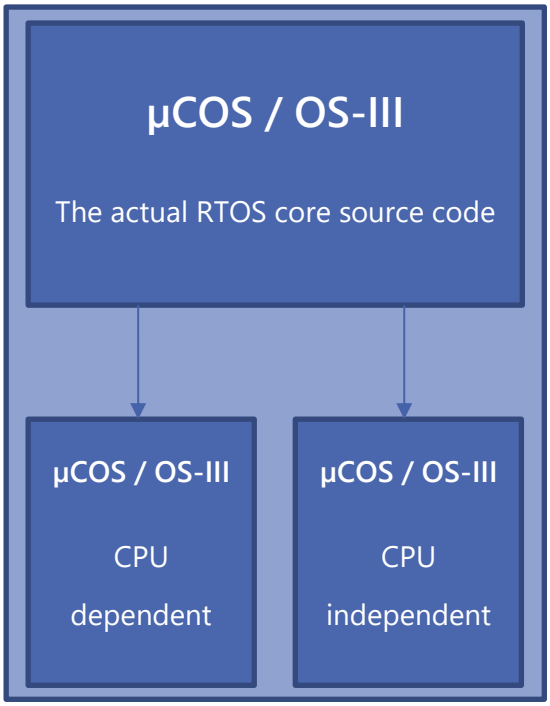
The CPU-Independent libraries used by RTOS

Just like the Standard ANSI-C library, the libraries used in RTOS were designed to be used in any platform/target. They are completely CPU-Independent.

It is composed of:

1) **lib_ascii.c** – Contains ASCII/Character libraries
2) **lib_ascii.h** - Contains ASCII/Character libraries' declarations/prototypes
3) **lib_def.h** – Miscellaneous declarations can be found here, such as **DEF_TRUE** and **DEF_FALSE**.
4) **lib_math.c** – Handles functions used for mathematical calculations
5) **lib_match.h** - Handles functions used for mathematical calculations
6) **lib_mem_a.asm** – Declares memory-related functions, such as Mem_Clr, Mem_Set and Mem_Copy.
7) **lib_mem.c** - Declares memory-related functions
8) **lib_mem.h** - Declares memory-related functions
9) **lib_str.c** – String related functions are found here, for example **Str_Copy** and **Str_Cat**.
10) **lib_str.h** - String related functions are found here

4.A.3) RTOS CORE COMPONENTS



This is by far the most important component. This is what makes all this source code an actual RTOS. All the **scheduling**, **pre-emption**, **task creation**, **interrupting**, **semaphores**, **mutexes** and **task messages**, can be found here.

In this component, we find the files (all CPU-independent):

1) os_cfg_app.c
2) os_type.h
3) os_core.c
4) os_dbg.c
5) os_flag.c
6) os_int.c
7) os_mem.c
8) os_msg.c

9) os_mutex.c
10) os_pend_multi.c
11) os_prio.c
12) os_q.c
13) os_sem.c
14) os_stat.c
15) os_task.c
16) os_tick.c
17) os_time.c
18) os_tmr.c
19) os_var.c
20) os.h

Renesas' RX63N dependent RTOS (Ported) code:

1) os_cpu_a.src
2) os_cpu_c.c
3) os_cpu.h

4.B) BOARD SUPPORT PACKAGE (BSP) SOFTWARE FOR SPECIFIC DEVELOPMENT BOARDS

BSP – Board Support Package

Very important component. It is in this component where the interrupt vectors and the clocks are initialized, including the code for specific Evalboards

The BSP component contains software designed specifically for certain microcontrollers and certain evaluation boards, for example, the YRDKRX63N.

It contains the CPU's interrupt vectors and clock initializers. It also includes evalboard peripheral drivers, such as LEDs, Serial (UART), LCD (Glyph), Accelerometer, Temperature sensor, Networking, etc.

Because the BSP component varies from board to board, only some of the most relevant files will be listed:

1) bsp.c – Initializes interrupts, clocks, etc.
2) bsp.h – Contains function prototype '`void BSP_Init (void);`'
3) bsp_adt7420.c/h – Driver for the temperature sensor
4) bsp_adxl345.c/h – Driver for the accelerometer

5) bsp_glcd.c/h – Driver for the LCD screen
6) bsp_ser.c/h – Driver for the Serial Communication Port
7) bsp_tick_a.asm – The RTOS' tick interrupt handler can be found here
8) bsp_tick_c.c - The RTOS' tick is initialised here, by starting the CMT0 component as the timer that pre-empts the kernel.

4.C) GENERAL FEATURES FROM THE RX63N MICROCONTROLLER

4.C.1) INITIALIZATIONS

In order to initialize the Micrium RTOS kernel, we can use the functions:

```
void  OSInit (OS_ERR  *p_err);
void  BSP_Init (void);
void  CPU_Init (void);
void  Mem_Init (void);
```

We call these functions on the main function as soon as the program starts running.

4.C.2) CREATING TASKS

In order to create a task, we can use the following function prototype:

```
void  OSTaskCreate (OS_TCB        *p_tcb,
                    CPU_CHAR      *p_name,
                    OS_TASK_PTR    p_task,
                    void          *p_arg,
                    OS_PRIO        prio,
                    CPU_STK       *p_stk_base,
                    CPU_STK_SIZE   stk_limit,
                    CPU_STK_SIZE   stk_size,
                    OS_MSG_QTY     q_size,
                    OS_TICK        time_quanta,
                    void          *p_ext,
                    OS_OPT         opt,
                    OS_ERR        *p_err);
```

- **Parameter 1 (p_tcb)** – A pointer to the task control block. This pointer is the handle to the entire task and it contains vital information about all aspects of the task.
- **Parameter 2 (p_name)** - A pointer / string of the task name.
- **Parameter 3 (p_task)** – The address of the function which will be run after this method call is completed. This function is what we call the actual task.

- **Parameter 4 (p_arg)** – This pointer will hold the address of the argument that will be passed onto the task being created. This pointer will be held in the task's stack.
- **Parameter 5 (prio)** – The priority of the task being 'spawned'.
- **Parameter 6 (p_stk_base)** – The starting address of the task's stack.
- **Parameter 7 (stk_limit)** – The maximum limit of the size of the stack.
- **Parameter 8 (q_size)** – Maximum number of messages allowed for this task.
- **Parameter 9 (time_quanta)** – Resolution of the preemption frequency (how fast will the task preempt).
- **Parameter 10 (p_ext)** – Pointer to a TCB extension.
- **Parameter 11 (opt)** – This variable holds task options, such as stack memory clearing on task startup and saving floating point registers on context switch.
- **Paratemer 12 (p_err)** – Holds the return value of the function call.

4.C.3) SCHEDULING AND RESOURCE MANAGEMENT

Two mechanisms are used by the RTOS' scheduling algorithm.

1) Pre-emption – On every timer interrupt, switch to next task
2) Round Robin - Switch to next task using the round robin algorithm (each task has a certain TTL (time to live) associated to it)

These functions are used in the task switching process:

```
void  OSSched (void);
void  OSSchedLock (OS_ERR  *p_err);
void  OSSchedUnlock (OS_ERR  *p_err);
void  OSTimeDlyHMSM (CPU_INT16U   hours,
                     CPU_INT16U   minutes,
                     CPU_INT16U   seconds,
                     CPU_INT32U   milli,
                     OS_OPT       opt,
                     OS_ERR      *p_err);
```

In order to avoid resource conflict between tasks, eight software algorithms are commonly used:

1) Reentrancy
2) Interrupt disabling
3) Locks
4) Mutexes
5) Semaphores
6) Queues
7) Messages
8) Pipes

4.C.4) TASKS COMMUNICATIONS

Task communication can be achieved by using the API for Messages already present on the RTOS.

Some of the functions are:

```
// Create a message queue
void  OSQCreate (OS_Q         *p_q,
                 CPU_CHAR     *p_name,
                 OS_MSG_QTY    max_qty,
                 OS_ERR       *p_err);

// Wait for a message
void  *OSQPend (OS_Q          *p_q,
                OS_TICK        timeout,
                OS_OPT         opt,
                OS_MSG_SIZE   *p_msg_size,
                CPU_TS        *p_ts,
                OS_ERR        *p_err);

// Send a message
void  OSQPost (OS_Q          *p_q,
               void          *p_void,
               OS_MSG_SIZE    msg_size,
               OS_OPT         opt,
               OS_ERR        *p_err);
```

These functions need to be used in parallel between two or more tasks. For instance, a task shouldn't call **OSQPend** if no other task has been spawned because this function call blocks the task's thread.

TASK 5 – COMPARISON BETWEEN THE LINEAR PROGRAM AND RTOS

5.A) DISADVANTAGES OF THE LINEAR APPROACH

The biggest and clearest disadvantage of using the linear approach while creating an application for embedded systems is the fact that only one task is running at any given time. This is a huge problem because the responsiveness of the system is one of the most critical points while evaluating the quality of an embedded system.

This problem occurs because the system only has 1 core and 1 thread. In order to obtain multitasking, even on modern multi-core systems, a kernel must exist in any case. Without a kernel, it's impossible to have two tasks running at the same time.

Not only is using the normal linear approach a big disadvantage in regards to the multitasking aspect, by not using RTOS we lose a lot of useful features already provided by the big companies, such as modular BSP packages, File Systems, Net Stacks and USB.

Also, in some cases where the code gets too large, not using an RTOS means the programmer will have to develop all the system management, i.e. Memory and CPU management code, System Calls, etc.

All of the management code is already implemented in RTOS and this is the main reason why choosing the multitasked alternative is the best choice – because there is no need to reinvent the wheel.

To further underline the point of the linear approach being a bad alternative, the URS has been reanalysed taking into mind the advantages that would be obtained by using RTOS.

Original URS	Analysis - Comprehension
1. USW is thinking of automating its mail delivery to various buildings in its Trefforest campus. Such a system would involve a vehicle that will be able to navigate safely and autonomously, through a route, recognizing its position and avoiding any hazards in its path. For the purpose of this project, a scaled down prototype should be produced that should consist in its first prototype: • Mechanical components to be determined • Electrical components that include a Power Unit capable to drive the mechanical parts and provide enough juice for the rest of the system. • Electronic components built around a command and control unit-CCU that will allow the vehicle to scan its environment to determine the path to follow on a point to point strategy.	• A self-driving vehicle that carries a package/load for delivery throughout the USW Campus. • This vehicle must learn and recognize the path in order to prevent unwanted collisions. • **By introducing RTOS, the car will be able to learn, recognize the path and avoid the obstacle all at the same time. We obtain responsiveness.** • Three major components are present in this design: the mechanical parts, the electrical components and the software which controls both mechanical and electrical parts.

2. The prototype is expected to be programmable so it can serve many purposes. For this, any future enhancements would ideally be accommodated in the original design. For example, it is viewed that many of these units might be built and used jointly for a purpose that might arise in the future. So, communication between each vehicle or via a central command should be envisaged.	• The prototype needs to be designed as a multi-purposed vehicle. • **RTOS might be ideal for adding new functionality to the car, since it was designed modularly (through BSP packages).** • It will require a communication link with other vehicles. • These vehicles could be connected to one central server (centralized communication). • The alternative option would be to implement a mesh network and use that instead for communication. The user/programmer would treat any mesh node as the entry point to the network (decentralised communication). • It would also be a good idea to introduce the feature of wireless re-programming.
3. Any other features that are omitted or not obviously stated in this document that will enhance the vehicle might be incorporated prior to an approval based on the planning (timescale & budget).	• More features may be added in the future, which will be liable for approval due to the project timescale and budget.

4. The function of the system is to monitor and report on the state of each leg of the path. Monitoring consists of scanning the environment, checking for any obstacles in view of avoiding them. Reporting consists of allowing processor to interrogate the sensors and working the location and the distance of any obstacles over a period of time the format of a scanning is: Sensor no. Sensor Location. Distance This consists of a sensor number, the location of the sensor which originated the reading, and, finally, the distance of the obstacle. Each reading is issued every second.	• The system/CPU will periodically request/poll and store the data from the sensors used in the mechanism for locomotion which will indicate possible collisions and will offer a way to prevent them in the future. • This data will be packed in memory RAM and stored with the format: Sensor no. Sensor Location. Distance on any non-volatile memory storage, such as Flash memory, EEPROM or SD Card. • **File systems are much easier to implement on the system's algorithm while using RTOS.**
5. On receiving a reading that is outside safety limits the system should take appropriate actions to steer the AV safely and securely away from the obstacle(s).	• Whenever the sensor readings indicate an imminent collision, the microcontroller will control the motors to steer away from the obstacle. • A good feature that could be implemented would be to scan the environment first and decide what direction to turn next. • Also, the use of GPS and machine learning algorithms could help the software's algorithm in pre-recognising static obstacles before the vehicle even gets close to them. • **The sensor readings would be more accurate and the system would be more responsive if an RTOS was being used, since all the sensors' data would be collected at the same time.**

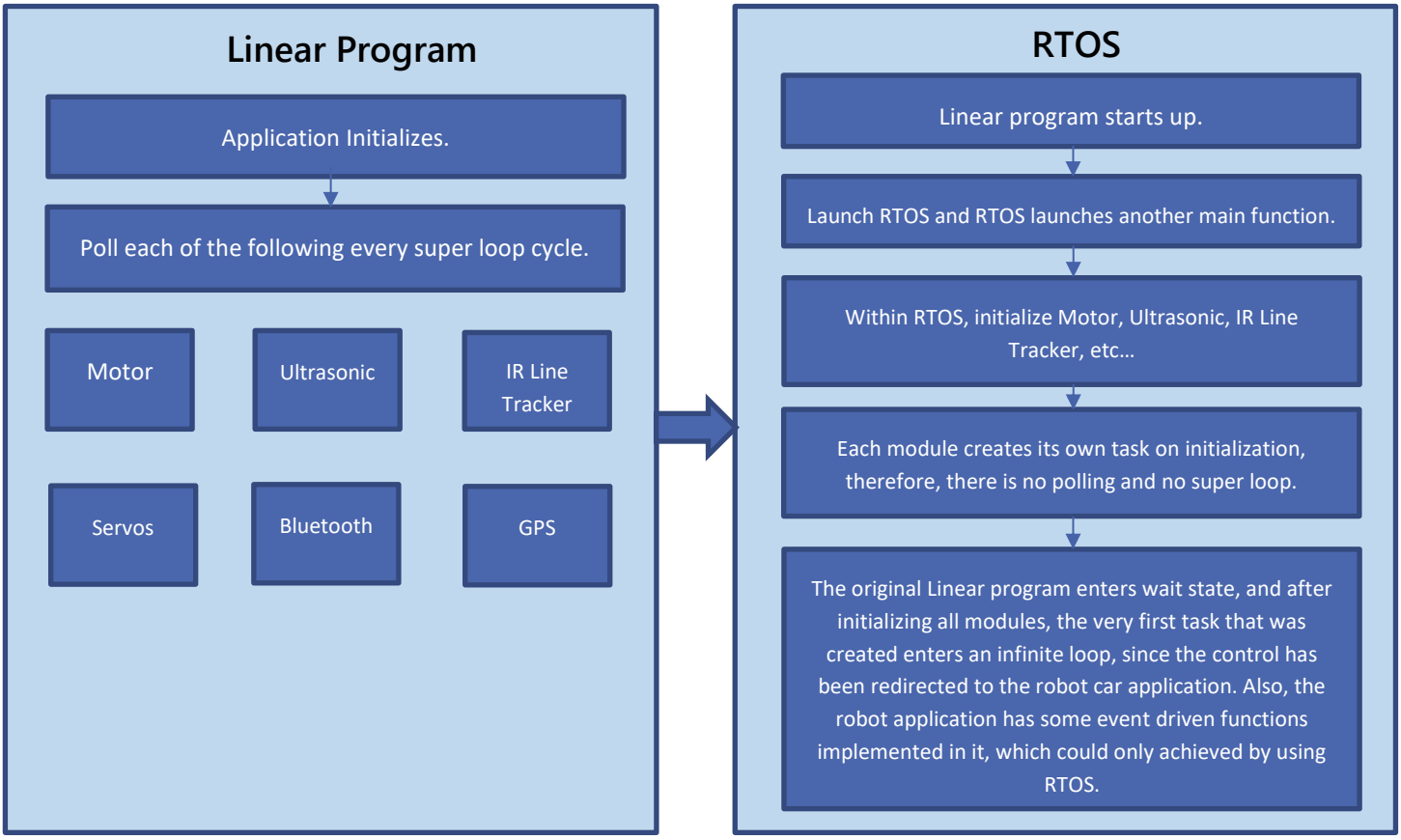
6. The system should allow the operator to check on the state of operation of each sensor over the past 48 hours via an interface display that should be as flexible as possible.	• The machine should record every sensor data for a period of at least 48 hours so that the operator can examine the occurrences throughout that time. Also, the vehicle should include an interface which allows the operator to easily and quickly analyse the data. • Multiple interfaces could be used to visualise this data, such as an LCD screen, the Wi-Fi link or even Bluetooth. • **RTOS was designed to execute tasks deterministically. Scheduling an event to manage the data every 48 hours so that the operator can analyse it would be one of the features of RTOS.**
7. The system response should be the maximum possible achievable	• The vehicle should act as quick as possible whenever an obstacle is encountered, in order to prevent catastrophic failure. • **This could be achieved by using an RTOS.**
8. The system should be programmed in C. All file handling should be performed by calls on operating system routines. The fixed price for the contract will be £20,000 which should take into account in its conception the suggestions in Appendix-A.	• **The software must be written using C language with the help of Real Time Operating Systems (RTOS) which will allow the handling of log data files.** • **Not only is RTOS good for using software stacks (TCP/IP, File systems, etc…), it is most importantly used for time-constrained applications.** **This is the main reason why this project is using RTOS.** • The allocated budget for this project is £20,000

5.B) ADVANTAGES OF INTRODUCING RTOS

As was already described on both tasks 4) and 5.A), the biggest advantages of introducing RTOS are:

1) Multiple tasks can be run at the same time, thus achieving higher throughput / responsiveness.
2) RTOS provides system functions that allows the user to investigate the state of the internal system (which is something that is non-existent in the linear program). For example, it helps the programmer to debug the amount of memory RAM left, or how much CPU processing power is currently being used.
3) The RTOS is very easy to use. The only requirement is to download the source code that was described on task 4) and open the dummy project that comes with it.
4) A lot of libraries and drivers are already provided by Micrium, i.e. USB drivers, Network drivers and File systems.
5) It's very simple to add new BSP packages.
6) The documentation is extremely clear and reader friendly.
7) Micrium µCOSIII is an open source project, which means any one can add new functionality to the original source code, as long as credit is given to the original author.

5.C) PLANNED CHANGES FROM LINEAR TO RTOS



TASK 6 – IMPLEMENTING THE CHANGES (CODE IN RTOS)

The very first thing the code must execute is to initialize the RTOS kernel and to launch the very first task, which is named kmain.

```
1   #include "platform.h"
2   #include <stdio.h>
3   #include <rtos_inc.h>
4
5   #define KMAIN_STACKSIZE 128
6   static  CPU_STK mainStk[KMAIN_STACKSIZE];
7   static  OS_TCB  mainTCB;
8
9   extern void kmain_early(void);
10  extern void kmain(void * p_arg);
11
12  void main(void) {
13          OS_ERR err;
14
15          // We might want to execute something before launching the RTOS.
16          // That code will be inside this function call
17          kmain_early();
18
19          // Disable interrupts
20          R_BSP_InterruptsDisable();
21
22          // Initialize RTOS
23          OSInit(&err);
24
25          // Initialize the microcontroller's components
26          BSP_Init();
27          CPU_Init();
28          Mem_Init();
29
30          // Create the first task, the kmain (Kernel Main)
31          OSTaskCreate((OS_TCB     *)&mainTCB,
32                                   (CPU_CHAR   *)"Main",
33                                   (OS_TASK_PTR ) kmain,
34                                   (void       *) 0,
35                                   (OS_PRIO     ) 10,
36                                   (CPU_STK    *)&mainStk[0],
37                                   (CPU_STK_SIZE) KMAIN_STACKSIZE / 10u,
38                                   (CPU_STK_SIZE) KMAIN_STACKSIZE,
39                                   (OS_MSG_QTY  ) 0u,
40                                   (OS_TICK     ) 0u,
41                                   (void       *) 0,
42                                   (OS_OPT      )(OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
43                                   (OS_ERR     *)&err);
44
45          // And finally, launch the RTOS!
46          OSStart(&err);
47
48          // The program should never reach this point
49          while(1);
50  }
```

We can clearly observe from this code that the task is being created with the function `OSTaskCreate(...)`. This function launches the kmain task, which will initialize all the different drivers used in the robot car application.

The drivers being initialized (in the right order) are:

1) L298N H-Bridge (for the motors)
2) Bluetooth
3) GPS
4) Infrared receiver (not the line tracker)
5) The two ultrasonic sensors
6) Accelerometer
7) Infrared line tracker sensor
8) Hall effect sensor
9) Two SG90 servo motors
10) A status RGB LED, used to signal the user the status of the program (error, information, ok, warning, etc.)

After initializing these components, the actual robot application is launched.

The entire code for the kmain.c file is shown below:

```
/*****************************************************************/
/*                                                               */
/*      PROJECT NAME :  Smart Car                                */
/*      FILE         :  kmain.c                                  */
/*      DESCRIPTION  :  Main Program                             */
/*      CPU SERIES   :  RX600                                    */
/*      CPU TYPE     :  RX63N                                    */
/*                                                               */
/*      This file is generated by e2 studio.                     */
/*                                                               */
/*****************************************************************/

/***********************************/
/** Author: Miguel Santos 14031329 **/
/***********************************/

#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <platform.h>
#include <globals.h>

/*********************************************************************/
/** Declare the main hardware components used in this project: **/
/*********************************************************************/
suart_t    * module_bluetooth;
suart_t    * module_gps;
irrem_t    * module_infrared;
usonic_t   * module_usonic1;
usonic_t   * module_usonic2;
dhall_t    * module_hallsensor;
ltracker_t * module_ltracker;
sled_t     * module_status_led;

/*************************************/
/** This task is running at 101.8KHz: **/
/*************************************/
void poller(void) {
       suart_poll();  /* Poll the Bluetooth and GPS   */
       usonic_poll(); /* Poll the Ultrasonic sensors */
       spwm_poll();   /* Poll the SPWM library (used by the Status LED) */
       irrem_poll();  /* Poll the Infrared receiver   */
}

void bluetooth_on_rx(uint8_t * buff, uint32_t bufflen) {
       robot_on_bt(buff, bufflen); /* Redirect the Bluetooth reception packet to the robot application */
}

void gps_on_rx(uint8_t * buff, uint32_t bufflen) {
       /* Not using the GPS due to available development time restrictions */
}

void infrared_on_rx(uint16_t * buff, uint32_t bufflen) {
       robot_on_ir(*buff); /* Redirect the Infrared reception packet to the robot application */
}

void dhall_on_detection(dhall_t * handle) {
       suart_printf(module_bluetooth, "\n\rHALL DETECTED\n");
}


```

```
62  void kmain_early(void) {
63          /* This function gets called before the RTOS starts running.
64           * It was implemented only because this project can also serve as a template
65           * RTOS project to be used in other projects.
66           * We won't do anything here for now... */
67  }
68
69  void kmain(void * args) {
70          /* Initialize the motor driver for the H-Bridge L298N: */
71          l298n_init();
72
73          /* Initialize all Communication and Navigation channels: */
74          module_bluetooth  = suart_init(SUART_MAX_BAUD, BTOOTH_RX,  BTOOTH_TX,          bluetooth_on_rx, 64, 30);
75          module_gps        = suart_init(SUART_MAX_BAUD, GPS_RX,     (gpio_port_pin_t)0, gps_on_rx,       0,  30);
76          module_infrared   = irrem_init(IRREM_RX, infrared_on_rx, 1);
77
78          /* Initialize Ultrasonic sensor driver for two channels: */
79          module_usonic1    = usonic_init(USONIC1_ECHO, USONIC1_TRIG, 1);
80          module_usonic2    = usonic_init(USONIC2_ECHO, USONIC2_TRIG, 1);
81
82          /* Initialize Accelerometer: */
83          accel_init();
84
85          /* Initialize Tracker sensor driver: */
86          module_ltracker   = ltracker_init(TRACKER_LEFT, TRACKER_CENTER, TRACKER_RIGHT, NULL_CBACK, LTRACKER_MODE_LINE_BLACK);
87
88          /* Initialize Hall Effect sensor driver: */
89          module_hallsensor = dhall_init(DHALL_SIGNAL, dhall_on_detection, DHALL_MODE_ONESHOT);
90
91          /* Initialize both Servo channels: */
92          sg90_init_channels();
93
94          /* Initialize Status LED: */
95          module_status_led = status_led_init(SLED1_RED, SLED1_GREEN, SLED1_BLUE);
96
97          /* Start polling! */
98          OS_FastTickRegister(poller, 0);
99
100         /* Launch User Application: */
101         spawn_task("robot_task", start_robot);
102
103         /* Do nothing for this task: */
104         while(1)
105                 rtos_delay(1000);
106 }
107
108 void kmain_cleanup(void) {
109         /*
110          * Nothing to cleanup for now...
111          * (Reason: the RTOS never lets the program reach this point)
112          */
113 }
```

As can be seen, each hardware component is treated as a complex structure that is independent of any other component. In other words, a type struct was declared for every component as a handle pointer in order to improve resource management and avoid what is called spaghetti code.

Each module manages its own resources and has its own task. Those tasks are created the following way (taking the line tracker module as an example):

```
/* Create RTOS task for polling the 3 signal pins: */
spawn_task_args("ltracker_poller", ltracker_poller, channel_ptr);
```

Inside the function `spawn_task_args` we find:

```
void spawn_task_args(char * proc_name, OS_TASK_PTR task_ptr, void * args) {
	spawn_task_args_quanta(proc_name, task_ptr, args, 0);
}
```

And inside the function `spawn_task_args_quanta` we find:

```
#define PROCESS_STACKSIZE       128
#define PROCESS_DEFAULTPRIORITY 5
#define PROCESS_MAXCOUNT        20

typedef struct {
	OS_TCB       tcb;
	CPU_STK_SIZE stacksize;
	CPU_STK      stack[PROCESS_STACKSIZE];
} process_t;

process_t procs[PROCESS_MAXCOUNT];
uint8_t   proc_i = 0;

void spawn_task_args_quanta(char * proc_name, OS_TASK_PTR task_ptr, void * args, uint16_t quanta) {
	OS_ERR err;

	if(proc_i >= PROCESS_MAXCOUNT) return;

	process_t * proc = &procs[proc_i++];
	proc->stacksize = PROCESS_STACKSIZE;

	OSTaskCreate((OS_TCB     *) &proc->tcb,
		     (CPU_CHAR   *) proc_name,
		     (OS_TASK_PTR ) task_ptr,
		     (void       *) args,
		     (OS_PRIO     ) PROCESS_DEFAULTPRIORITY,
		     (CPU_STK    *) &proc->stack[0],
		     (CPU_STK_SIZE) proc->stacksize / 10u,
		     (CPU_STK_SIZE) proc->stacksize,
		     (OS_MSG_QTY  ) 0u,
		     (OS_TICK     ) quanta,
		     (void       *) 0,
		     (OS_OPT      ) (OS_OPT_TASK_STK_CHK | OS_OPT_TASK_STK_CLR),
		     (OS_ERR     *)&err);
}
```

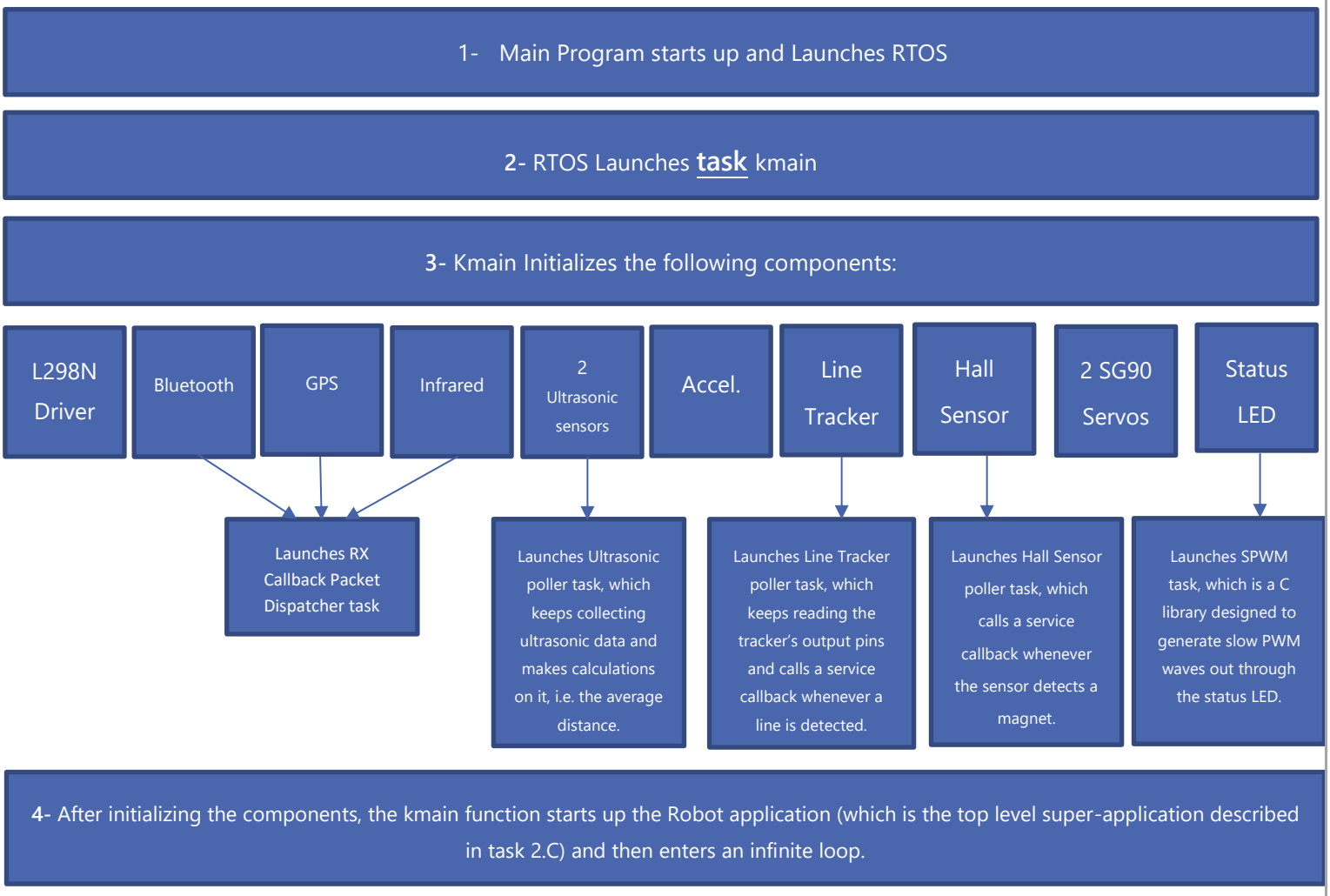
This is how tasks are created. We simply call one of these functions:

```
1   void spawn_task_args_quanta(char * proc_name, OS_TASK_PTR task_ptr, void * args, uint16_t quanta);
2   void spawn_task_args(char * proc_name, OS_TASK_PTR task_ptr, void * args);
3   void spawn_task_quanta(char * proc_name, OS_TASK_PTR task_ptr, uint16_t quanta);
4   void spawn_task(char * proc_name, OS_TASK_PTR task_ptr);
```

And we don't need to worry about calling `OSTaskCreate` every time we want to create a new task.

This was done because a lot of tasks are being created for this project and it is much easier to call one single function instead of declaring the static stack, the TCB variable and all of the other required arguments to create a new task. The only trade-off is that every task gets the same priority, stack size and time quanta.

Looking at the system at a high level, we can see how the tasks are organized by using a diagram.



In terms of inter-task communication, no RTOS messages are being used. This is because the communication system that is being used is event driven. For instance, the robot application task will always be updating the sub-program's logic on a loop, but it will not be checking for any Bluetooth communications on every loop cycle. Instead, a callback is being used to handle the Bluetooth data reception, which runs in parallel with the robot application. The callback function will then decide if the robot application should react to this reception.

In order to understand this explanation even further, let's examine closely the robot task creation on the kmain function:

```
1   /* File: kmain.c. Line: 101 */
2   /* Launch User Application: */
    spawn_task("robot_task", start_robot);
```

In this line we are spawning/creating the task that launches the robot application. Inside the function `start_robot` we find:

```
1   void start_robot(void * args) {
2           /* Set initial motor speed: */
3           l298n_ctrl(L298N_CHANNEL_BOTH_SIDES, DIR_FORW, L298N_MOTOR_MIN_SPEED);
4           l298n_stop();
5
6           while(1) {
7                   /* Update the mode variable of the robot every cycle: */
8                   robot_update_mode_logic();
9
10                  /* Update Robot application logic: */
11                  switch(robot_mode) {
12                          case ROBOT_MODE_IDLE: /* Do nothing */ break;
13                          case ROBOT_MODE_AUTONOMOUS:
14                                  robot_auton_update();
15                                  break;
16                          case ROBOT_MODE_LINEFOLLOWER:
17                                  if(!robot_lfollower_update()) {
18                                          /* Something went wrong while updating the logic of this mode.
19                                           * Perhaps the car found a very weird line and didn't manage to
20                                           * understand its shape.
21                                           * In this case, we must just switch back temporarily to autonomous and only
22                                           * allow the car to return to line follower mode until a valid line has been found.
23                                           * If a valid line hasn't been found for a certain amount of time (timeout), we will
24                                           * just ignore that the car found a bad line, and will default into normal autonomous mode */
25
26                                          robot_switch_mode(ROBOT_MODE_AUTONOMOUS);
27                                          robot_ltracker_found_weird_line = true;
28                                          status_led_update(module_status_led, STATUS_NOTE_WARNING, STATUS_ANIM_FASTBLINK);
29                                  }
30                                  break;
31                          case ROBOT_MODE_RCCAR: /* This mode is event driven. Nothing to update */ break;
32                          default: break;
33                  }
34
35
```

```
36  #if ROBOT_DEBUG_ENABLE == (1)
37                  robot_debug();
38  #endif
39
40                  rtos_delay(ROBOT_UPDATE_DELTA);
41          }
42  }
```

If we look closely, we can see that the variable responsible for controlling the mode of the algorithm is the `robot_mode` enum on the line number 11.

```
1   /* From file robot.h Line 20: */
2   enum ROBOT_MODES {
3           ROBOT_MODE_IDLE,
4           ROBOT_MODE_AUTONOMOUS,   /* The car scans the environment avoiding obstacles using the two ultrasonic sensors */
5           ROBOT_MODE_LINEFOLLOWER, /* The car follows a black line on a white background */
6           ROBOT_MODE_RCCAR         /* Remote Control Car Mode */
7   };
8
9   /* From file robot.c Line 13: */
10  enum ROBOT_MODES robot_mode = ROBOT_DEFAULT_MODE;
```

So, in order for this variable to change from one mode to another, a separate task will need to communicate with the robot application task via callback.

```
1   void robot_on_bt(uint8_t * buff, uint32_t bufflen) {
2
3   #if ROBOT_DEBUG_ENABLE == (1)
4           /* Echo back the received data: */
5           suart_tx_buff(module_bluetooth, buff, bufflen);
6   #endif
7
8           uint8_t key = decode_bt_cmd(buff, bufflen);
9
10          if((robot_mode == ROBOT_MODE_AUTONOMOUS || robot_mode == ROBOT_MODE_LINEFOLLOWER) && key == NONE)
11                  return;
12
13          if(key != SELECT && key != START) {
14                  if(robot_mode != ROBOT_MODE_RCCAR) {
15                          robot_auton_restart();
16                          robot_lfollower_restart();
17                          status_led_update(module_status_led, STATUS_NOTE_OK, STATUS_ANIM_SLOWFADE);
18                          robot_mode = ROBOT_MODE_RCCAR;
19                  }
20          } else {
21                  robot_mode = ROBOT_MODE_AUTONOMOUS;
22                  if(key == START) {
23                          robot_mode_enable_line_follower = true;
24                          robot_mode_switch_latch = false;
25                  } else {
26                          robot_mode_enable_line_follower = false;
27                  }
28                  robot_update_mode_logic();
29          }
30
31          if(robot_mode == ROBOT_MODE_RCCAR)
32                  robot_rcc_on_bt((enum BT_CODES)key);
33  }
```

Here we can see the callback change the robot mode variable on lines 18 and 21.

This is how inter-task communication is being done.

All in all, the role of RTOS in this application is to:

1) Launch the kernel main function
2) Provide debugging functionality and systems calls
3) Manage the memory (both stack and heap)
4) To easily create tasks at will
5) To delay and reschedule the many components that were initialized by the kernel main function

CONCLUSION

It is at this stage when a conclusion must be drawn from the work developed by the student on the previous six tasks in order to contemplate and evaluate the level of understanding and knowledge acquired.

First, after analysing the difficulty and the flexibility that such a project of this nature can have, it is possible to conclude that a very good understanding of an URS is the very first fundamental step that an engineer should adopt while starting a new project.

In other words, a bad URS, or a misunderstanding between the engineer/student and the client/lecturer, would lead to a bad project output.

The extraction of nouns and verbs allowed the interpretation of the URS to be effective and precise.

Secondly, even after realising the fact that the project was missing critical components for a short period of time due to practical constraints (university technician was busy), it was still possible to develop many

features of the car even without the PCB, which demonstrates the flexibility of both the project type and the student.

Finally, after reading the sample code provided on tasks 3.B, 4.C and 6, it's easy to realize how simple it is to create an application for a certain hardware component by using the FIT modules and RTOS.

The fact that the code for the LCD, RTC, Timers, GPIO and RTOS was already provided by the companies Renesas and Micrium proves just how much both companies care about their customer base.

This type of support is excellent because it promotes the growth of the engineer and hobbyist community.

All thanks to these companies by making all of their source code available free of charge to anyone.

REFERENCES

CONRAD, J. M., & DEAN, A. G. (2013). *Embedded Systems: An Introduction using the Renesas RX63N Microcontroller.* Weston, FL 33326, USA: Micrium Press.

Renesas. (2014). *RX63N Group, RX631 Group User's Manual: Hardware.*

APPENDICES

1- Renesas YRDKRX63N Evalboard: https://www.renesas.com/en-eu/products/software-tools/boards-and-kits/renesas-demonstration-kits/yrdkrx63n-for-rx63n.html

2- Micrium Software: https://www.micrium.com/downloadcenter/download-results/?searchterm=mi-rx63n&supported=true

3- Github page hosting this project: https://github.com/miguelangelo78/Robotty