Areg Babayan

The Cyborg v3.0

Finalizing the Foundation for an NTNU Mascot

Master's thesis in Cybernetics and Robotics
Supervisor: Associate Professor Sverre Hendseth,
PhD Candidate Martinius Knudsen
June 2019

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of Science and Technology

Areg Babayan

The Cyborg v3.0

Finalizing the Foundation for an NTNU Mascot

Master's thesis in Cybernetics and Robotics
Supervisor: Associate Professor Sverre Hendseth,
PhD Candidate Martinius Knudsen
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Task Description

The NTNU Cyborg project aims to enable communication between living nerve tissue and a robot, thereby developing a cyborg. The work in this project focuses on bringing the Cyborg robot to a state where it is ready for demonstration, this entails restructuring and improving the software on the Cyborg robot and integrating existing and previous work. The student shall:

- Evaulate and reimplement the existing software with the overall aim of making the Cyborg less complex and easier to work with.
- Design and implement a coordinator module for frequently used output modules.
- Resume the work with the LED controller presented in the authors specialization project and integrate it into the Cyborg.
- Assist and provide guidance to groups from the Kyborg Experts in Team village on their work with the Cyborg.
- Finish the Cyborg body and LED dome.

Abstract

The NTNU Cyborg project aims to enable communication between living nerve tissue and a robot, thereby creating a cyborg. This thesis presents work done on one of the projects robots, *The Cyborg*, aimed at being transformed into a cyborg and university mascot. The focus is on making the Cyborg ready for demonstration. The Cyborg body and LED dome needs to be finished and integrated, and all current Cyborg ROS modules integrated.

Motivation for a simpler software structure and an evaluation of the Cyborg is presented. The current Cyborg ROS modules are partly reimplemented, with the aim of reducing complexity. A behavior module is also designed and implemented. The module makes commonly used output modules available through a single interface, while providing a way to configure simple behavioral presets that can be used by other modules or as states for the Cyborg state machine. The Cyborg Start-up Box is upgraded, adding the ability to terminate and select new modes of operation on the Cyborg, the name is changed to Mode Selector in order to better fit the new function.

As a continuation of the authors specialization project, the controller for the LED dome is realized and the corresponding software finished. The author has assisted Experts in Team groups with their work on the Cyborg body, LED dome, and software, their contribution has proven greatly beneficial for the project. The Cyborg body is finished and the LED dome integrated into the Cyborg.

An argument is made for why the restructured software was the right

choice, and a list of proposed future tasks is presented. At the end of this thesis the Cyborg satisfies the presented specifications and is ready for demonstration.

Sammendrag

NTNU Cyborg-prosjektet har som målsetning å muliggjøre kommunikasjon mellom levende nerveceller og en robot, og dermed utvikle en cyborg. Denne oppgaven presenterer arbeid utført på en av prosjektets roboter, *The Cyborg*, som på sikt skal transformeres til en cyborg og maskot for universitetet. Fokuset er på å gjøre the Cyborgen klar for demonstrasjon. Kroppen og LED domen til Cyborgen må ferdigstilles og integreres, og resterende Cyborg ROS moduler integreres.

Motivasjon for en enklere programvarestruktur og en evaluering av Cyborgen presenteres. De nåværende Cyborg ROS modulene blir delvis reimplementert, med formål om å redusere kompleksiteten. En adferdsmodul blir designed og implementert, modulen gjør flitting brukte utgangsmoduler tilgjengelig via et felles grensesnitt, og lar en forhåndsinstille adferder som kan bli brukt av andre moduler eller som tilstander i tilstandsmaskinen.

Cyborg start boksen blir oppgradert for å gjøre det mulig å stoppe og starte nye moduser, og navnet blir endret for å passe den nye funksjonaliteten bedre. Som en videreføring av forfatterens spesialiseringsprosjekt, blir kontrolleren til LED domen realisert, og tilhørende programvare ferdigstilt. Forfatteren har assistert grupper fra Eksperter i Team med deres arbeid på kroppen, LED domen og programvare relatert til Cyborgen, deres bidrag har vist seg å være til stor nytte for prosjektet. Kroppen til Cyborgen er ferdigstilt, og LED domen integrert i Cyborgen.

Det blir argumentert for hvorfor en omstrukturering av programvaren var

det rette valget, og en liste over foreslåtte fremtidige oppgaver presenteres. På slutten av denne oppgaven er tilfresstiller Cyborgen de angitte spesifikasjonene, og den er klar for demonstrasjon.

Acknowledgements

I would like to express my deepest grattitude to Sverre Hendseth for his invaluable guidance throughout the work for this thesis. The regular meetings, with topics ranging from structure of a thesis to interesting philosophical conversations about life in general have been greatly motivating and appreciated. His willingness to dedicate substantial parts of his time to a great number of students is nothing but amazing.

I would also like to extend my thanks to Martinius Knudsen for facilitating the Cyborg and making himself available for assistance when requested. I hope my contributions to the project will be of as much value as I enjoyd being involved.

Finally, I wish to thank my grandfather Enn Tõugu, without his inspiration and dedication to cybernetics, I would never have ended up here in the first place.

Preface

This Master's thesis has been conducted at the Department of Engineering Cybernetics at the Norwegian University of Science and Technology. During my work for this thesis, I have been granted full access to all hardware related to the Cyborg, including the Pioneer LX robot that serves as the base for the Cyborg and my work. Through regular meetings, supervisor Associate Professor Sverre Hendseth has provided me with guidance and advice regarding the procedure and layout for my thesis. In addition, assisting supervisor PhD Candidate Martinius Knudsen has made himself available when requested, his contribution has been on the form of consultation regarding my own approaches to the work on the Cyborg.

Contents

Abstract **i**

Acknowledgements **v**

Preface **vi**

1 Introduction **1**

1.1 The Cyborg: An NTNU Mascot 2
1.2 Motivation and Goal 2
1.3 Contributions of the Author 3
1.4 Previous work on the NTNU Cyborg Project 4
1.5 Work not Mentioned any Further 4

2 Background **6**

2.1 Evolution of the Cyborg 6
2.2 Hardware Structure of the Cyborg 6
2.2.1 The Cyborg Base; Pioneer LX 9
2.2.2 The Cyborg LED Dome 10
2.2.3 The Cyborg LED Controller 10
2.2.4 Arduino 11
2.2.5 NodeMCU ESP-32S 11
2.2.6 The Start-up Box 12
2.2.7 Jetson TX2 Developer Kit 13

2.2.8 Zed Stereoscopic 3D Camera 13
2.2.9 MEA2100-Systems Microelectrode Array 13
2.3 Software and Tools 14
2.3.1 RViz 14
2.3.2 Shell Scripts 15
2.3.3 Signals 15
2.3.4 Rhinoceros 3D 15
2.3.5 Fritzing - Open Source ECAD 16
2.3.6 FreeRTOS 16
2.4 ROS - The Robot Operating System 18
2.4.1 Concepts of ROS 18
2.5 SMACH - A State Machine Library 26
2.5.1 SMACH States 27
2.5.2 How to Create a Simple SMACH State Machine 28
2.6 Autonomous Navigation of Mobile Robots 31
2.6.1 Adaptive Monte Carlo Localization 31
2.7 Software Structure of the Cyborg 32
2.7.1 Controller 35
2.7.2 Navigation 35
2.7.3 LED Dome 36
2.8 Our Long Term Vision 36
2.9 Terms and Definitions 37

3 Evaluating the Cyborg 41

3.1 Introduction 41
3.2 The Cyborg v3.0 41
3.2.1 Lack of Integration 42
3.3 Overall Software Structure 43
3.4 Proposing a new Software Structure 44
3.5 Navigation 45
3.5.1 Preliminary Testing Issues 46

3.5.2 Navigation Module 47
3.5.3 Proposed Task List 48
3.6 Audio 48
3.6.1 Proposed Task List 48
3.7 Controller Node 49
3.7.1 Proposed Task List 49
3.8 LED Dome 50
3.8.1 Proposed Task List 50
3.9 Discussion 51
3.10 Conclusion 52

4 Assisting Experts in Team Groups 54
4.1 Introduction 54
4.2 Group 1 - Simple Neural Response Interpreter - SiNRI 55
4.3 Group 3 55
4.3.1 Frosting and Assembling the LED Dome 56
4.3.2 Sanding and Painting the Body 57
4.4 Discussion and Conclusion 57

5 The Mode Selector Box 59
5.1 Introduction 59
5.2 Software Modifications 60
5.2.1 Cyborg Base 61
5.2.2 Mode Selector 61
5.3 Scripts 62
5.4 How to add new Modes 62
5.5 Discussion 63
5.6 Conclusion 64

6 The Behavior Module 66
6.1 Introduction 66

6.2 Requirements and Specifications 66
6.3 Design 67
6.4 Implementing the Behavior ROS Module 68
6.4.1 How to add new Behavioral Presets 72
6.5 Discussion 72
6.6 Conclusion 73

7 Cyborg Audio Module **74**
7.1 Introduction 74
7.2 Requirements and Specifications 75
7.3 Design 75
7.4 Implementing the Cyborg Audio ROS Node 75
7.4.1 Playback 76
7.4.2 Text To Speech 77
7.5 Discussion 78
7.6 Conclusion 79

8 The Event Scheduler Module **80**
8.1 Introduction 80
8.2 Requirements and Specifications 81
8.3 Design 81
8.4 Implementing the Event Scheduler ROS Node 82
8.5 Discussion 83
8.6 Conclusion 83

9 Primary States Module **84**
9.1 Introduction 84
9.2 Requirements and Specifications 85
9.3 Design 85
9.3.1 Wandering Emotional State 86
9.3.2 Navigation Planning state 86

9.4 Implementing the Primary States Action Server Node 87
9.5 How to Implement new States 88
9.6 Discussion 89
9.7 Conclusion 89

10 Navigation Module 90

10.1 Introduction 90
10.1.1 Design 90
10.1.2 Reimplementing the Navigation ROS Node 92
10.2 Discussion 93
10.3 Conclusion 94

11 LED Dome 95

11.1 Introduction 95
11.2 Implementing the LED controller Circuit 95
11.3 Redesigning and Implementing the LED Dome Software . . . 96
11.3.1 Technical Considerations and Requirements for the LED Controller Interface 96
11.3.2 Interface for the LED Controller 97
11.3.3 The LED Dome ROS Module 100
11.4 Preparing the LED Dome for Integration into the Cyborg . . . 101
11.4.1 Casing 101
11.4.2 LED dome ROS module 102
11.4.3 LED-dome 102
11.5 Discussion 102
11.6 Conclusion 103

12 The Finishing Touches 104

12.1 Introduction 104
12.2 Cyborg Body and LED Dome 104
12.2.1 Protective Fan Cover 104

12.2.2 Covering the Gap Between the LED Dome and Cyborg Body 105
12.2.3 Aligning and Fastening the Body Panels 105
12.2.4 Mounting the LED Controller 106
12.3 Configuring the State Machine 107
12.4 Preparing the Cyborg for Testing 107
12.5 Testing the Cyborg v3.0 108
12.6 Discussion 110
12.6.1 Navigation 111
12.6.2 Cyborg Body and Hardware 111
12.7 Conclusion 112

13 Discussion **115**
13.1 Proposed Future Work 115

14 Conclusion **118**

Appendices **121**

A Diagrams **121**

B Notes and Documentation for Navigation Tuning **123**
B.1 Tuning Guides 123
B.2 Parameters in MobileEyes 123
B.3 Errors Encountered 124
B.4 Relevant Excerpts 124

References **127**

Chapter 1

Introduction

The work presented in this thesis is carried out as a part of the NTNU Cyborg project at the Norwegian University of Science and Technology (NTNU). The NTNU Cyborg project is an interdisciplinary project involving several departments at NTNU, the project is driven foreward by NTNU researchers, PhD candidates, MSc students, student specialization projects and student groups from the Kyborg Experts in Team (EiT) village. The overall goal of the project is to enable communication between living nerve tissue and a robot, thereby creating a true cyborg, in the hope of achieving a better understanding of consiousness, and in the process also create a platform for interdisciplinary collaborations and teaching. This is going to be achieved by enabling communication between a biological neural network located at St. Olavs Hospital and a robot at the university campus. The biological neural network is produced by growing stem cells, extracted from either humans or rats, over a Micro-Electrode Array (MEA). The MEA enables us to capture activity in, and also stimulate the neural network, thereby enabling interaction with the neurons. The hope is to create a closed loop system with realtime bidirectional communication between neurons and the robot. Using the activity in the neurons to activate and control the robot, while sending sensory data from the robot to the neural network, giving the

biological neural network actual senses, and achieving a system which will allow the neural culture to operate and learn within its working environment.

1.1 The Cyborg: An NTNU Mascot

One of the robots in NTNU Cyborg fleet is the Pioneer LX robot, an autonomous robot base, capable of autonomous navigation. The robot base serves as a foundation upon which various other components can be mounted, the whole ensemble is fittingly dubbed *The Cyborg*. The long term vision is to transform the robot into an autonomous cyborg, capable of roaming the university hallways and interact with people, while using activity in neurons for some tasks. The hope is that the Cyborg will garner a lot of attention and pique technological interest, while showcasing several fields of forefront research, ultimately achieving the status of a true NTNU Mascot.

1.2 Motivation and Goal

At the onset of this thesis the Cyborg has been in development for over four years without being properly operational, thereby by not fulfilling its ultimate goal. Throughout these years the Cyborg has undergone great changes, bringing it closer and closer to the overall goal of an autonomous cyborg. But in order for the Cyborg to garner attention, it must first achieve a state in which it can be used actively in public. More attention can in turn potentially lead to an increase in students wanting to involve themselves, thereby aiding the project by further accelerating the development of the Cyborg. This makes it even more important to finally make the Cyborg ready for demonstration and get it out there to be exposed. In order for the Cyborg to be ready for demonstration, it should satisfy the requirements of the Cyborg v3.0 stated in the Section 3.2.

Previous EiT groups have shown to be great assets to the project, and

the author wishes to utilize this potential and facilitate for their work. The author has previous experience with many of the different tasks at hand, and it is important to keep track of all the tasks that need to be done in order to achieve the goal of the Cyborg v3.0. It is specially important that the bodywork is done properly, both because of the high production cost and the fact that we want to expose the Cyborg. As such, an effort will be made to provide assistance and guidance to the groups involved with the Cyborg spring 2019.

1.3 Contributions of the Author

The work in this thesis aims to make the Cyborg ready for demonstration. This entails finishing the Cyborg body and hardware, enhancing current software, and finishing the software for the LED dome. A list of the authors contributions is presented:

- The current software is evaluated and partly reimplemented in order to enhance the Cyborg software and make it easier to work with.
- A behavior module is implemented, providing an easy way to configure new states, while gathering the most commonly used output modules under a single access point.
- The interface for the LED controller for the LED dome is reimplemented, and the corresponding Cyborg ROS module made ready for integration.
- The Mode Selector box is upgraded, adding the ability to shut down and start new modes without restarting the Cyborg base.
- The Cyborg body and LED dome is finished in collaboration with EiT group 3.
- A state machine is configured, and all implementation is tested at the university campus.

1.4 Previous work on the NTNU Cyborg Project

I was first involved with the Cyborg project through my specialization project fall 2018 [1], working with the LED dome and preparing it for integration into the Cyborg. I have designed and tested a prototype circuit for controlling the leds on the LED dome, implemented a ROS module for visualizations, and software for the led controller circuit. In addition, I have created documentation for several of the existing ROS modules.

1.5 Work not Mentioned any Further

- A lot of time was spent on tweaking navigational parameters and collecting information regarding navigation tuning. I did not find a proper way to present all this material, and I did not want to write a new navigation tuning guide since it has already been done before me. Instead, I have decided to include the information, both own notes and documentation in Appendix B, hoping that my efforts might ease the burden for the students resuming this task.
- I surveyed solutions for the gap between the LED dome and Cyborg body, and went shopping for the fan cover and rubber gasket for the LED dome.
- I attended EiT demonstrations and workdays.
- At the onset of this thesis I thought perhaps there was an issue with the battery for the Cyborg base. It looked like the battery did not charge, and the manuals for the robot were lacking proper information regarding battery status. I contacted Omron Norway in order to evaluate our options regarding a new battery, only later to find out that manuals for the upgraded version of the robot contained all the necessary information, and that the battery was actually in perfectly good shape.

- Parts of Sections 2.2 - 2.6 are based on corresponding parts from my specialization project, and are added for the sake of completeness.

Chapter 2

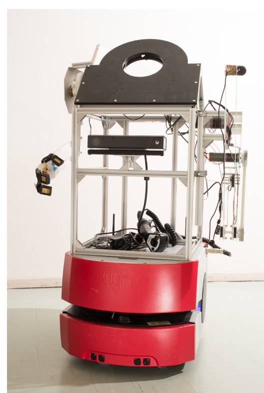
Background

2.1 Evolution of the Cyborg

The Cyborg has undergone some major design changes since the birth of the project. Early interations were characterized by an open design with moving parts, but the vision has since evolved into a more compact and closed off design. The final design vision features a sturdier Cyborg, better equipped to cope with curious spectators, and with the aestetichs to mach the role of a proper NTNU mascot. The evolution of the Cyborg design is shown in Figure 2.1, the final vision is seen in Figure 2.2.

2.2 Hardware Structure of the Cyborg

The hardware structure of the Cyborg is presented in Figure 2.3.

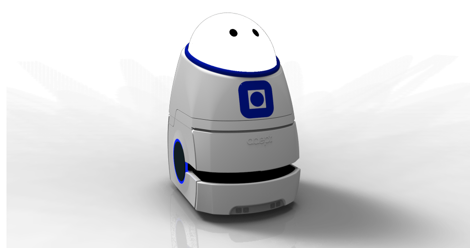


(a) First generation

(b) Second generation

(c) Cyborg at onset of thesis

Figure 2.1: Evolution of the Cyborg.



(a) Front

(b) Back

Figure 2.2: Final vision for the Cyborg.

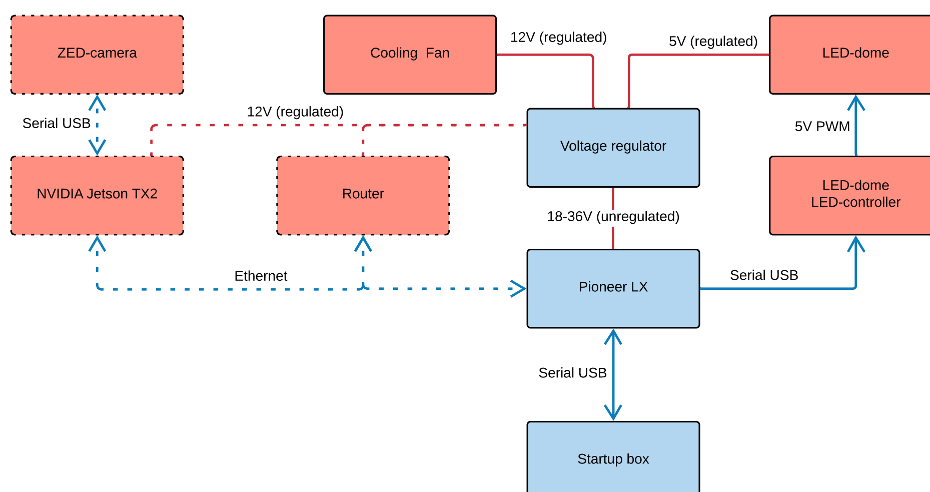


Figure 2.3: Overview of the hardware structure of the Cyborg. Red boxes indicate components not integrated at the onset of this thesis, while dashed components and lines indicate future plans.

2.2.1 The Cyborg Base; Pioneer LX

The Cyborg features a Pioneer LX robot [3], serving as a foundation for the rest of the components to be mounted upon. The Pioneer LX robot is based on a platform for autonomous indoor vehicles, its intended use is to function as a platform for research, education, and development. It features an onboard computer compatible with both Linux and Windows, various sensors and tools, powered wheels for moving around, and support payloads of up to 60 kg. Some software libraries and tools are also supplied, including software for navigation and mapping of environment, described in more detail in Section 2.3. The Pioneer LX features the following hardware:

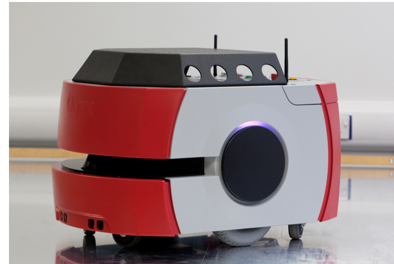


Figure 2.4: The Pioneer LX robot by MobileRobots. Image courtesy of [2].

- Intel D525 64-bit dual core CPU @1.8 GHz
- Intel GMA 3150 integrated graphics processing unit
- Intel 6235ANHMW wireless network adapter
- Ports for ethernet, RS-232, USB, VGA, and various other analog and digital I/O
- SICK 300 and SICK TiM 510 laser scanner, for navigation and object detection.
- Sonar sensors and a bumper panel.
- Joystick for manual control.
- A 60 A h battery, expected to power the robot continuously for up to 13 hours.
- Automated charging station, allowing the robot to dock autonomously.

2.2.2 The Cyborg LED Dome

The LED dome shown in Figure 2.5 is an integral part of achieving the goal of a true cyborg. The LED dome provides a display for visualizations of neural data from the biological neural network at St. Olavs, enabling interaction between robot and living tissue. In addition to this, it can also be used to show text and other animations. The LED dome consist of a molded plastic dome, with a WS2812B LED strip attached to the surface. The LED strip consists of several smaller pieces, mounted together to get one consecutive strip of the desired length, incorporating a total of 791 LEDs. The leds are activated by a 5 V-PWM signal from the LED controller, and require a 5 V voltage supply.

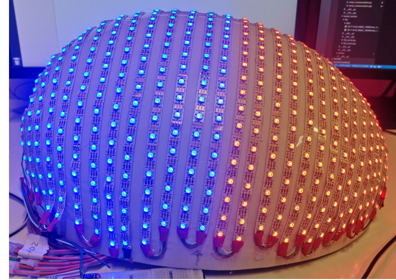


Figure 2.5: The Cyborg LED dome.

2.2.3 The Cyborg LED Controller

The LED controller for the LED dome is a prototype circuit presented in the authors specialization project [1], a hardware diagram for the LED controller is seen in Figure 2.6. The LED controller parses serial data from a Cyborg ROS module, and activates the leds on the LED dome accordingly.

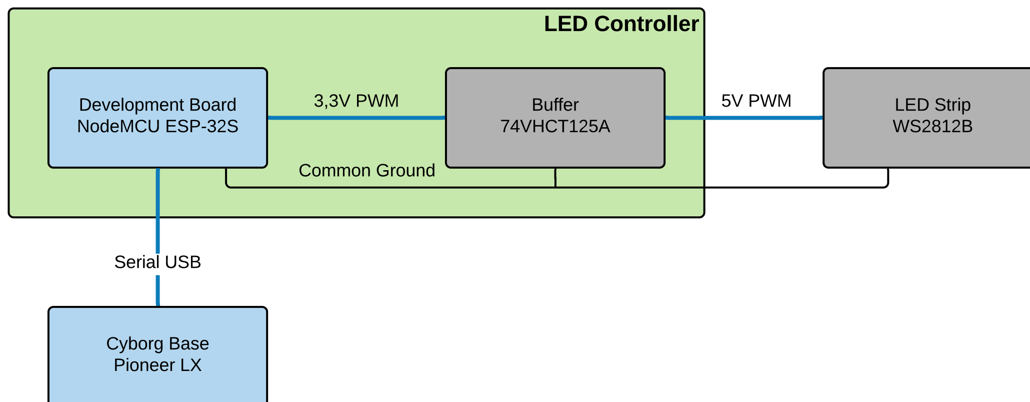


Figure 2.6: Hardware diagram for the Cyborg LED dome controller.

2.2.4 Arduino

Arduino [4] is an open-source hardware and software project and company. They design and manufacture microcontrollers and single-board computers, and through the project provide the Arduino integrated development environment (IDE).

2.2.5 NodeMCU ESP-32S

The ESP-32S (ESP32) is a development board manufactured by NodeMCU [6], based on the ESP-WROOM-32 module [7] by Espressif Systems. Specifications for the ESP32 is listed below:

- 32-bit dual core architecture.
- 240 MHz clock speed.
- 3.3 V operating voltage.
- Accepted input voltage 6-20 V.
- 38 I/O pins, with support for UART, pulse width modulation, and DAC output.

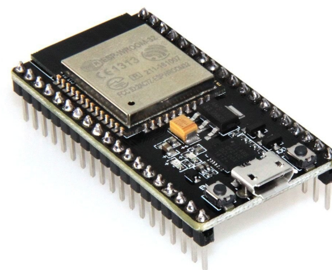


Figure 2.7: NodeMCU ESP-32S development board. Image courtesy of [5].

<table>
<tr><td colspan="6">Lua interpreter</td></tr>
<tr><td>PIO</td><td>ADC</td><td>CAN</td><td>PWM</td><td>LoRa</td><td>MQTT</td></tr>
<tr><td>SPI</td><td>I2C</td><td>UART</td><td>...</td><td>Sensors</td><td>...</td></tr>
<tr><td colspan="6">Real-Time micro-kernel</td></tr>
<tr><td colspan="6">Hardware Abstraction Layer</td></tr>
</table>

Table 2.1: The 3-layer design of Lua RTOS is illustrated.

- USB Micro-B connector. Can be used for powering and interfacing the board.

The ESP32 runs the Lua real-time operating system (Lua RTOS), and is programmed in the Lua programming language, utilizing the eLua IDE. Lua RTOS features a 3-layer design, illustrated in table 2.1, where the middle layer real-time micro-kernel is powered by FreeRTOS. Through some minor configurations, it possible to switch out the top layer and instead use the Arduino IDE and programming language, while keeping FreeRTOS in the middle layer. [8]. This makes coding simpler for non-concurrent tasks, and in addition enables the use of many of the libraries already written for Arduino boards. Its important to note that the Arduino code is compiled and ran on only one core, if one wishes to utilize both cores, FreeRTOS tasks have to be used.

2.2.6 The Start-up Box

The Start-up Box presented by Waløen in his thesis[9] is used to select wanted mode of operation when powering on the Cyborg. The modes of operation are defined by which modules that are executed on the Cyborg, if one wishes to change mode, the Cyborg base has to be restarted.

2.2.7 Jetson TX2 Developer Kit

The Jetson TX2 Developer Kit[11] from NVIDIA shown in Figure 2.8 is an AI supercomputer on a module. Intended at artificial intelligence processing in real-time, suitable for deployment of computer vision and deep learning. Its low power consumption to performance ration, makes it ideal for use in intelligent edge devices like robots and drones.

Figure 2.8: NVIDIA Jetson TX2 Developer Kit. Image courtesy of [10].

2.2.8 Zed Stereoscopic 3D Camera

The Zed stereoscopic 3D camera by Stereolabs shown in Figure 2.9 can be used for capturing 3D video, depth perception, spatial mapping, and positional tracking with 6 degrees of freedom. It features two 4MP cameras, and is according to Stereolabs, the worlds fastest depth camera [13].

2.2.9 MEA2100-Systems Microelectrode Array

Microelectrode arrays (MEAs) can be used for interfacing between biological neurons and electronic circuits, or in other words interpret and deliver signals in living tissue through a computer. This is achieved by measuring and exciting electrical currents in the biological neurons. The MEA2100-System is a stand-alone multi-channel MEA system, it provides means for data acquisition, signal amplification, stimulus generation, temperature control, and online signal processing.

Figure 2.9: Zed Stereoscopic 3D Camera. Image courtesy of [12].

2.3 Software and Tools

The Pioneer LX robot base is delivered with the following software and tools:

- ARIA - Advanced Robot Interface for Applications, a legacy library by MobileRobots, used for controlling Pioneer-compatible mobile robots. This is the core development library for the robot, providing access to and management of the robot controller, effectors and sensors. Access to the robots peripherals through this library is done through the ROS interface module *ROSAIRA*.

- ARNL - Advanced Robot Navigation and Localization, also by MobileRobots. ARNL is built on top of ARIA, and is a development library for including indoor laser localization and autonomous navigation. The library also includes *arnlServer*, used for interactive initiation of autonomous navigation, either through custom software or the provided *MobileEyes*.

- Mapper3 - Used for converting and editing maps for use with ARNL and MobileSim.

- MobileSim - A simulator for testing the robot software. When in use, ARIA connects to MobileSim instead of the physical robot.

- MobileEyes - MobileEyes provides a GUI for remote visualization, teleoperation, and software configuration. The software lets the user monitor and control the robot, and configure its system parameters remotely. MobileEyes interfaces *arnlServer* and connects wirelessly to the robot or simulator.

2.3.1 RViz

RViz (ROS Visualization) is a 3D visualization tool for ROS . The tool can be configured to display and visualize a variety of data through plugins, providing

more configurations than *MobileEyes*.

2.3.2 Shell Scripts

A shell script is a program designed to be ran in the Unix Shell, the commands in the script file are ran as if written directly on the command line. Shell scripts provide an interface for the user to use operating system services, they enable us to run multiple command line commands without having to type, letting us avoid doing repetitive work e.g. when executing a system comprised of multiple modules.

2.3.3 Signals

Signals are software generated asynchronous notifications that can be sent to processes or threads within processes [14], enabling us to manage and interrupt normal executional flow. In order to act accordingly when a signal is received, the signaled process needs to implements a signal handler. A various selection of different signals are available, the ones most relevant for this thesis are *SIGINT*, *SIGTERM*, and *SIGKILL*. *SIGINT* is sent when a user want to interrupt a process, this signal can be caught and handled, and gives the process the ability to shut down gracefully. In contrast to this, *SIGKILL* terminates the process immediately, without letting the program save unsaved data.

2.3.4 Rhinoceros 3D

Rhinoceros (Rhino3D) is a commercial 3D CAD software [15], the software can be used for rapid prototyping and 3D printing.

2.3.5 Fritzing - Open Source ECAD

Fritzing [16] is an open-source initiative, aimed at developing hobbyist CAD software for designing electronics hardware, aiding the process of moving hardware design from the experimental prototyping stage onto more permanent circuits. The application is an easy to understand ECAD tool mainly used for designing PCBs, its not as advanced as many alternatives, but in return requires minimal effort to get into.

2.3.6 FreeRTOS

FreeRTOS is a real-time os-kernel for embedded systems [17] distributed permissive free under the MIT License [18]. FreeRTOS is designed to be small and simple, it leaves a small memory footprint while providing a low overhead and fast execution. Methods for multiple threads, tasks, software timers, mutexes, and semaphores are provided, in addition to support for thread priorities. When a development board running a FreeRTOS kernel is set up to work in the Arduino IDE, FreeRTOS can be used freely. A short example on how to use tasks assigned to specific cores is given:

```
1  TaskHandle_t Task1;
2
3  xTaskCreatePinnedToCore(
4      TaskCore0,     /*Function to implement the task */
5      "Function1",   /*Name of task*/
6      1000,          /*Stack size of task*/
7      NULL,          /*Task input parameter*/
8      1,             /*Priority of task*/
9      &Task1,        /*Task handle*/
10     0);            /*Core to run the task on*/
```

- An object handle for the task is created in line 1.
- The task is created in line 3-10, this has to be done in the *setup*-function of the Arduino Sketch.

- The task can be given a priority, higher number means higher priority.

```
Void TaskCore0(void * parameter){
    for(;;){
    /*Code for the task is implemented her*/
    }
}
```

- The function behaves similar to the Arduino *loop()*-function, and should implement an infinite loop.
- The task can be stopped by using executing *vTaskDelete(Task1);*

For passing data between functions on different cores, FreeRTOS queues are a good thread-safe alternative. Tasks that try to consume data from an empty queue, or insert data into a full queue can be blocked:

```
QueueHandle_t queue;

void setup(){
    queue1 = xQueueCreate(4, sizeof(int));
}
int i = 1;
xQueueSend(queue1, %i, portMAX_DELAY);

int rec;
xQueueReceive(queue1, &rec, portMAX_DELAY);
```

- An object handle for the queue is created on line 1.
- The queue is created on line 4.
- Data is put into the queue on line7, and consumed while storing it in *rec* on line 9-10.
- The maximum blocking time can be chosen by modifying *portMAX_-DELAY*.

2.4 ROS - The Robot Operating System

The Robot Operating System (*ROS*)[19] is an open source robotics middleware, aimed at simplifying working with complex robotics projects. The framework provides means for implementing a highly modular peer-to-peer network of processes, suitable for controlling distributed systems, enabling easy scaling of projects. As stated on the ROS wiki, the primary goal of ROS is to support code reuse in robotics research and development, and not to be the robotics middleware with the most features. The idea is that if an expert laboratory has developed a module for a specific tasks, others should not have to reinvent the wheel, and instead be able to gain from their expertise and save valuable time by using what has already been implemented. ROS provides services for low-level device control and hardware abstraction, and implements functionality for message passing between processes and package management. It is an ever growing collection of tools and libraries, most of them licensed under a variety of open source licenses, and it is encouraged to contribute to the collection of available libraries.

ROS is designed to be thin and flexible, as part of their mantra *'we do not wrap our main()'* main functions are not wrapped[20], making it easy to use code written for ROS with other robot software frameworks. ROS is also language independent, with implementations for C++ Python and Lisp, client libraries *roscpp*, *rospy*, and *roslisp*, respectively. Code examples presented in this chapter are for the *rospy* client library.

2.4.1 Concepts of ROS

Nodes

Nodes are the atomic grains in ROS, connected together they make up a robot control system. They can be viewed as processes that perform computation, and should each be responsible for one task. For example, in a robot control system, you might have one node responsible for driving the wheels of the

robot, one for planning the path the robot should take, one for localizing where the robot is, etc. Nodes communicate peer-to-peer, using topics, services, and the Parameter Server. A node is implemented using one of the available client libraries, each one with its differences. Nodes are identified by their name, best practices for ROS states that the name chosen should be descriptive of the task the node is responsible for. The usage of nodes has several benefits. Crashes are isolated to the faulty node, making the system fault tolerant. They support the modularity aim of ROS, and are easy to interchange. And they reduce code complexity, compared with monolithic systems.

ROS Master

All systems need a ROS Master, the ROS Master is started by issuing either the *roslaunch* or the *roscore* command on the command line. The ROS Master is a name service for ROS, it keeps track of all the running nodes, topics and services available, and enables nodes to find each other by making all this information available to them. Once the nodes have located each other through the ROS Master, they connect directly using the provided methods. All nodes publish their registration information to the ROS Master, each time the registration information of a node changes, the ROS Master updates all nodes with this information through a callback. This allows nodes to dynamically create connections when a new node is made available.

Parameter Server

A Parameter Server is running inside the ROS Master, it provides nodes with a central location to store and retrieve data at runtime [21]. It is not a high-performance solution, and is intended for storing globally accessible static data, such as configuration parameters. As such, this also provides an easy way of monitoring and modifying system parameters. The parameters should be stored in a hierarchical fashion, in accordance with the ROS naming convention. The provided command-line tool *rosparam* can be used to get a

list of, access, and modify the parameters stored on the Parameter Server. Some commonly used lines of code for the parameter server in Python is given below:

```
# Fetch parameter
value = rospy.get_param("/node_name_space/parameter_name",
    default_value)

# Check parameter existence
rospy.has_param("parameter_name")

# Setting parameter
rospy.set_param("parameter_name" , parameter_value)
```

- A parameter is fetched on line 2. If no parameter is found, the optional default value is used instead.
- Line 5 checks if a parameter exists, returns *True* if parameter is set, else *False*.
- A parameter is set on line 8.

Messages

Communication between nodes is done through message-passing. A message is a simple data structure, supporting standard primitive types like integer, float, boolean, and arrays of supported primitives. Much like a data structure in C, messages also support arrays and arbitrary nested data structures. Messages consist of fields and constants, the fields contain the data being sent, while the constants can be used for defining useful values that can be used to interpret the defined fields. Each field consist of a type and a name, whereas a constant also assigns a value. The format of a message is simply a field or constant on each line. Messages are stored as *.msg*-files in the `msg/` subdirectory of the package, a typical message is on the form:

```
fieldtype1 fieldname1
```

```
fieldtype2 fieldname2
constanttype1 CONSTANTNAME1 = constantvalue1
```

and is normally initialized like this:

```
message = MessageType()
message.data = value
```

Where the first line initializes a message of type *MessageType()*, and the second line sets the value of the argument *data* in the message. A set of standard messages is already provided in the *std_msgs* package, however it is also possible to define custom message types if needed. In order to use messages, they need to be translated into source code by the Client Libraries[22].

Topics

Topics are the transport layer of messages, they follow a publish-subscribe model, where a topic can have multiple publishers and subscribers. This allows for easy communication between nodes, since nodes interested in a particular topic can subscribe without being aware of who the publisher is. The name of a topic is used to identify the content on the topic, a node that wants to send a message simply publishes it on the topic with the corresponding name. Topics are intended for unidirectional asynchronous communication, and are most commonly used for publishing continuous streams of data, remote procedure calls between nodes are better handled by Services. Publishers and subscribers in Python are declared like this:

```
pub = rospy.Publisher('chatter', String, queue_size=10)
rospy.Subscriber('chatter', String, callback)
```

The first line declares that your node is publishing a message of type *String* on the topic ‘chatter‘. *queue_size* is used for defining a max queue-length, in case messages are published faster than the subscribers can receive them. The second line declares that your node is subscribing to the same topic,

when new messages are received, the function "callback" is invoked with the received message as the first argument. To publish a message, you call the publish function of the publisher, with the message as argument:

```
pub.publish(message)
```

The command-line tool *rostopic* provides different services for working with topics, a list of all the active topics is made available by issuing the command *rostopic list*.

Services

Services are used where a synchronous interaction is needed, for example for remote procedure calls. A service is defined by a message pair, a request and a reply message. The node offering a service acts as a server identified by a string name, clients can request the service through a request message to the name of the service. When called upon, the service executes and a reply message containing the result of the operation is sent back to the client. Much like topics have their associated *.msg*-files, each service has its associated service type *.srv*-file that defines request and response parameters, the service type is defined by the name of the package combined with the name of the *.srv*-file. A *.srf*-file is essentially just two messages put together and separated by a line of "- - -", where the first part is the request message, and the second part is the response message. Service files are stored in the `srv/` subfolder of the package, and are on the format:

```
requesttype1 requestname1
requesttype2 requestname2
---
responsetype1 responsename1
responsetype2 responsename2
```

Like messages, service files need to be built before they can be used, more on this can be found in [23]. A service is declared with the line:

```
s = rospy.Service('servicename', ServiceType, servicefunction)
```

where *'service'* is the name of the service, and *ServiceType* is the service type. *servicefunction* is the function providing the requested service, it is called when the service is requested with *ServiceType*-instances as input and output. As a minimum, a service client usually contains the following lines of code:

```
1 rospy.wait_for_service('servicename')
2 service = rospy.ServiceProxy('servicename',ServiceType)
3 res = service(request)
4 return res.response
```

- The first line utilizes a convenience method for blocking until the service is available, while the second provides the handle for calling the service.
- The service is called on line 3, with the input variable *request* and stores the result as an object of the service type.
- On line 4 the service response value stored in the *response*-variale of the service object is returned.

Two command-line tools for ROS services are provided, *rossrv* and *rosservice*, used for displaying information about *.srv*-files and querying information about ROS services, respectively.

Actions

Actions are an alternative to services, intended at controlling long-running tasks. In contrast to services, actions can be preempted and the action server can send feedback while executing. Tools for creating action client interfaces and preemptable action servers are provided through the *actionlib* package. The communication between action clients and action servers goes through the *ROS Action* protocol, which is built on top of ROS messages. As with topics and services, actions also need messages to define the action specifications, *.action*-files, saved in the `action`-subfolder of the package. An action-file consist of three messages, *goal*, *feedback*, and *result*. The goal is sent from a client to a server, and states what you want to accomplish with the action.

For example, if you want to move a robot arm to a specific position, the goal could be certain joint-angles for the arm. The feedback message is for the server to send asynchronous progress-feedback during execution of a goal. Upon completion of a goal, the server responds with a result. In contrast to feedback, the result is only sent once. The layout of a *.action*-file is shown below:

```
#goal definition
goaltype goalname
---
#result definition
resulttype resultname
---
#feedback
feedbacktype feedbackname
```

.action-files need to be generated during the make process.

The *ROS Action protocol* automatically sets up five topics for communication between the server and client, *goal*, *result*, *feedback*, *cancel*, *status*. The first three are used for the messages described above, while *cancel* and *status* are for messages predefined in the `actionlib_msgs`-folder. *cancel* lets clients send cancel requests to action servers, and *status* lets action servers send updates to clients regarding the status of all goals the action server is tracking. When an action goal is sent, the action client registers a timestamp and generates a unique *Goal ID*. The goal message is then wrapped in a *ActionGoal*-message, together with the timestamp and Goal ID, before it is sent to the server. The Goal ID provides a way for client and server to associate specific goals with messages being transported. The Goal ID is also appended onto messages going over the *cancel*, *feedback*, and *result* topic (for *cancel* the timestamp is appended as well), providing different benefits for each of the topics.

The following example shows the minimum lines of code needed to set up a ActionClient and send a goal to a ActionServer called "server":

```python
from Example.msg import ExampleAction,ExampleGoal

#create client and connect to server
client = actionlib.SimpleActionClient("server", ExampleAction)
client.wait_for_server()

#create and send goal
goal = ExampleGoal(goalname = "do_action")
client.send_goal(goal, callback_done)
```

- The action type and messages are imported on line 1.
- On line 4 the client is initiated and connected to the action server, with action type ExampleAction.
- Line 5 makes execution wait until the client is properly connected to the server.
- The goal is created and filled on line 8.
- On the last line the goal is sent to the server. In addition an optional callback is connected, the callback executes when the server returns a result.

While theActionServer is set up as follows:

```python
from Example.msg import ExampleAction, ExampleResult

#create message for result
result = ExampleResult()

#create and start server
actionserver = actionlib.SimpleActionServer("server",
    ExampleAction, execute, auto_start = False)
actionserver.start()

def execute(self, goal)
    #Implement action here

    #Publish result and set server state
    result.resultname = "action succeeded"
```

```
15        self.actionserver.set_succeeded(result)
```

- The action type and messages are imported on line 1.
- A result message is created on line 4.
- On line 7-8, the ActionServer "server" is created and started, with action type ExampleAction. "execute" is executed when a goal arrives.
- The function for the action is implemented on line 10-15.
- On line 14 the result message is filled.
- Line 15 sets the terminal status of the action server, and publishes the result message to the client.

2.5 SMACH - A State Machine Library

SMACH [24] (*S*tate *MACH*ine) is an open source Python library for building hierarchical and concurrent state machines. SMACH is useful for handling structured tasks, task-level execution and coordination, and it allows for easy building, maintenance, and debugging of complex state machines.

At its core, SMACH is independent of ROS, the *executive-smach* stack provides a ROS integration, including integration of the *actionlib* package described in 2.4.1, and Smach Viewer. Smach Viewer is a GUI for SMACH, it can visualize transitions between states, current active states, and values of data passed around, enabling easy introspection of SMACH state machines. The state classes provided by the library, support ROS protocols, means for passing user data between states is also provided. Upon building a SMACH state machine, consistency check of state transitions is provided by the library.

The SMACH core library provides two main interfaces, *State* and *Container*. *State* represents a state of execution, with potential outcomes defined prior to execution. A SMACH *Container* is a collection of one or more states, implementing some execution policy. Containers also define behaviour for dealing with preemption requests, this entails functionality for dealing with termination signals, and lets the system be canceled in a controlled fashion.

SMACH is in essence a library for creating state machines, and provides the *StateMachine* container. But SMACH also goes a little beyond this, providing other containers as well, *StateMachine* being the simplest one, while *Concurrence* container provides means for executing more than one state at the same time.

2.5.1 SMACH States

A formal state machine defines states as a system configuration, where the system is waiting for certain transitional criteria to be met, upon which the system executes specific actions related to the transition. A SMACH state differs from the formal state machine, here a state represents a local state of execution, states describe what the system is doing rather than the configurational state of the system. This in turn allows a shift in focus for the user, from defining transitional points between states to what the states are executing and the results of that particular execution. Moreover, SMACH state machines are also states, meaning they can have outcome of their own, this also makes it possible to build nested state machines.

All SMACH states must implement an *execute*-function, this is the main body of the state. When a state is activated, the *execute*-function of the state is called. The function blocks until the state is finished, upon which it returns an *outcome* the state machine can use to determine the transition leading to the next state. The state outcomes are a property of the state, and must be declared prior to execution. They define the interface to the SMACH container, and the outcome from a state execution must lead to a transition to the next state for the consistency of the system to hold. If needed, user data can be passed between states and state machines by specifying input and output keys prior to execution. Remapping of input and output keys can be done to specify how user data shall be passed.

Some ROS SMACH state classes are provided, including *State* and *SimpleActionState*. *State* is the state base, this class provides no outcomes.

SimpleActionState is used to link states to actionlib actions, this class also provides means for Goal generation and result processing callback, as well as some outcomes.

2.5.2 How to Create a Simple SMACH State Machine

The following code shows implementation of a state inheriting the *State* base class:

```
class SimpleState(smach.State):
    def __init__(self, outcomes =["outcome1","outcome2"],
                        input_keys=["input"],
                        output_keys=["output"])
        #state initialization
    def execute(self, userdata):
        #state execution
        if userdata.input == 1:
            return "outcome1"
        else:
            userdata.output = 2
            return "outcome2"
```

- State initialization is done in the *__init__*-function declared on line 1. The function should be non-blocking, outcomes for the state must be passed to this function.
- Input and output-keys for passing data to and from the state are specified on line 3 and 4.
- Behaviour of the state is implemented in the *execute*-function declared on line 6, this function can be blocking.
- The user data is accessed on line 8, and possibly modified on line 11.
- Based on the input value, outcome *outcome1* on line 9, or *outcome2* on line 12 is returned.

A *SMACH* state machine is created as follows:

```
sm = smach.StateMachine(outcomes=["outcome3","outcome4"])
sm.userdata.sm_variable = 0

with sm:
    smach.StateMachine.add("SIMPLESTATE1",Simplestate1(),
                    transitions={"outcome1":"SIMPLESTATE2",
                                 "outcome2":"outcome3"},
                      remapping={"state1input":"sm_variable",
                                 "state1output":"sm_variable",})

    smach.StateMachine.add("SIMPLESTATE2",Simplestate2(),
                    transitions={"outcome2":"SIMPLESTATE1",}
                      remapping={"state2input":"sm_variable",
                                 "state2output":"sm_variable"})

outcome = sm.execute()
```

- On line 1, statemachine *sm* with possible outcomes *outcome3* and *outcome4* is created.
- The *userdata*-variable *sm_variable* for *sm* is created on line 2.
- The container is opened on line 4. On line 5, the state *SIMPLESTATE1* is added to the container, common convention dictates state names in caps.
- The transitions of a state must specified when the state is added to a state machine container, this is done on line 6-7.
- On line 8 and 9, keys for *userdata* are remapped, this is not a requirement, but considered good practice as it avoids potential confusion between variables used in states and state machines.
- A second state is added on lines 10-13, before the state machine is executed on line 15.

The resulting state machine is visualized in Figure 2.10.

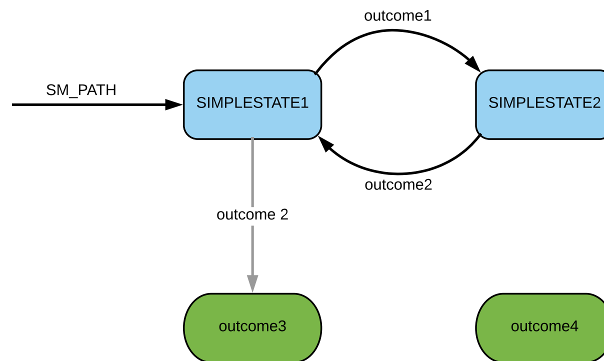


Figure 2.10: The figure illustrates the SMACH state machine created in 2.5.2. Transitions are indicated by the arrows. The blue and green boxes portray states and outcomes, respectively.

SMACH Sequence Container

SMACH also has libraries supporting the creation of sequential state machines, provided through the *Sequence* container [25]. The *Sequence* container is an extended version of the *StateMachine* container, where auto-generated transitions are added in order to create a sequence of states. The sequence is defined by the order of which states are added. The following code illustrates how a sequential state machine is implemented:

```
1 sq_sm = Sequence (outcomes, input_keys, output_keys,
    connector_outcome)
2
3 with sq_sm:
4     Sequence.add("SEQUENTIALSTATE1", SeqState1())
5     Sequence.add("SEQUENTIALSTATE2", SeqState2())
6     Sequence.add("SEQUENTIALSTATE3", SeqState3())
```

- The state machine is created on line 1. The connector outcome specifies the automatic transition, if standard transitions are provided in addition to the connector outcome, they can override the automatic transition.

- The container is opened on line 3, and the states added on lines 4-6.

2.6 Autonomous Navigation of Mobile Robots

Autonomous navigation of robots is one of the greater challenges in mobile robotics. In order for a robot to be self navigating, it needs to be able to accurately estimate its global position, and track its local position. More precisely, it needs to be able to determine its position in a map (localization), with no other info than the map it is in, and be able to keep track of its position after it has been localized.

2.6.1 Adaptive Monte Carlo Localization

One of the popular approaches to this challenge is the Adaptive Monte Carlo Localization (AMCL) [26]. AMCL is based on the simpler standard Monte Carlo Localization (MCL), which relies on a combination of grid-based Markov Localization and Kalman filtering based techniques [27]. In short, the algorithm tries to solve the problem of estimating the state of a system, given given environmental observations and control input. It is then improved by sampling the estimated state particles in an adaptive manner, giving us the AMCL approach (also called particle filter localization). For each particle, the probability of the sensed particle based on the position of the robot is calculated, and the more likely particles are chosen. The particles are then resampled and new state estimates calculated. If the particles converge to the actual position of the robot, localization is successful. If all particles converge to an erroneous state, or the robot is moved after the particles have converged, localization will fail.

Unfortunately, no actuator is perfect, when motion updates are received (result of control inputs), results will always start drifting to some extent, which in turn leads to diverging particles. Sensory updates on the other hand, by way of the algorithm, will hopefully lead to converging particles.

Proper calibration of odometry will help mitigating drifting results from motion updates, while a higher number of samples can result in more precise localization. But even though ramping up the number of samples might be tempting, setting the number of samples too high will also result in poor localization. Several reasons exist for why this might happen, the most obvious culprit being limited computing powers. As the number of samples increase, so does the computational load, and in turn the time needed to compute the correct pose. For a mobile robot, this might mean that the robot has moved considerably while new localization data is being computed, and the data might be outdated before it was even made available, since the robot now has moved to a different pose than the computed one. In the case of the Pioneer LX robot, the localization task tries to localize at 10 Hz, this means that if the localization task takes more than 100 ms, localization might start to suffer.

The following list contains specifications relevant for proper tuning of navigation:

- The pose estimate is provided to software along with other robot state information every 100ms.
- The laser scanner provides 500 readings in a 250° field of view, with a typical range of about 15m. It operates in a single plane, located 191mm off the ground.
- Maximum translational speed is 1800 mm/s.
- Maximum rotational speed is 300 mm/s.
- The analog gyroscope provides accurate readings up to 320 °/s.

2.7 Software Structure of the Cyborg

The Cyborg consists of several software modules, a simplified class diagram showing the structure at the onset of this thesis is provide in Figure 2.11. Modules `cyborg_controller`, `cyborg_navigation`, `cyborg_conversation`,

`cyborg_idle`, `cyborg_music`, and `cyborg_text_to_speech` are all implemented by previous students.

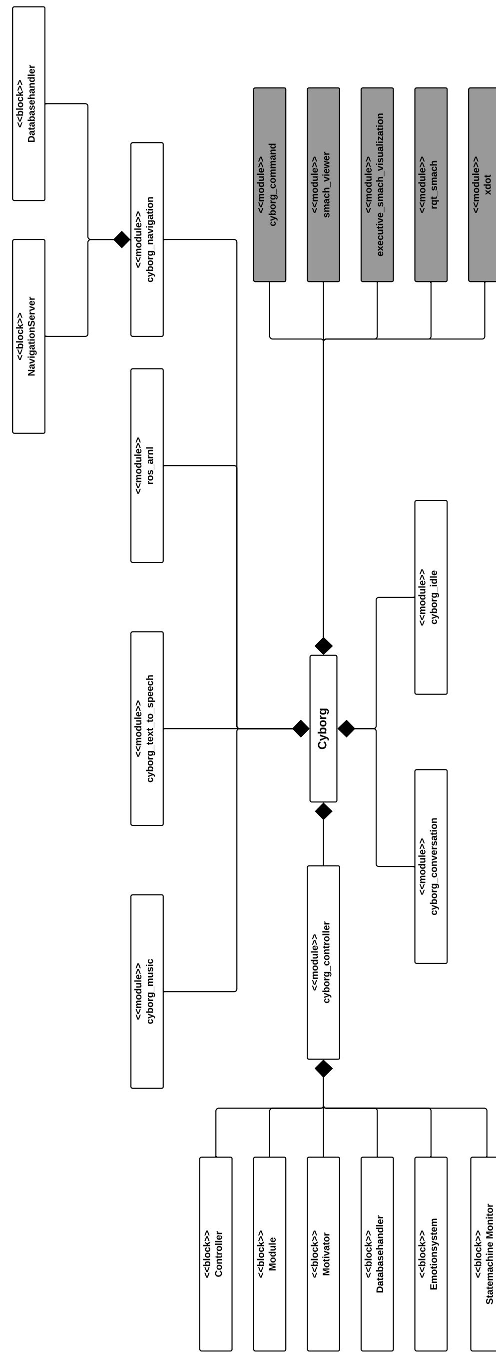


Figure 2.11: Simplified class diagram for the Cyborg at the onset of this thesis.

2.7.1 Controller

The controller is the main node of the Cyborg [28][9], responsible for coordinating all other nodes. It implements a state machine that organizes all actions (ROS actionlib servers) into states for the Cyborg. It also features an emotion system based on a PAD emotion state model, responsible for handling the emotional state of the Cyborg, and in addition a motivator responsible for motivating the Cyborg to execute various behaviors when no external or scheduled events are available. The inner workings of the controller module is seen in Figure 2.12, behavior modules are used to supply the Cyborg with various functionality.

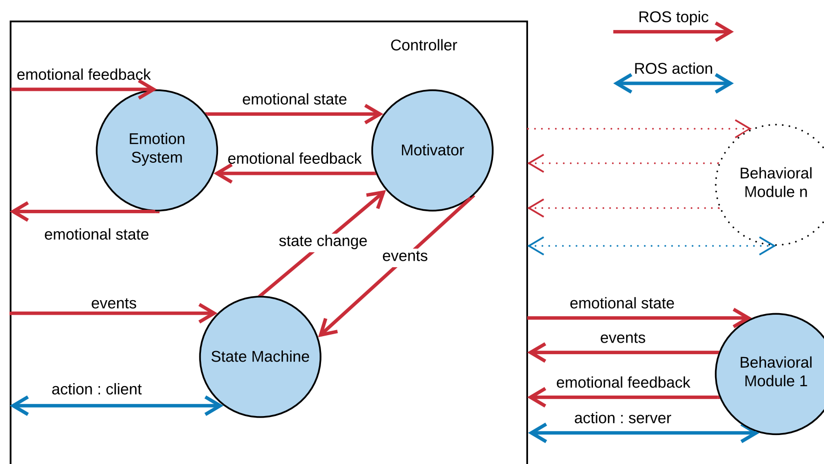


Figure 2.12: Inner workings of the Cyborg controller module. Behavior modules supply the Cyborg with various functionality.

2.7.2 Navigation

The navigation module implements navigational functionality for the Cyborg, through the use of the `ros_arnl` module. The functionality is provided through four action servers, `server_planning`, `server_moving`, `server_talking`, and `server_go_to`. In addition, the navigation module contains a

scheduler function, responsible for finding and publishing scheduled navigational events to the Cyborg controller.

2.7.3 LED Dome

The software for the LED dome consists of two different parts, the LED dome ROS module and the interface for the LED controller. The ROS module for the LED dome is implemented as a SMACH state machine, subscribing to a topic over which commands are sent from other Cyborg modules. State diagram for the ROS module is presented in Figure 2.13, a simplified class diagram for the same module is seen in Figure 2.14.

2.8 Our Long Term Vision

As stated in 1.1, the long term vision is to create an interactive and social cyborg. The Cyborg project aims to achieve this goal through the tasks presented at the NTNU Cyborg website [29]. A summary of the tasks presented on the NTNU Cyborg website relevant for this thesis is as follows:

ROS-based statemachine - This project entails working with the Pioneer LX robot and the ROS based state-machine. The state-machine is already well developed, the task now is to get everything up and running so the robot can operate autonomously in Glassgården. This project aims to finish this work into our final product. The project also entails working with the onboard Arduino mode selector which enables choose the operation mode of the robot (state-machine, manual operation etc.).

LED head animation - In this project you will work on the robots LED head/dome which displays animations on the robot. The LED module should communicate with the ROS state-machine described above.

In addition, some of the more specific tasks presented on the NTNU Cyborg Wiki [30], are as follows:

- Finish the ROS-based state-machine, and tie all the different software modules together.
- Create LED visualizations.
- Improve the startup box and script.
- Paint the body.
- Install fan inside body.
- Mount the LED head properly.
- Finishing touches on the body to make it look nice.
- Using rqt available in ROS, develop an interface for monitoring and controlling the robot externally (e.g via a veb browser).
- Update and improve this wiki.

2.9 Terms and Definitions

This thesis continues the use of definitions specified in Andersen thesis for the Controller Module [28], the definitions are presented below for the sake of convenience, albeit with some slight changes, taking into account recent upgrades on the Cyborg.

Module: Hardware or software that provides some functionality/feature for the Cyborg through a ROS node.

Input Module: ROS node that takes input from hardware and makes the data (in some form) available for the ROS network, i.e., for other ROS nodes. An example is a ROS node that takes sound input from the microphone and publishes the text on the ROS network.

Output Module: ROS node that ”directly” controls the hardware (through drivers) (and often makes it accessible for the other ROS nodes). An example is a node that takes in text and provides a voice output through the speakers.

Behavior Module: ROS node that provides some functionality, adds specific behavior to the Cyborg, e.g., the Selfie module (now obsolete). It may for example receive data from an input module, and then tell an output module to do something, but it does not directly interact with the hardware (drivers).

Event: Something, often external, that has happened, been detected by a behavior module and that the module wants to act on, e.g., a user rising an arm (signaling that the user wants the Cyborg to follow).

In addition, the following terms are specified:

Mode of operation: Modes of operation for the Cyborg, defined by which modules are executed, e.g., autonomous mode, where the state machine for the autonomous cyborg is in control, or manual operation where the Cyborg is controlled either with a joystick or through a wireless interface like MobileEyes.

Behavior: Some specific functionality, for example playing sounds through the speakers.

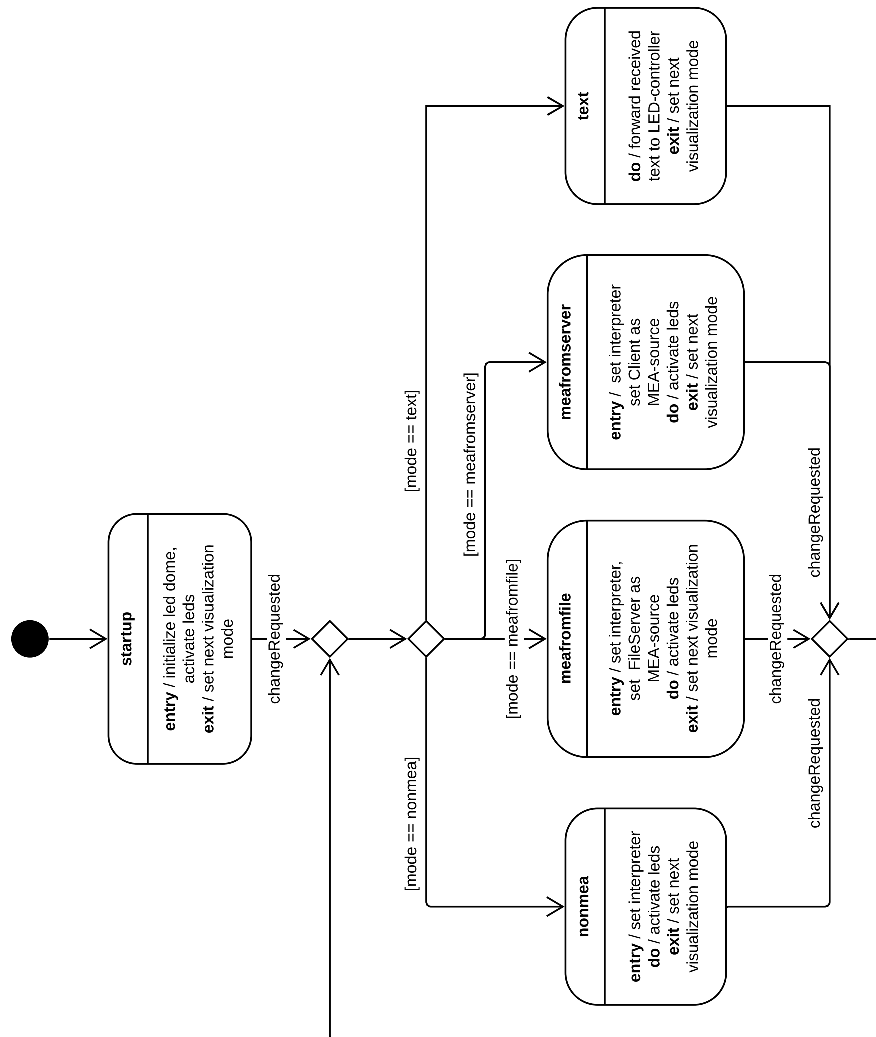


Figure 2.13: State diagram for the Cyborg LED dome ROS module.

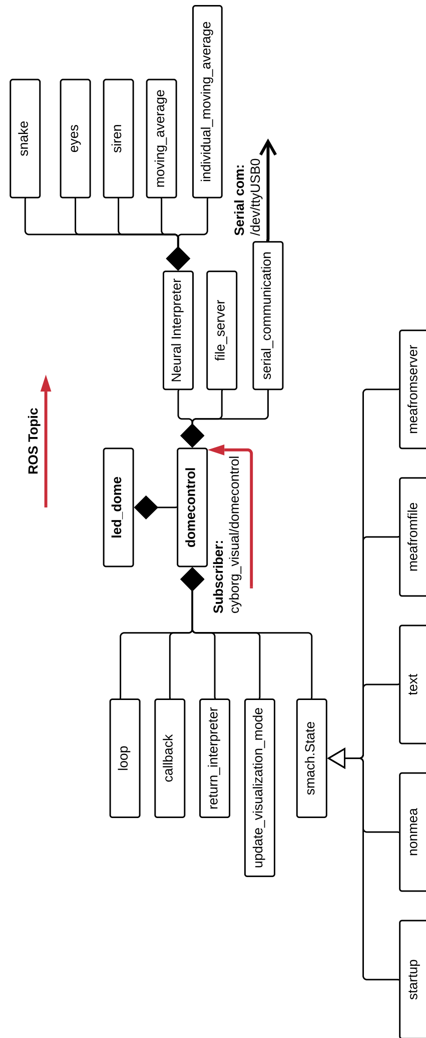


Figure 2.14: Simplified class diagram for the LED dome ROS module.

Chapter 3

Evaluating the Cyborg

3.1 Introduction

This chapter aims to present an evaluation of the Cyborg, and the various tasks deemed necessary in order to bring the Cyborg closer to the short and long term goal. The tasks presented here are in direct correlation with the vision presented in 2.8, some tasks also emerged as a result of work and testing that was done on the Cyborg, and were appended to the list when the obstacle manifested. The software evaluation is a continuation of the proposed path presented in the authors specialization project [1], with emphasis on simplifying and improving the overall software structure of the Cyborg.

3.2 The Cyborg v3.0

In his thesis for the Cyborg Waløen presents **The Cyborg v2.0**[9]. Once again the Cyborg has undergone some big structural changes, the new and probably last visual design iteration is completed at the end of this thesis, the hardware architecture is changed, and the software structure of the Cyborg is reworked. This motivates the new version of the Cyborg, **The Cyborg v3.0**.

In order to achieve the goal of the Cyborg V3.0, it is first necessary to define what the Cyborg must be able to do. The following specifications have been defined:

- The Cyborg must be able to roam around autonomously and of its own will.
- The Cyborg must be able to visualize biological neural activity on the LED dome, and in addition show text and animations.
- The Cyborg must be able to speak and play sounds.
- The Cyborg must be easy to operate and support mode changes on the fly.

In order to satisfy the goal of the Cyborg V3.0, the following tasks have been selected as critical:

- The LED dome controller and ROS node must be fully integrated into the Cyborg.
- The existing Cyborg ROS modules must be properly integrated.
- The Mode Selector box must be upgraded in order to facilitate mode changes without having to restart the Cyborg base.
- The LED dome must be finished and integrated into the Cyborg.
- The Cyborg body must be finished and properly mounted.
- The Cyborg must be painted, and the end result aesthetically pleasing.
- Cooling fan must be installed inside Cyborg body.
- All other hardware must be properly mounted, not able to move when the Cyborg is operational.

Tasks 4-8 are done in collaboration with EiT group 3.

3.2.1 Lack of Integration

When reading through earlier reports dealing with the Cyborg, lack of integration and a wish to remedy the challenge is mentioned and can be seen several times:

The NTNU Cyborg: Robot Hardware Infrastructure [31] - *"The general problem for the NTNU Cyborg project has been lack of integration of all its parts, and this has been my main focus."*

Controller Module for the NTNU Cyborg [28] - *"The short term goal for the NTNU Cyborg is to have a robot prototype up and running by the summer of 2017."*

The NTNU Cyborg v2.0: The Presentable Cyborg [9] - *"focused on making an integrated solution..." "project difficult to enter..." "lack of structure..." "lack of of a good, reliable foundation that all smaller parts of the project could be built around..."*

Control System and Object Detection System for the NTNU Cyborg [32] - This thesis presents implementation of a decision-tree based alternative to the state machine that is already implemented.

The second thesis in the list mentions the goal of having a robot prototype up and running by summer of 2017, although this goal was not properly met, the ground work for a proper system structure was laid down. The third thesis still mentions many of the same challenges as the first thesis. While the fourth thesis mentioned tries to circumvent parts of the problem, by avoiding it all together. The author mentions challenges regarding the complexity of growing state machines, and the reuse of states with tightly coupled transitions.

The development of the Cyborg is driven forward by students, mastering a new and complex system is time consuming and not a trivial task.

3.3 Overall Software Structure

While working on my specialization project fall 2018, it became apparent that a simplification of the software structure could potentially benefit the overall project, this is further motivated by 3.2.1. After conferring with assisting supervisor Martinius, it was decided that a rework of the software structure of

the Cyborg was the right choice. In authors opinion, there are two challenges that should be addressed. An overall complex software structure, and the complexity of the implemented states. The state machine is hard to configure, and implementing new states is time consuming and complex. Several of the modules currently implemented handle a great variety of tasks, and input and output is many cases directly paired with states for the state machine.

3.4 Proposing a new Software Structure

In order to achieve the goal of a less complex software structure, current modules are going to be decomposed, isolating and cutting back on the tasks handled. States will be reworked in the same manner, isolating input and output in separate modules. States with coupled output makes it easy to add new behavior as nested states in more complex states. But by providing output in separate modules, and in turn use those modules by states, we obtain a more modular design that is easier to maintain, while still enabling nested states in the same fashion as before.

The reworked software structure with isolated output modules motivates a common interface for the most used output modules, making them easier to use, not having to worry about conflicting commands. Expanding on this idea and providing an easy way of making new behavioral presets, without having to implement a new action server state each time, will make it easier and less time consuming to configure new behaviors for the Cyborg state machine.

The proposed software structure is presented in Figure 3.1. As before, the state machine in the Cyborg *Controller* connects to the behavior modules through *Module*. *Behavior* is the proposed common interface module, that connects to the output modules, while also providing simple behavioral presets. For more complex behaviors than provided by *Behavior*, a separate *State* module is used between *Module* and *Behavior*. The choice of the optional module is done for two reasons, it lets us keep the behavioral presets simple,

and it lets us use *Module* and states provided through action servers as before, meaning that an already implemented core concept does not have to be switched out. For states not requiring any output modules, ex. a planning state where the only goal is to chose the next state, all modules below *State* are omitted.

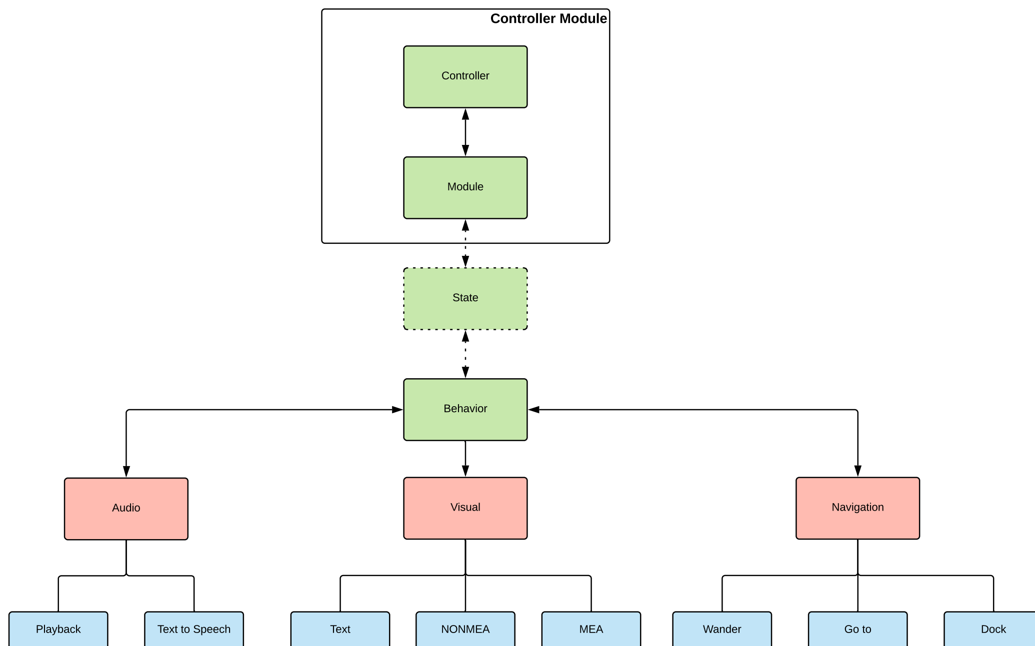


Figure 3.1: A proposed software structure for the Cyborg. Red components indicate output modules, blue indicate functions provided by the output module. Dashed component and lines indicate optional module.

A class diagram for the new software structure is seen in Figure 3.2.

3.5 Navigation

Although autonomous navigation already should be fully implemented on the Cyborg, it has not been properly tested and tuned, which only became apparent when testing for this thesis first began. The preliminary testing

brought several issues to the table, and in addition some deficiencies were identified.

3.5.1 Preliminary Testing Issues

When simulating the Cyborg, it does not dock properly, testing in real life is necessary in order to see if the problem exists as in the simulator. In addition, the Cyborg would not follow a path in the simulator, the path was properly calculated, but instead of driving along the path, the Cyborg was only rotating from side to side. This turned out to be an issue with the sonar, which has to be disabled when simulating, in order to get the Cyborg to follow a path. Simulating the Cyborg also masked some problems, which of course turned up when the navigation was tested on the Cyborg itself, this has probably contributed to the lack of information on the issues faced. The first issue that manifested itself when commanding the Cyborg around, was the lack of computing power needed in order to satisfy navigation. The software is provided with default values for the navigational parameters, which should suit the robot in question perfectly as they are set by the manufacturer itself. Unfortunately, with the default values for the navigational parameters, the `ros_arnl` node is unable to execute in a satisfactory fashion, repeatedly throwing messages about the localization task taking too long.

When testing the autonomous navigation capabilities of the Cyborg in Glassgården, it exhibited an erratic behavior, often stopping abruptly, standing still for a couple of seconds, maybe rotating a little, before it eventually continued along its path. This turned out to be because of the sonar, which was automatically disabled and reenabled, before the Cyborg continued moving. The slate tiled floor is not an ideal surface for the sonar, the joints between tiles are sometimes identified an obstacles right in front of the Cyborg, which makes it stop. Another issue with the floor in Glassgården is the sun glare, when the Cyborg drives from a shaded area to an area with strong sun light, the transition line on the tiles is seen as an obstacle.

Then there is the issue of the closure of Mobile Robots, mentioned in my specialization project [1]. The Cyborg base is heavily dependant on software provided by a company which no longer exists, and it is probably only a question of time before this will be a growing issue. The provided software is also rather restrictive compared to the alternative navigation stacks presented in the navigation guide on the ROS wiki[33], with less options for customization. Given the fact that the software is provided by a commercial party and should be working as is this is not surprising, and if the solution had been up to par, providing proper working autonomous navigation, this would not have been an issue. Unfortunately, the lack of options makes it harder to tune the navigation to a satisfactory degree.

The last issue is regarding recovery of the Cyborg when it loses localization or gets stuck, which can and will happen, for example because of the highly dynamic environment it is navigating in. If localization is lost, the Cyborg will not be able to move again without external intervention. It would be beneficial to implement recovery behavior. This can for example be achieved by backtracking and revisit previously visited poses, while trying to reinitialize localization. Or by trying to initialize localization with a set of different poses scattered across the map, or around points where localization is more likely to fail.

3.5.2 Navigation Module

The navigation module is one of the main modules that need to be reworked, stripping away all behavioral functionality, keeping only the parts needed for proper connection to the `ros-arnl` node. This entails removing all "navigation talking" states from the module, as well as the *scheduler* function. The navigation modules also lacks the ability to dock the robot, this has to be implemented in order to enable the Cyborg state machine to recharge at night or when the battery is depleted.

3.5.3 Proposed Task List

The proposed task list for the navigation module is as follows:

- Rewrite navigation module, stripping away behavioral functionality.
- Implement docking.
- Implement recovery functionality.
- Adjust sonar parameters in order to avoid erratic behavior.
- Tune overall navigational parameters, with regard to available computing power and accuracy.
- Change navigation stack on the Cyborg, removing the need for legacy software provided by Mobile Robots.

3.6 Audio

The Cyborg can play sounds through the speakers on the Cyborg base, both playback and text to speech is implemented. Playback is handled through an action server state called music, which is activated by the state machine, while text to speech is implemented in a separate output module, which makes itself available for the other modules through a ROS topic. The playback module mixes behavior and output functionality, motivating a reimplementation. It also lacks the ability to play different sound clips, loading only one hard coded filename.

The text to speech module works, but execution is not preemptive. In addition, the quality of the text synthesizer is rather poor, it lacks proper options for customization, and quite often it is impossible to actually understand what is being said.

3.6.1 Proposed Task List

The following tasks are proposed regarding audio:

- Rewrite the playback module, stripping away behavioral functionality, with possibility to play different sound clips.
- Investigate alternatives to the text synthesizer that is used for text to speech.
- Rewrite the text to speech module, and add the ability to preempt execution.
- Combine playback and text to speech in a separate audio module, with a common interface for both.

3.7 Controller Node

The state machine for the Cyborg is implemented in the controller module, in order to avoid unnecessary clutter in the controller module, it would be preferable if states used multiple times were configured separately and imported where needed.

The state machine does not want to execute, complaining about that the nested states do not have the proper input and output keys available, this is probably due to a recent upgrade in the *SMACH* library. The state machine has to be reimplemented anyway, taking into account changes done to the other modules, presenting a natural opportunity to address this issue. The emotionsystem, motivator, and database will be kept as is.

3.7.1 Proposed Task List

The following task list is stated:

- Relocate state machine configurations.
- Reimplement the state machine in accordance with new system structure.

3.8 LED Dome

Resuming the work presented in my specialization project fall 2018, some tasks still remain before both hardware and software is ready for integration into the Cyborg. A context diagram for the LED dome is presented in Figure 3.3. The dual-core version of the interface for the LED controller does not support both text and other visualizations in the same version, and only a small horizontal part of the LED dome is used for text animations. Since data is passed between tasks on separate cores on the LED controller, special care must be taken regarding memory protection. In addition, the prototype circuit needs to be realized and integrated into the Cyborg.

3.8.1 Proposed Task List

A list of the tasks necessary to fully integrate the LED dome onto the Cyborg is presented below:

- Implement the prototype LED controller on a stripboard.
- Frost the translucent plastic dome that is going to be mounted over the LED-dome.
- Integrate the LED-head hardware and software into the Cyborg.
- Improve concurrency handling and memory protection in the dual-core version of the ESP32 software.
- Implement text animations in the dual-core version of the ESP32 software.
- Remap the whole LED-dome for text animations.

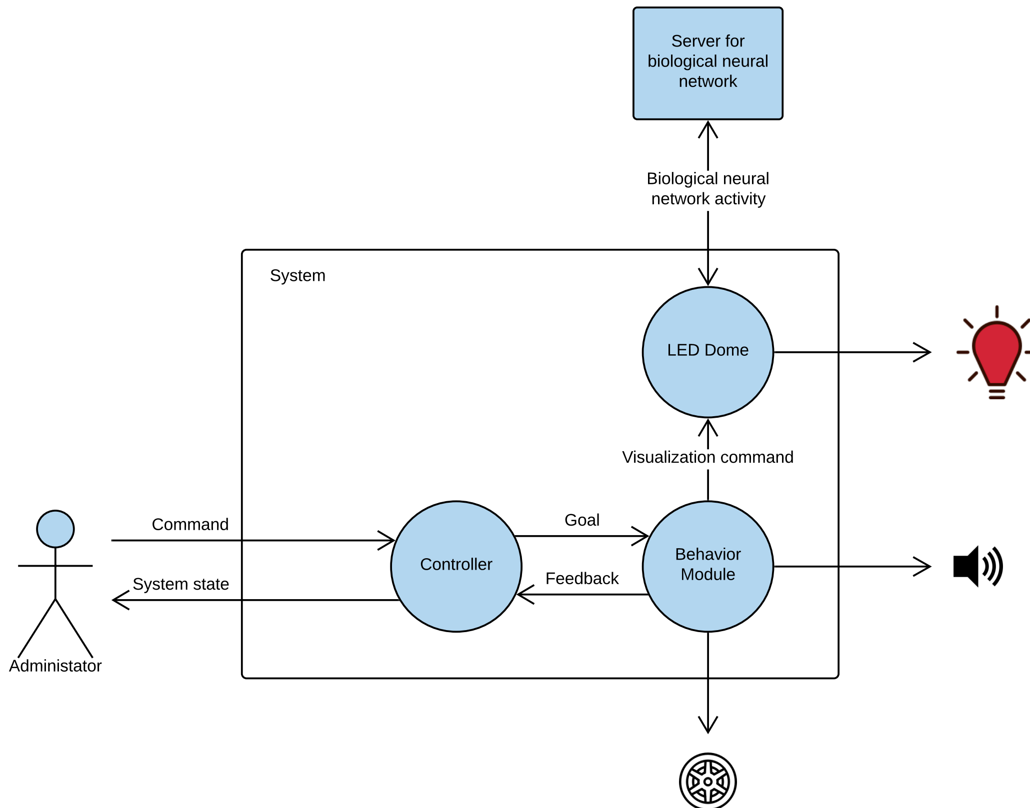


Figure 3.3: Context diagram for the Cyborg LED dome.

3.9 Discussion

The software evaluation tries to address the complexity of the Cyborg software and make it easier to configure new behaviors for the state machine. The proposed behavior server enables us do just that, without removing the option for more advanced programmers to implement complex states. As mentioned earlier, more advanced states can also be built from multiple simple states, which aids modularity and reusability. Evaluation of the navigation stack and a proposed change, as well as a change to ROS 2 for the whole Cyborg was not prioritized.

3.10 Conclusion

The Cyborg v3.0 has been defined, and the different tasks deemed necessary in order to bring the Cyborg to a state where it cover the specifications of the Cyborg v3.0 and is ready for demonstration have been evaluated and stated. An evaluation of the Cyborg software has been motivated and presented, and tasks for the different software modules stated. The evaluation is done with regards to making the Cyborg easier to work with. A behavior module is proposed, the module offers an easy way to configure state presets and acts as a single interface for commonly used output modules.

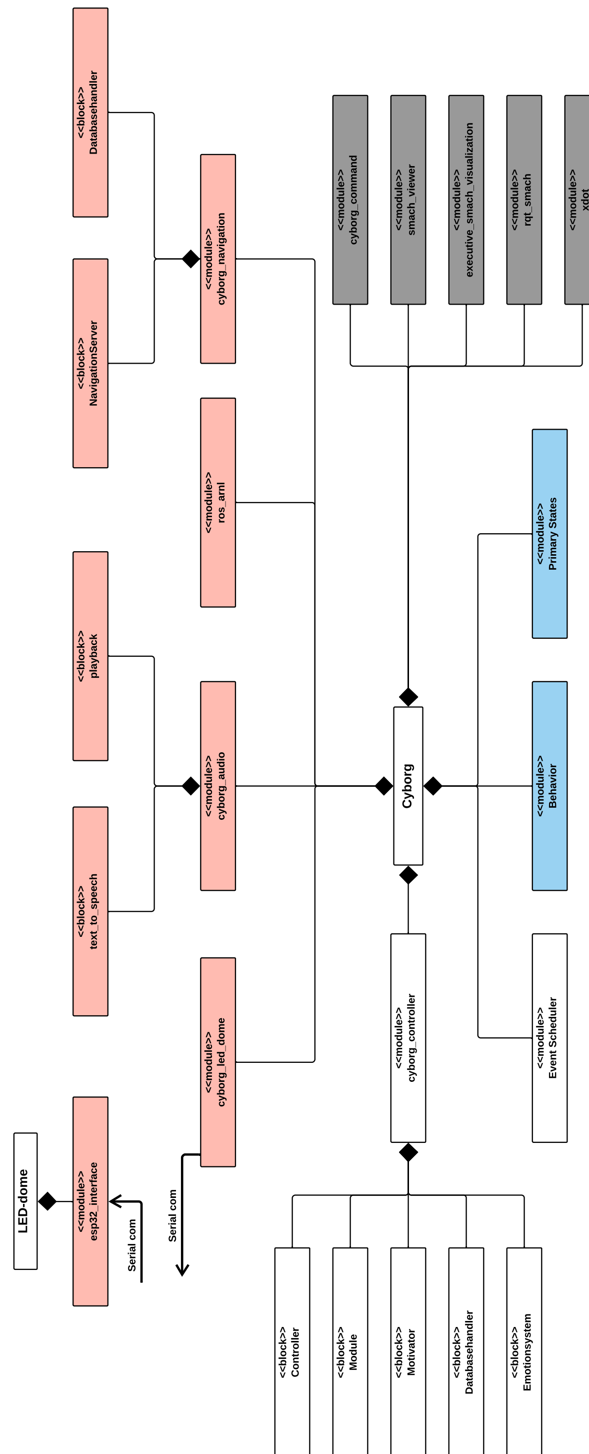


Figure 3.2: A simplified class diagram for the new structure of the Cyborg. Components related to output are marked with red, while grey indicates modules pertaining to remote access and monitoring.

Chapter 4

Assisting Experts in Team Groups

4.1 Introduction

As a part of this thesis, I have chosen to assist and provide guidance for the different groups from Experts in Team that were involved with the Cyborg. As I already had previous experience with the Cyborg, co-supervisor Martinius also requested that I should present some of the tasks relevant for getting it to a state where it is ready for demonstration. Throughout the semester I have made myself available on *Slack*, and attended EiT workdays both when requested and at my own initiative to check up. And where I had relevant experience, I have tried to assess their proposed solutions and offer my advice when requested.

4.2 Group 1 - Simple Neural Response Interpreter - SiNRI

Group 1 chose to work with the neural interface for the Cyborg, their work is presented in project report [34]. Their plans incorporate a sonar sensor that should be used to detect and notify the Cyborg state machine when people are standing in front of the Cyborg. At the start of the semester, the group requested some assistance regarding ROS and the overall software structure of the Cyborg. I have explained and shown them the basics on how to design a ROS module that can notify the Cyborg state machine about occurring events. In addition to this, I have set up a computer with ROS at their disposal at the Cyborg office. Although the sonar will probably not be used, their work has added incremental updates to the neural interface for the Cyborg.

4.3 Group 3

This group aimed at finishing the Cyborg body and mounting the LED dome, their work is presented in project report [35]. Their original plans also entailed creating more animations for the LED dome, but this idea was discarded after a meeting where I explained briefly what had to be done on the Cyborg body, warning them about taking on more tasks than they could handle in the given time frame, and that their aim should be to deliver a completely finished product. The group had no previous experience regarding spackling, sanding, and painting related to the bodywork, and assistance was requested several times during the semester. In addition to this, co-supervisor Martinius instructed me that I was in charge of evaluating their work related to the body and LED dome, and check if it was of acceptable quality. As the Cyborg will garner a lot of attention, it is instrumental that it looks the part.

4.3.1 Frosting and Assembling the LED Dome

In order to prepare the LED dome for integration into the Cyborg, the outer plastic shell needed to be frosted. The group tested multiple solutions on how to frost the outer shell on some leftover pieces of plastic, and opted to use a frosting spray. Before they used the frosting on the outer shell I checked on their frosting experiments, and as I was not particularly pleased with the result they had attained, I conducted my own wet sanding tests. I also performed some scratch tests on the frost spray using only a fingernail, and it was evident that the result was not very durable. I then voiced my concerns about the frosting spray with the group, and showed them the result with the wet sanding test, proposing they should use this solution instead. The group opted to use the frosting spray, but as the end result was deemed unacceptable, they ended up wet sanding over the frosting spray. The outer shell is more translucent than was expected, and a better end result can probably be attained with some more effort.

Figure 4.1: Preparing the LED dome and outer shell for the dome fasteners.

I also assisted with drilling holes in the outer shell and the LED dome for the dome fasteners, and mounting these pieces together. As it is instrumental that the holes align properly, we assembled the pieces on a metal table before drilling, exploiting the magnets in the dome fasteners in order to keep everything from moving. A picture taken during this process is seen in Figure 4.1.

4.3.2 Sanding and Painting the Body

Regarding the spackling and sanding of the body, I offered some advice on how to get a proper result as I have done similar work in the past. Regretfully, I did not show them how to actually apply the spackle and how to determine if the surface was smooth enough after sanding. This resulted in two rounds of spackle and sanding before the body was ready for painting, as it was quickly discovered that the body looked less smooth after a couple of layers of primer was applied after the first round. After the two rounds, the body looked smooth and they body and Cyborg base was painted. I checked up on the paint work several times, and the end result was deemed acceptable, although the paint that was used is rather thin, resulting in some faint glare from the red parts of the Cyborg base. Overall though the body looks good, and after the decals were applied it looks even better.

4.4 Discussion and Conclusion

The contributions from EiT group 1 and 3 have benefited the project greatly, resulting in new software for the neural link, and an almost assembled Cyborg.

The Cyborg body looks nice, and only some small parts are still missing. Group 3 installed a fan which needs protective covers, and a gap between the LED dome assembly and the Cyborg body needs to be addressed. In addition, the two 3D printed body parts of the Cyborg do not align properly and the backside tends to slide off when operating the Cyborg, which also needs to be addressed before the work on the body is complete. In hindsight, firmer boundaries for quality should have been set right from the beginning, and in addition I should have checked on some of group 3 solutions before letting them proceed, as this could have saved them some time and effort. It is hard to know how much to intervene with someone elses work, but I could potentially have saved them some time if I had asked about their knowledge on the subjects at hand.

All in all, both groups have contributed greatly, bringing the Cyborg one step closer to the overall goal.

Chapter 5

The Mode Selector Box

5.1 Introduction

At the onset of the thesis, co-supervisor Martinius expressed a wish about being able to change modes directly on the **Startup Box**. The **Startup Box** is programmed in such a way that its only possible to select a mode when powering on the Cyborg, if one wants to change modes, a restart of the whole Cyborg is needed, wasting time and making the Cyborg harder to operate than necessary. There is many scenarios in which easy switching between modes of operation for the Cyborg is desirable. For example, when demoing the Cyborg, the *Aria Demo* might be used to drive the Cyborg with the joystick to the destination, before an autonomous mode is selected for the demo itself. Another issue that emerged when testing the Cyborg base in Glassgården, was that the *Aria Demo* was crashing quite often, an easy way of restarting modes on the fly would make such issues less cumbersome. Since the modifications add functionality outside the scope of the original name, a rebranding of the component is deemed to be in order, hence the new name, **Mode Selector**.

5.2 Software Modifications

When powering on the Cyborg, **Mode Selector** waits for a ready signal from the cyborg base. Once the ready-signal is received, the available modes are presented on the screen and the user is prompted for input. When a valid sequence is chosen, the display tells which sequence is chosen, as well as an option to stop the sequence. If the running sequence is stopped, the user is once again prompted to chose a new sequence. A sequence diagram showing the interaction between the Mode Selector box and the Cyborg base is presented in Figure 5.1.

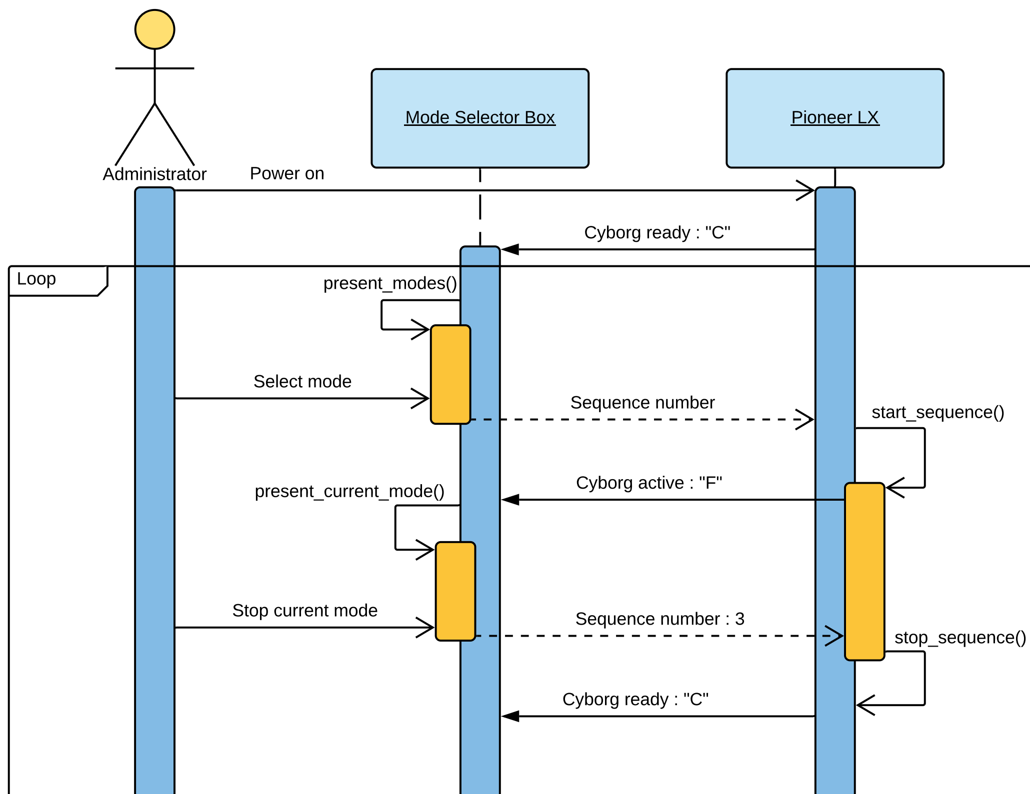


Figure 5.1: Sequence diagram for the Mode Selector box and Cyborg base.

5.2.1 Cyborg Base

start.py

This script initiates communication between the cyborg base and the **Mode Selector**, parses the input, and executes the chosen sequence. The original version of this script contained one loop, which initiated serial communication, and when a valid sequence input was received, the loop was terminated and the sequence executed. The modified version is made up of three loops, one indefinite outer loop, containing two more loops. The first inner loop handles execution of sequences, while the second inner loop handles termination of the active sequence. A activity diagram showing control flow for both the original and modified version of the script is seen in Figure 5.2.

sequences.py

In the modified version of this script, the sequences in function *start_sequence* are modified to return the number of bash instances spawned. An exception is made for the sequence for **Aria Demo**, which returns "-1". In addition, a second function, `stop_sequence` is added, which handles termination of sequences. For all other sequences than the **Aria Demo**, it terminates the proper number of bash instances, by searching for the pid(s) of the latest bash instance(s), and sending a *SIGTERM* signal. While for the **Aria Demo** sequence, the shutdown sequence script `shutdown_aria.sh` is ran.

5.2.2 Mode Selector

The software for the **Mode Selector** is modified to accommodate the changes made on the python scripts running on the Cyborg base.

main.c:

The portion of code responsible for presenting the menu on the display has been moved to `cyborg_menus.c`. Handling of the running marker has been implemented, and in addition presentation of the currently running mode on the display, together with the associated option to terminate.

cyborg_menus.c:

Two new functions have been added, `present_modes` and `present_current_mode`.

present_modes - Contains the portion of code removed from `main.c` responsible for presenting the menu.

present_current_mode - Presents the running sequence, together with the option to terminate.

5.3 Scripts

Shutdown ARIA

The **Aria Demo**-mode differs from the other modes available, as it is ran as an application, which can be shut down with a "SIGINT" signal. When shutting down **Aria Demo**, the **Shutdown ARIA** script is ran. This script runs the following sequence:

```
rosnode kill -a
killall -SIGINT demo
```

Where the first command kills all ROS-nodes, and the second one signals the demo application

5.4 How to add new Modes

Some changes apply to how new modes of operation are added.

sequence.py - New sequences are added here as before. In addition, the number of bash instances that are spawned when executing said sequence, need to be returned after the sequences are executed.

cyborg_menus.c - The name of the new sequence is swapped with the desired sequence in function `present_modes`. A new case for the switch in function `present_current_mode` is added, along with the associated name and number.

The software is compiled and uploaded as described in [9].

5.5 Discussion

When assisting supervisor Martinius first requested the ability to change modes on the Cyborg without having to restart it every time, he did not know the original Startup Box well enough to comment on if it was feasible regarding the implementation.

More time than originally planned was spent on this task, but the task was deemed important by the author of this thesis, and in the end it was time well spent. With the Mode Selector it is now possible to stop the current mode and select a new one. The upgraded component has been used extensively, every time the Cyborg has to be restarted because of crashing modules while testing, or for every mode change, eg. when transporting the Cyborg, the new functionality enables us to save roughly two minutes. While this might not seem like a lot of time, it quickly adds up to a considerable amount when working with the Cyborg. It also adds to ease of use for the Cyborg, by virtually eliminating the need to use the power buttons on the Cyborg base, considering that Cyborg should be on all the time when it is used more actively.

5.6 Conclusion

To summarize, the software on both ends for the Mode Selector has been modified, adding the ability to shut down the currently running mode on the Cyborg and selecting a new one. The Mode Selector works as expected when tested, the new features make the Cyborg easier to work with since less buttons are used actively, and will result in saved time for every future participant working with the Cyborg.

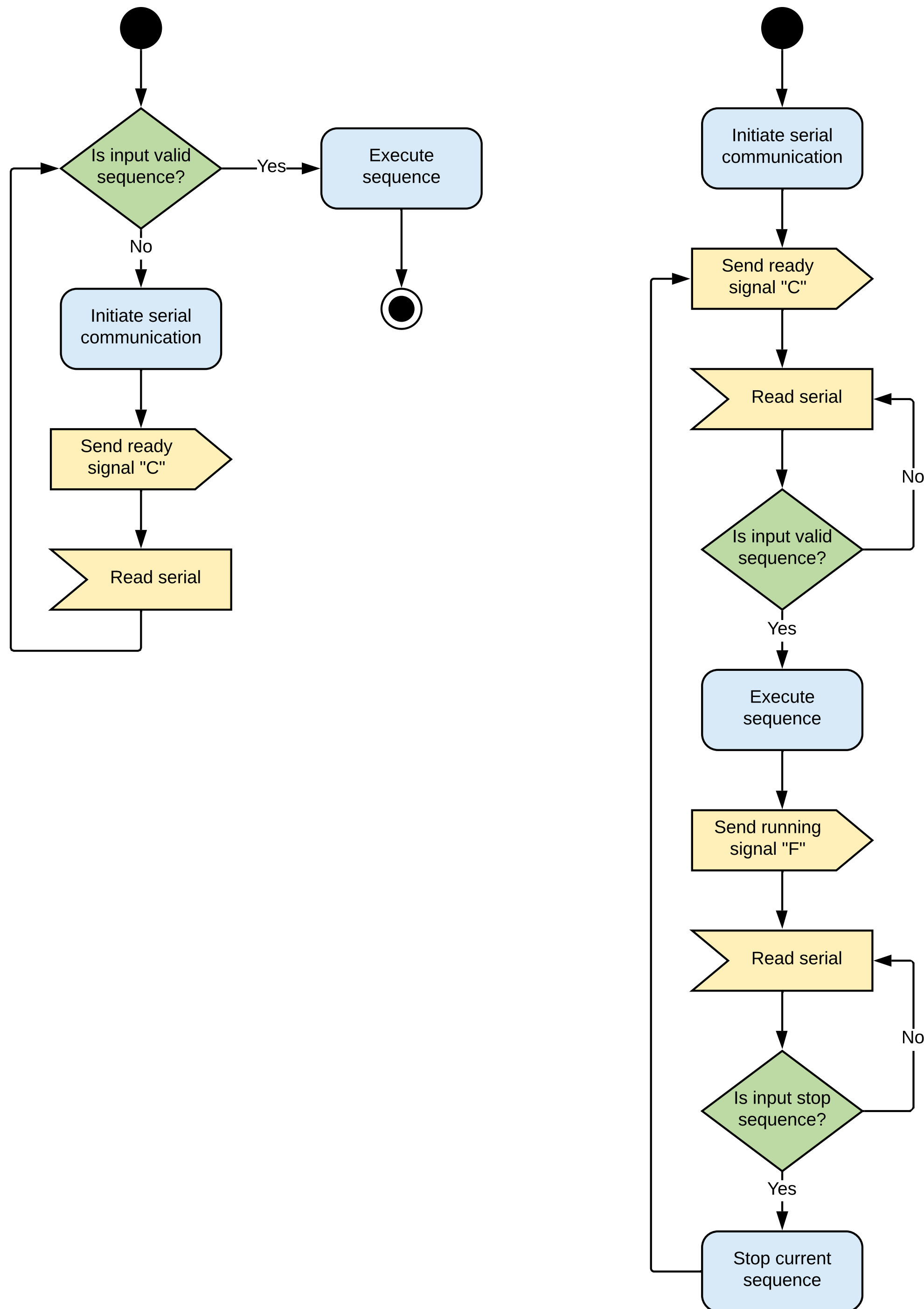


Figure 5.2: Activity diagram for the start.py script. Original version on the left, modified vesion on the right.

Chapter 6

The Behavior Module

6.1 Introduction

The purpose of the behavior module is to serve as a common interface for the most used output modules, and in addition provide an easy way to make and execute new behavioral presets and simple states. The behavior module is intended to be interfaced in the same way as the action server states already implemented, to avoid heavy modifications to the controller module.

In order to cover the role of a complete action server state for the Cyborg, the module must be able to exploit the emotional state of the Cyborg to choose which behavior to execute in the active state, and provide emotional feedback while doing so. It should also implement a way to change audible and visual modes in certain states, facilitating dynamic behavior while a state is active. A context diagram for the behavior module is seen in Figure 6.1.

6.2 Requirements and Specifications

Based on the system structure and evaluation presented in 3, the following requirements and specifications have been selected:

- The module must be implemented in ROS, as a ROS node.

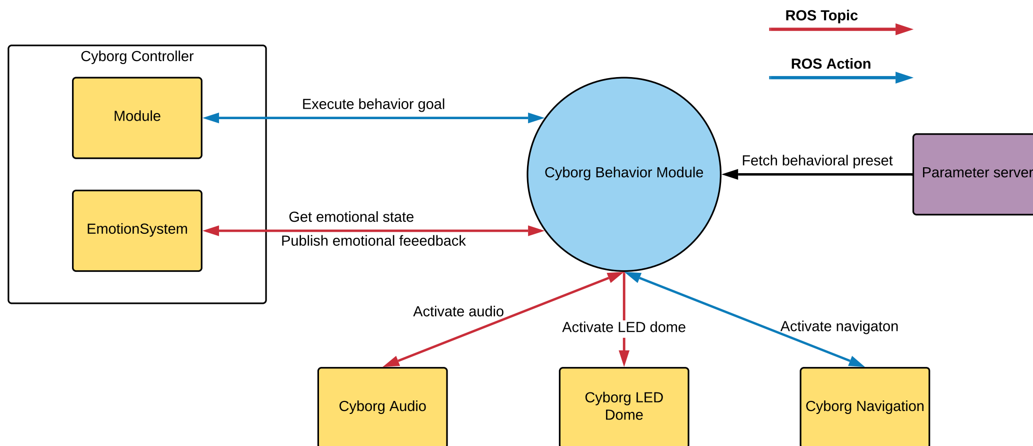


Figure 6.1: Context diagram for the behavior module.

- All communication with other ROS nodes must be through ROS protocols.
- The module must offer its service as an action server.
- The module should have a database for all the behavioral presets.
- The module must take advantage of the Cyborgs emotional state, and provide emotional feedback.
- The module must interface the audio module, led_dome module, and the navigation module.
- The module must provide a way to configure what triggers the completion of a state.
- The module must provide a way to change audible and visual modes while a state is active.

6.3 Design

The behavior module is designed as a single ROS node, interfaced through the actionlib protocol, all communication with other modules is through ROS protocols. The module connects to the controller module to get the emotional state of the Cyborg, and to provide emotional feedback. The

module also connects to the audio and visual modules in order to activate them. If the behavior incorporates navigation, the module activates a client for the navigation module.

The module has a database for behavioral presets where the parameters are loaded from, separate presets can be configured for each emotional state if one wishes to incorporate the emotional state of the Cyborg into the behavior. A behavioral preset defines commands for the audio, visual, navigation module, and emotional feedback. The preset also defines the completion trigger for the behavior, and if the behavior state is dynamic. The trigger can either be a duration, or completion of audio or navigation execution, callbacks are used in order to handle the executional feedback from these modules. When a behavior state is dynamic, it is possible to activate different behaviors in the same behavior state, this is done by publishing a command to the behavior module. The reasoning for including this option is that for states with relative long duration, for example a state where the Cyborg is following a long path, it can be restrictive to only be able to execute one type of visuals or audio. As most long lasting states are navigational states, the dynamic behavior commands are limited to only activating the audio or visual module. A class diagram for the modules action server Python class is presented in Figure 6.2.

6.4 Implementing the Behavior ROS Module

The behavior module is implemented as a single Python class *BehaviorServer*, instantiated by the ROS node `cyborg_behavior`. It provides behavior states through an action server, utilizing the *StateMachine.action* action file implemented in the controller module. Behavioral presets are saved in the modules launch file `behavior.launch`, which is accessed through the ROS parameter server. The form of a behavioral preset is as follows:

```
1 <rosparam param = "preset_name">
```

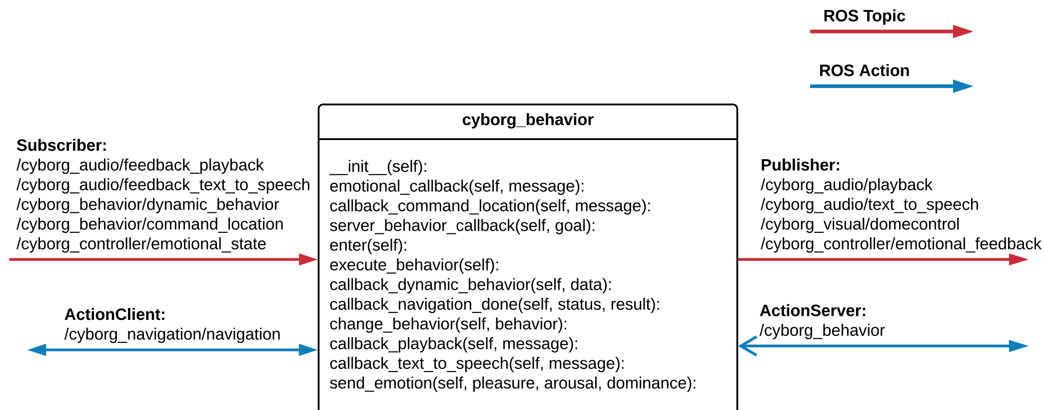


Figure 6.2: Class diagram for the behavior module *BehaviorServer* Python class.

```
2     visual_mode: "visual_command"
3
4     playback :  "playback_command"
5
6     utterance: "utterance"
7
8     navigation_order:  #Can be "navigation_go_to", "navigation_wander", or "navigation_dock".
9
10    location: "location"
11
12    dynamic : True   #Omit if False. True if state permits behavioral changes while active.
13
14    completion_trigger:  #Can be "navigation", "utterance", "playback", or "time xx", where xx is the duration in seconds.
15
16    timeout:  #Set False to disable behavior timeout, set duration in seconds to configure, omit if default value is used.
17
```

```
18     emotional_feedback: { p: value, a: value, d: value,
    continuous: boolean } #Continuous variable is set to True for
     continuous emotional feedback.
19 </rosparam>
```

Parameters not relevant for a preset can be omitted in the launch file when configuring new presets, default values are used by the module if it cant find certain parameters for the behavioral preset, as indicated in the example above. Comments and spaces have been added for readability, and are not part of an actual preset.

As seen in the class diagram presented in Figure 6.2, the behavior module uses ROS topics to connect to the other modules, callbacks are implements all subscribers. In addition, an action client is used to interface the navigation module, utilizing the *Navigation.action* action file implemented in the navigation module. For behaviors involving navigation to a specific location, there are multiple ways to supply the location. The location can be defined in the behavioral preset, it can be published on the `/cyborg_behavior/command_location` topic before activating the navigation state, or it can be sent with the action goal for the module, by supplying the location name in the "order" field of the `StateMachine.action` action message. The different options were included in order to facilitate less modifications to the old software structure. A simplified activity diagram for the behavior module is presented in Figure A.1.

A description of the functions in the module is given:

emotional_callback - Callback for the subscriber to topic `/cyborg_controller/emotional_state`. Updates the modules emotional state variable when a message is received.

callback_command_location - Callback for subscriber to topic `/cyborg_behavior/command_location`. Updates the modules *command_location* variable when a message is received.

server_behavior_callback - Callback for the `/cyborg_behavior` action server. When an action goal is received, the callback checks if the requested preset exists, first by appending the name of the behavioral state to the name, and then without if no preset for that particular emotional state is found. If no preset at all is found, the callback sets the server state as aborted and returns. If the name of the requested preset is found, the `enter` and `execute_behavior` functions are executed in sequence.

enter - Executed by `server_behavior_callback`. Retrieves parameters for the behavioral presets and updates the corresponding variables in the module. If the preset involves navigation, an action client for the navigation modules action server is instantiated.

execute_behavior - Executed by `server_behavior_callback`. Activates the different output modules, for the navigation module it sends navigation goal, while for the audio and visual module, commands are published on their respective topics. The function checks the `behavior_finished` variable to see if the behavior is finished, the state of the navigation server when it is used, and for preemption request from the module using the behavior action server. If navigation or the behavior server is preempted, a preemption command is sent to audio modules utilized in the current preset. When the behavior is finished, `send_emotion` is used to provide emotional feedback to the Cyborg controller.

callback_dynamic_behavior - Callback for the topic `/cyborg_behavior/dynamic_behavior`, only active when the parameter for dynamic behavior is set. Parses message and activates the corresponding output modules.

callback_navigation_done - Callback for the navigation module. Sets the behavior finished variable when called.

change_behavior - Called by `callback_dynamic_behavior` when a behavior command is intercepted. Retrieves behavioral parameters for the audio and visual module and executes.

callback_playback - Callback for topic `/cyborg_audio/feedback_playback`, handles feedback for audio playback and sets the *behavior_finished* variable accordingly.

callback_text_to_speech - Callback for topic `/cyborg_audio/feedback_text_to_speech`, handles feedback for text to speech and sets the *behavior_finished* variable accordingly.

send_emotion - Publishes emotional feedback when executed.

6.4.1 How to add new Behavioral Presets

New behavioral presets are configured by appending the desired preset to the `behavior.launch` file in the behavior module. Presets are added to the Cyborg state machine in the same way as other action server states, shown in the following code:

```
smach.StateMachine.add("<state name>",
    Module("<preset name>", "cyborg_behavior", transitions, resources),
    transitions,
    sm_remapping)
```

Where <state name> and <preset name> is replaced by the chosen state and preset name, which usually are identical.

6.5 Discussion

When I first designed the behavior module I was in doubt wether to incorporate emotional feedback or not. In the end I chose to incorporate it in order to

make the presets fully cover the role of a proper state, and also enabling us to remove more of the previously implemented states. In order to test the implementation, I defined a set of different presets in the corresponding launch file and tested the module by simulating the Cyborg. The module handles feedback from the audio module as it should, enabling us to use executional feedback from the audio module as trigger for state termination. This has proven valuable as it enables us to break down the size of states and implement simple state sequences based on something other than a set duration or navigation.

For future expansions I would recommend distributing the behavioral presets over several launch files, as the files quickly grows in length as new presets are appended. This can be done by including the files for behavioral presets in the main launch file for the module. I also thought about adding the option to configure a return event upon completion of a preset execution, but the idea was never fully evaluated. This might be valuable for more complex state machine configurations, but the downside can be added complexity.

6.6 Conclusion

A common interface for the most commonly used output modules has been designed and implemented. The module also provides a simple way to configure new behavioral presets that can be used as states for the Cyborg state machine, which in turn aids the overall configuration of the state machine by removing the need to implement new action servers for simple states. The simple states can be used to constuct more complex state sequences from states that are easy to understand, while enabling us to save time on implementation. The module has been tested by simulating the Cyborg and is ready for integration.

Chapter 7

Cyborg Audio Module

7.1 Introduction

The Cyborg audio module gathers functionality previously provided by modules `cyborg_music` and `cyborg_text_to_speech`, while decoupling output and Cyborg states. The module handles execution of playback and text to speech onto the Cyborg speakers, makes itself available for other modules through ROS topics, and provides executional feedback. A context diagram for the cyborg audio module is presented in Figure 7.1.

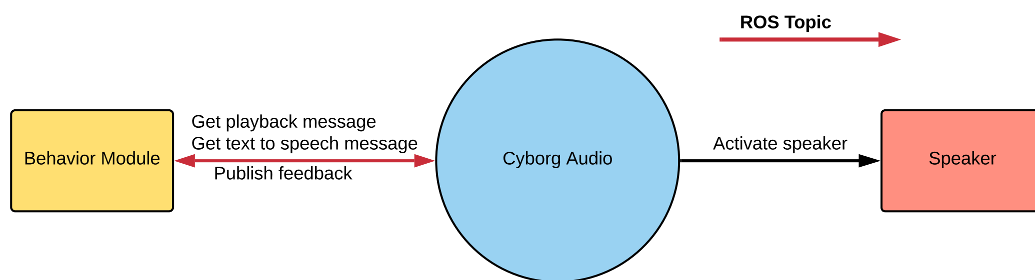


Figure 7.1: Context diagram for the cyborg audio module.

7.2 Requirements and Specifications

Based on the software structure of the Cyborg and the evaluation presented in Chapter 3, the following requirements and specifications have been stated for the audio module:

- The module must be implemented in ROS, as a ROS node.
- All communication with other ROS nodes must use ROS protocols.
- The module must be able to handle playback of audio files and text to speech.
- Implementation of playback and text to speech must be done as separate and independent instances.
- The module shall publish feedback when execution is finished or preempted.
- Optional: Execution should be preemptive.

7.3 Design

The audio node is designed as a single ROS node. As per the stated specifications, playback and text to speech are separate instances, each with their own channels for commands and feedback. A conceptual class diagram is shown in Figure 7.2. Both playback and text to speech handles preemption requests through messages, and both reply with a feedback message once execution is finished or preempted. A sequence diagram for the playback module is seen in Figure 7.3.

7.4 Implementing the Cyborg Audio ROS Node

The implementation is done in Python, as a single ROS node called `cyborg_audio`. Playback and text to speech are implemented as separate Python classes, instantiated in the `cyborg_audio` ROS node. Individual ROS topics

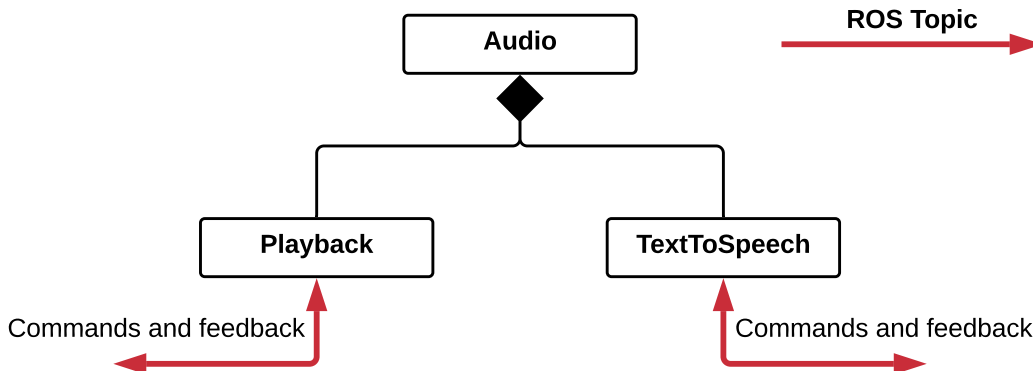


Figure 7.2: A conceptual class diagram for the Cyborg Audio ROS node.

with message type `std_msgs.msg.String` are used for commands and feedback. A class diagram for the audio module is presented in 7.4. ROS topics were chosen as they require less lines of code than actions.

7.4.1 Playback

The playback module is reimplemented using ROS topics instead of the actionlib protocol. The main body of the module is implemented as a threaded function using the `threading` library, python library `vlc` is used to play the audio files.

The module is made up of two functions:

playback - Main function of the module. Signaled by `callback_playback` when a message is received, loads the requested file into vlc and executes. Checks for preemption requests when active, and publishes a result message on the feedback topic upon completion or preemption.

callback_playback - Callback function for the ROS topic subscriber, stores the message content and signals `playback` through a shared variable.

The following commands are available:

- `"<filename>"` - Replace <filename> with your filename, the name is written without the filetype extension.

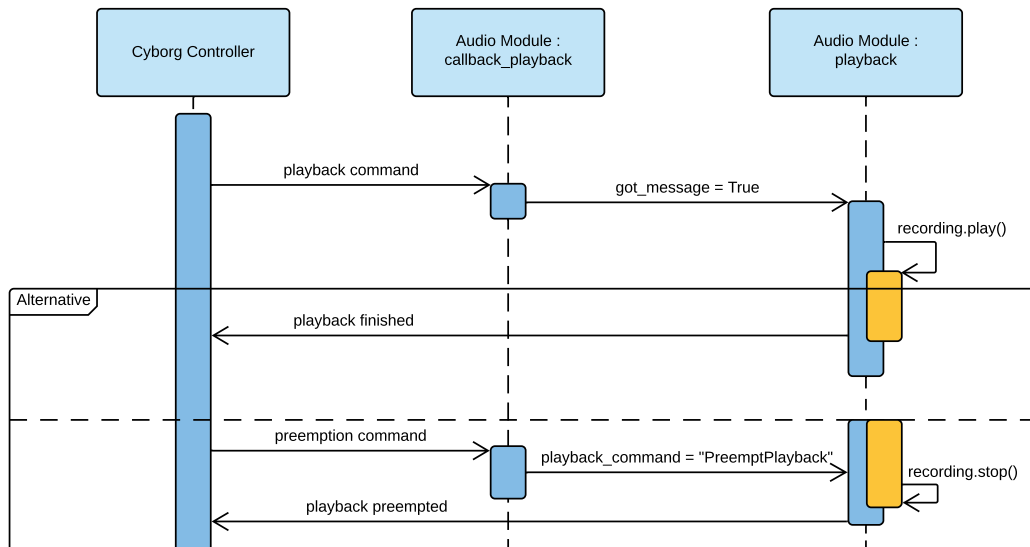


Figure 7.3: Sequence diagram showing the playback module being activated and preempted.

- `"PreemptPlayback"` - Preempts playback.

The following feedback is provided:

- `"playback finished"` - Published when playback is finished.
- `"playback preempted"` - Published when playback has been preempted.
- `"playback timeout"` - Published if playback has timed out.

The requested audio file must be located in the `homedir` folder, only mp3 files are supported.

7.4.2 Text To Speech

The text to speech module is reimplemented with a new text to speech engine `pyttsx3`, in addition preemption and feedback of execution has been added. The module is made up of three functions:

text_to_speech - This is the main function of the module, it starts the text to speech engine and connects to the `on_end_tts` function.

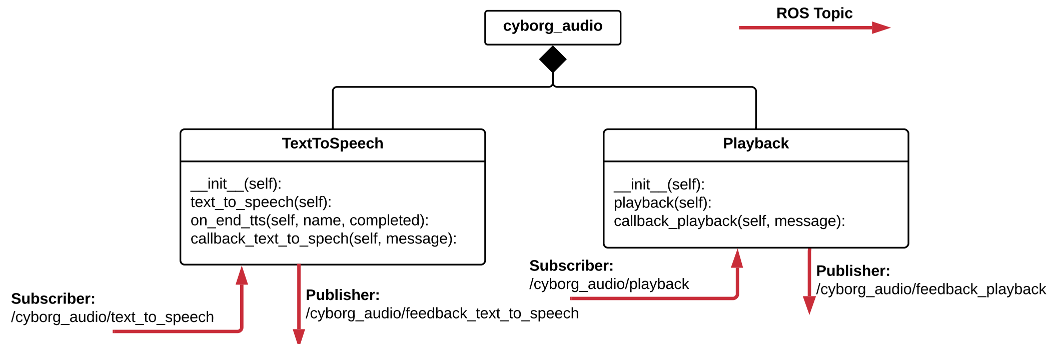


Figure 7.4: Class diagram for the Cyborg audio module.

on_end_tts - Executed upon completion, publishes feedback on the feedback ROS topic.

callbcak_text_to_speech - Callback function for the ROS topic subscriber, stores the message content and starts or preempts execution. Publishes feedback on the feedback ROS topic upon preemption.

The following commands are available:

- `"<utterance>"` - Replace <utterance> with the wanted utterance to execute text to speech.
- `"PreemptUtterance"` - Preempts utterance.

The following feedback is provided:

- `"utterance finished"` - Published when utterance is finished.
- `"utterance preempted"` - Published when utterance has been preempted.

7.5 Discussion

While testing the previously implemented text to speech module, I was disappointed by the quality of the speech synthesizer. The Python library that is used supports different engines for the speech synthesizer, but only

one of them is actually available on a Ubuntu system, namely *espeak*. I did not prioritize finding an alternative, but although the speech synthesizer does its job, it could definitely be better. For future upgrades, I would recommend changing to a online subscription based text to speech service like Watson Text to Speech by IBM or Google Cloud Text-to-Speech. The online versions are far superior in quality and require less processing power of the Cyborg. This solution of course would require a stable internet connection on the Cyborg, which one could argue should be provided any way. It is also possible to download commonly used phrases and use the playback module instead, or use the online service only the first time a phrase is used, and then save the file for future use.

The implemented module now gathers similar output functionality, decouples playback from Cyborg states, and provides them through a common interface, making the software structure of the Cyborg less complex and the module easier to interface. The feedback from the module is exploited by the behavior module, and can be used as a trigger for ending states.

7.6 Conclusion

An audio module for the Cyborg has been designed and implemented, gathering functions previously provided by two separate modules with different interfaces. The module supports text to speech and playback of audio files, execution can be preempted for both, and playback is no longer directly coupled with a state, enhancing modularity for the Cyborg software and states, while making it less complex to work with. Executional feedback is provided by the module, which in turn makes it possible to base a states duration on executions in this module. The module has been tested and is ready for integration into the Cyborg.

Chapter 8

The Event Scheduler Module

8.1 Introduction

The event scheduler module provides functionality pertaining to publishing scheduled events for the Cyborg. The module is also responsible for detecting and publishing other system events like when the Cyborg battery is starting to run low. The `scheduler` function previously found in module `cyborg_navigation` is isolated and moved to this module. A context diagram for the module is seen in Figure 8.1.

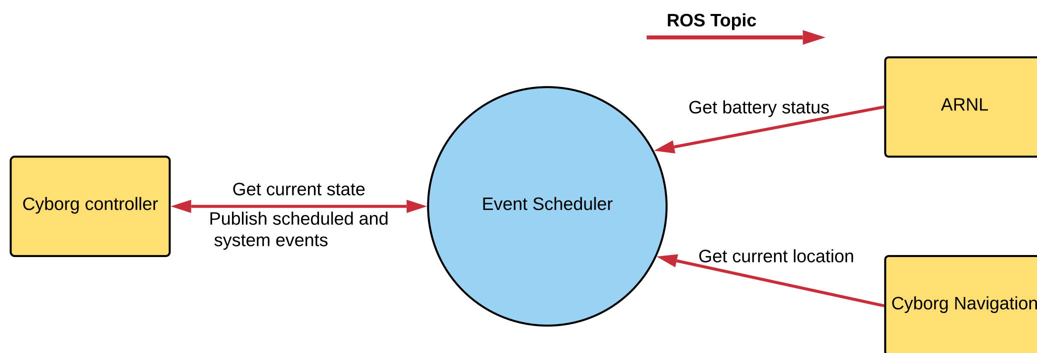


Figure 8.1: Context diagram for the event scheduler module.

8.2 Requirements and Specifications

The following requirements and specifications have been defined for the event scheduler module:

- The module must be implemented in ROS, as a single ROS node.
- All communication with other ROS nodes must use ROS protocols.
- The module must interface the navigation module for location information.
- The module must notify the Cyborg state machine about scheduled events.
- The module must notify the Cyborg when the battery is below a set limit.

8.3 Design

The event scheduler is designed as a single ROS node, all communication with other modules is through ROS protocols. The node subscribes to state data published by the state machine, location data published by the navigation module, and battery status published by the `ros_arnl` node. If a scheduled event at another location than the current one is found, the rest of the Cyborg state machine is notified, if the battery gets below a set limit while the Cyborg is not already charging or on its way to the charger, a corresponding event is published. The scheduler itself is almost identical to the function previously found in the navigation module, with some slight modifications done in order to make it fit the overall system structure. A class diagram, with corresponding ROS protocols is presented in Figure 8.2.

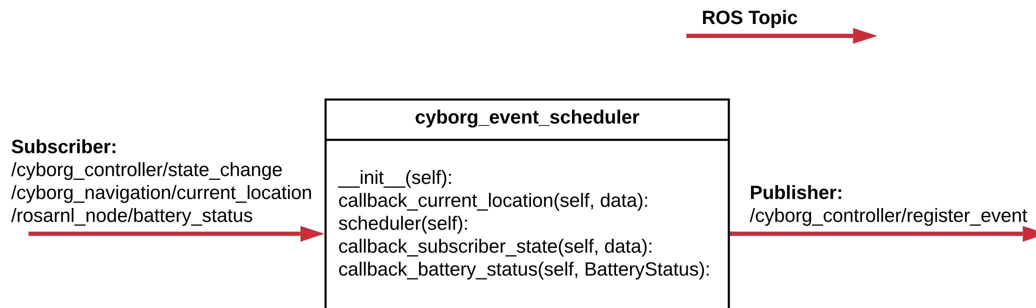


Figure 8.2: Class diagram for the event scheduler ROS node, with corresponding ROS protocols.

8.4 Implementing the Event Scheduler ROS Node

The event scheduler is implemented as a single ROS node, containing the scheduler function previously mentioned, and in addition callbacks for the topics it subscribes to:

scheduler - Checks for ongoing events, by searching through the database of events provided in the navigation module. If an event at another location than the current one is found, a *navigation_schedular* event is published on the event topic of the controller module.

callback_current_location - Callback for the location subscriber, updates the current location of the module.

callback_subscriber_state - Callback for the state subscriber, updates the module with the current state.

callback_battery_status - Callback for the battery status subscriber. If the battery percentage is below a set value while the Cyborg is not already charging og moving to the dock, a *power_low* event is published on the event topic of the controller module.

8.5 Discussion

Even though the scheduled events are only navigational event at this moment, this will possibly change. Part of the reasoning behind a separate event scheduler module is to keep the door open for a more advanced event scheduler in the future, aiding the goal of a software structure where similar functionality is gathered in separate modules. Removing the scheduler function from the navigation module has made the navigation module less complex, and easier to comprehend. The module has been tested both on its own and together with the rest of the Cyborg modules, and it works as expected.

8.6 Conclusion

Based on the evaluation done in Chapter 3, the scheduler from the navigation module has been isolated and implemented as a separate module. A function that detects and publishes an event when the battery is below a set limit is also implemented in the same module. The event scheduler module has been tested and is ready for integration into the Cyborg.

Chapter 9

Primary States Module

9.1 Introduction

The primary states module gathers action server states with more complex behavior than provided by the `Behavior` module, and states that do not use output modules, the old `wandering`, `idle`, and `navigation_planning` states are reimplemented here. The module provides emotional feedback to the controller while active, and can execute state changes in the Cyborg by publishing events. If behavioral outputs are needed, the *Behavior* module is also interfaced. A context diagram for the module is presented in Figure 9.1.

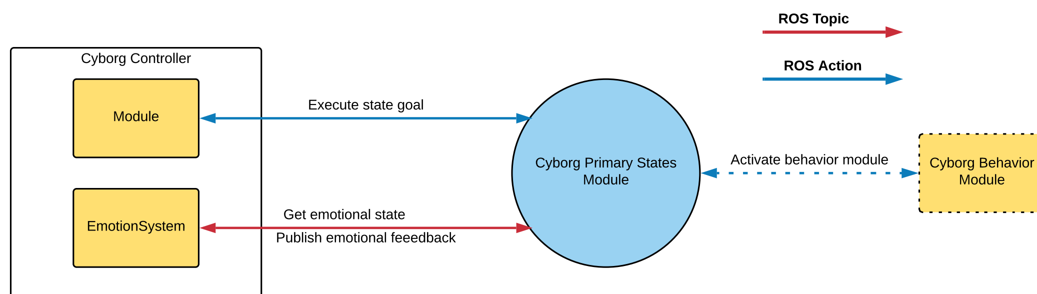


Figure 9.1: Context diagram for the primary states module.

9.2 Requirements and Specifications

Based on the evaluation presented in Chapter 3 and the software structure of the Cyborg, the following requirements and specifications are stated for the module:

- The module must be implemented in ROS, as a single ROS node.
- ROS protocols must be used for communication with all other ROS nodes.
- The module must interface the controller module in order to exploit the emotional state of the Cyborg.
- In order to search for navigation locations, the module must be able to use the databasehandler provided by the navigation module.
- The module must be able to interface the behavior module.
- The module shall provide states for the Cyborg state machine through a single ROS action server.
- The module shall provide emotional feedback in relevant states.

9.3 Design

The module is designed as a single ROS node, providing states through a ROS action server. In order to save resources, all states are provided through the same action server, using the callback for the action server to execute the wanted function based on the action goal. The module connects to the controller module to get the emotional state of the Cyborg, provide emotional feedback, and register events leading to state changes. For states using output modules, `Behavior` is also interfaced. A class diagram for the module is seen in Figure 9.2

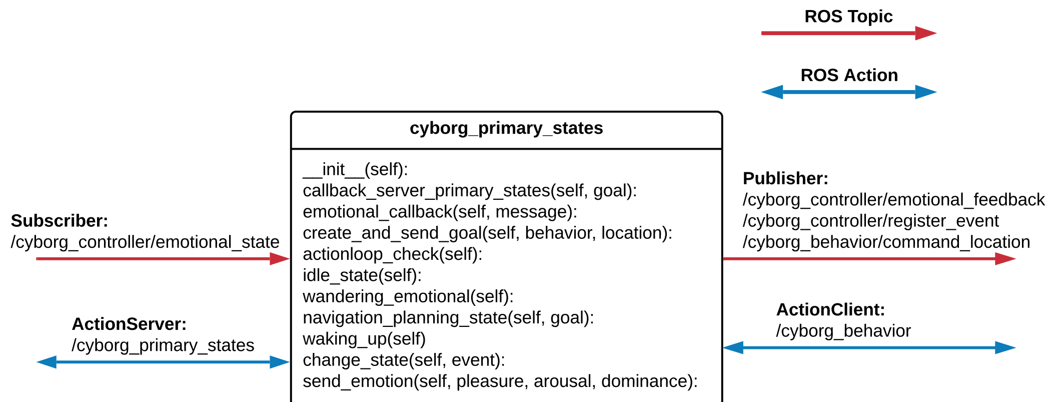


Figure 9.2: Class diagram for the primary states module.

9.3.1 Wandering Emotional State

Based on the *wandering* action server state previously implemented in the navigation module. The state activates a wandering behavior in the behavior module, and preempts itself when the emotional state of the Cyborg changes to something other than "bored", "curious", or "unconcerned".

9.3.2 Navigation Planning state

Based on the *navigation_planning* action server state previously implemented in the navigation module. The state executes when the Cyborg state machine receives a "navigation_schedular" or "navigation_emotional" event. It checks the current emotional state of the Cyborg and selects how and where the Cyborg should move, before providing emotional feedback and initiating a state change.

9.4 Implementing the Primary States Action Server Node

The module is implemented according to the class diagram presented in the previous section, ROS topics are used for communication with the controller and behavior module. The module provides its states as a single ROS action server that utilises `StateMachine.action` actions, and interfaces *Behavior* through an action client when needed:

callback_server_primary_states - Callback for the action server, checks the action goal and executes the corresponding function.

emotional_callback - Callback for the emotional state topic.

create_and_send_behavior_goal - Used to create and send action goals to the *Behavior* module.

actionloop_check - When executed, checks for preemption requests for the active state, and the executional status of the *Behavior* action server. Sets the terminal state of the action server according to the terminal state of the behavior server.

wandering_emotional - Activates wandering behavior in *Behavior*, preempts the state when the Cyborg is in the wrong mood.

navigation_planning_state - Based on the old `navigation_planning` state. Activated by either a *navigation_emotional* or *navigation_schedular* event. Checks the emotional state of the Cyborg, and decides and activates the next state. If the next state is `navigation_go_to`, the location is published on the `/cyborg_behavior/command_location` topic for the *Behavior* module, before the state change is executed.

change_state - Executes a state change by publishing events on the corresponding topic.

send_emotion - Used for publishing emotional feedback to the controller, by publishing a *EmotionalFeedback* message on the corresponding topic.

9.5 How to Implement new States

New states are implemented as standard Python functions and then added to the callback function `callback_server_primary_states` in order to make them available. An example is given:

```
def example_state(self):

    ### Implement state behavior here ###

    self.create_and_send_behavior_goal(behavior = "example_behavior")
    while not rospy.is_shutdown():

        ### Implement state executional checks here ###

        if self.actionloop_check() == True:
            return

        self.RATE_ACTIONLOOP.sleep()
    # set terminal goal status in case of shutdown
    self.server_primary_states.set_aborted()
```

- A state with name "example_state" is defined on line 1.
- State behavior is implemented before the while loop.
- On line 5 function `create_and_send_behavior_goal` is used to send action goal "example_behavior" to the behavior module.
- Variable and state execution checks are added inside the action loop.
- `action_loop` function is used to check the action goal state of the behavior module on line 10, terminating the loop on line 11 and ending

the state if the behavior module is finished or has stopped for some reason.
- The loop is suspended on line 13.
- And the terminal goal status for the action server is set on line 15 in case the while-loop exits for some unexpected reason.

9.6 Discussion

The main goal of this module was to gather action server state functionality that could be removed from other modules, while providing a common place for some of the most used states as well as commonly used future action server states. An effort has also been put into facilitating easy implementation of new states, by providing various functions that can eliminate some of the most common portions of code used in action server states, resulting in less time spent when implementing new states for the Cyborg.

9.7 Conclusion

In accordance with the evaluation in Chapter 3, a module for the most commonly used action server states has been designed an implemented. The module offers a common place for frequently used action server states, and states implemented in the future. The `navigation_planning` and `wandering` action server state functionality previously found in the `cyborg_navigation` module has been gathered and reimplemented in this module. In addition, functions aiding implementation of new states have been implemented. The module has been tested and is ready for integration into the Cyborg.

Chapter 10

Navigation Module

10.1 Introduction

The navigation module provides navigational behavior for the Cyborg through an action server. The new design keeps the databasehandler as is, while the navigationserver is stripped down, removing all functionality pertaining to talking behavior and the scheduler, only keeping what is needed in order to properly interface with the `ros_arnl` module. In addition, the navigation module should preferably provide docking for the Cyborg, which has to be implemented. Since the `scheduler` that is removed relies on location data, this has to be provided by the navigation module. A context diagram for the navigation module (omitting parts regarding the databasehandler) is presented in Figure 10.1.

10.1.1 Design

The overall structure of the navigation module is changed slightly, going from one action server per state, to one common action server for the whole module. When a module connects to the action server provided in the navigation module, the action goal is parsed by the action server callback, and

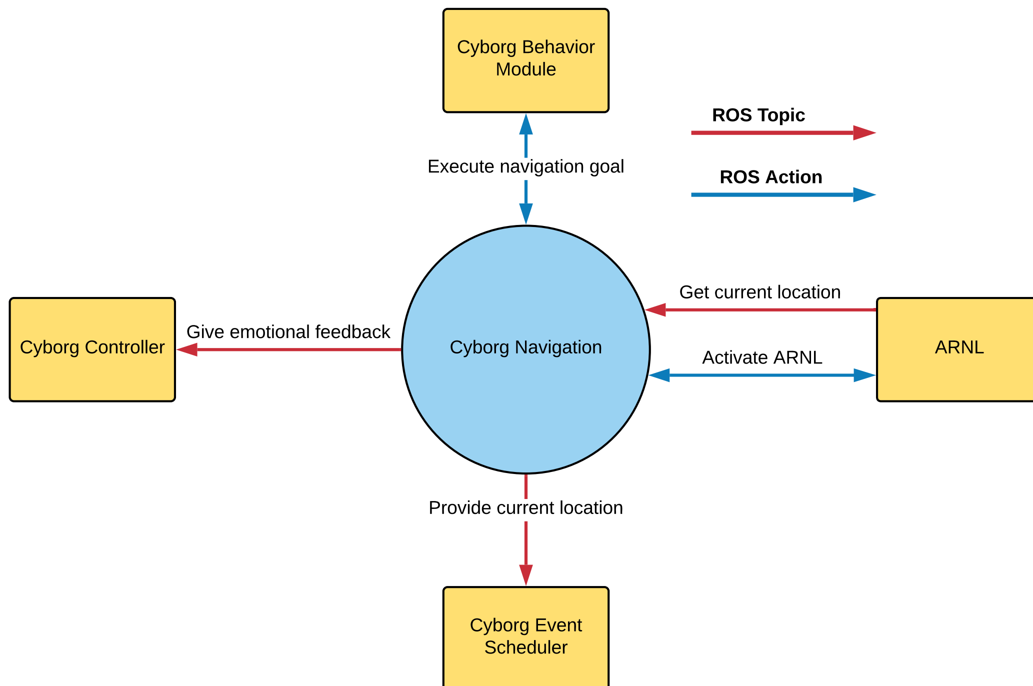


Figure 10.1: Context diagram for the navigation module, parts regarding the databasehandler are omitted.

the appropriate function executed. While one can argue that a single action server goes at the expense of modularity, a common action server is chosen in order to save some processing power, not having to offer multiple servers. And as the overall system structure of the Cyborg is changed, utilizing a common interface for the output modules, there is no reason to keep the few action servers still remaining as separate entities when they are all directly related to navigation output, other than modularity inside this particular module.

In order to satisfy the need for location data in other modules, the navigation module publishes the current location name on a ROS topic. Also different from before, before sending a location goal to the `ros_arnl` node, the navigation module first checks if `ros_arnl` is able to calculate a path to the wanted location. The previously implemented wandering function is also

stripped down somewhat, removing the part where the wandering state makes the choice about how long it should stay in this state based on the emotional state of the Cyborg. A class diagram for the new navigation module is seen in Figure 10.2.

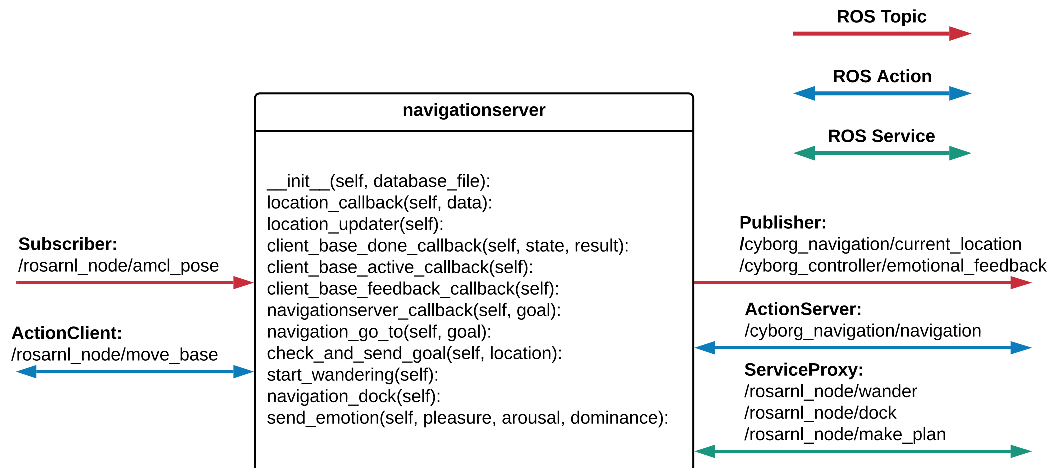


Figure 10.2: Class diagram for the navigation module, the databasehandler is omitted.

10.1.2 Reimplementing the Navigation ROS Node

The module is implemented as a single ROS node as before, ROS protocols are used for all communication with other nodes. The module implements the action server `cyborg_navigation/navigation`, actions are provided in the action file *Navigation*. The action file is based on the previously used *NavigationGoTo* action, where the variable "order" has been appended to the goal definition, and the action has been renamed since it is now used for all navigation actions. Only the new or changed functions are listed below:

location_updater - Checks if the current location is registered as a location in the databasehandler, and publishes the location name on a ros topic.

client_base_done_callback - The original function is modified in order to handle the aborted action server state.

navigationserver_callback - Callback for the navigationserver, parses the action goal and executes `navigation_go_to`, `navigation_wander`, or `navigation_dock` accordingly.

navigation_go_to - Based on `server_go_to_callback`. The action server callback is reimplemented as a standard function, and modified in order to provide emotional feedback and handle action server state "aborted" from `ros_arnl`. The function is executed by `navigationserver_callback` when action goal "navigation_go_to" is received.

check_and_send_goal - The original function is modified in order to check if the `ros_arnl` is able to calculate a path to the location, before the goal is sent. This is done by using the ROS service *MakePlan* provided by the `ros_arnl` node on path `/ros_arnl/make_plan`. If the service provides a negative response, the action server is signaled by setting a common variable indicating that the base goal is canceled.

start_wandering - The original function is modified, removing the parts of code responsible for preempting the state when the Cyborg is in the wrong emotional state.

navigation_dock - Implements docking behavior for the navigation module. The function activates the dock service provided by the `ros_arnl` node on path `/ros_arnl/dock`. The service is stopped when the action server gets a preemption request.

10.2 Discussion

The stripped down and reimplemented module has been tested with the rest of the Cyborg modules by simulating the system. As the social part of the

Cyborg is an important aspect, I would propose to reimplement the states that were removed from the navigation module somewhere else.

10.3 Conclusion

In accordance with the evaluation presented in Section 3.5, the navigation module has been partly reimplemented. Cyborg state and scheduler functionality has been stripped away, docking functionality has been implemented, and all navigation actions have been gathered in one action server. The reimplemented navigation module has a simpler structure with a more condensed task responsibility. The module has been tested by simulating the Cyborg in *MobileSim* and is ready for integration into the Cyborg.

Chapter 11

LED Dome

11.1 Introduction

As stated in Section 3.8, the LED dome ROS module and LED controller needs to be made ready for integration into the Cyborg. The interface for the LED controller needs to be redesigned and implemented, and the ROS Module must be modified in order to facilitating the changes. The overall tasks presented in Section 3.8 will serve as a basis for the work presented in this chapter.

11.2 Implementing the LED controller Circuit

The proposed implementation of the LED controller circuit that was presented in the authors specialization project [1] is modified slightly, making it simpler by removing a redundant wire. A sketch of the modified proposition is seen in Figure 11.1. The proposed circuit is realized on a stripboard, seen in figure 11.2. Sockets have been used for both chips, facilitating easy removal of the components.

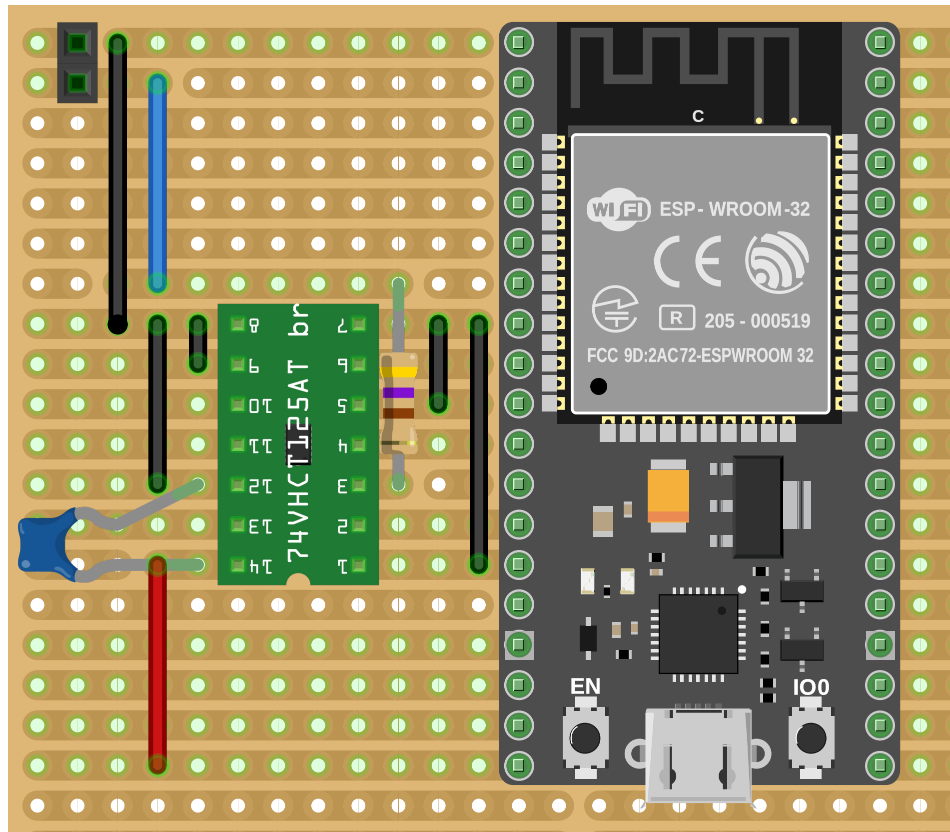
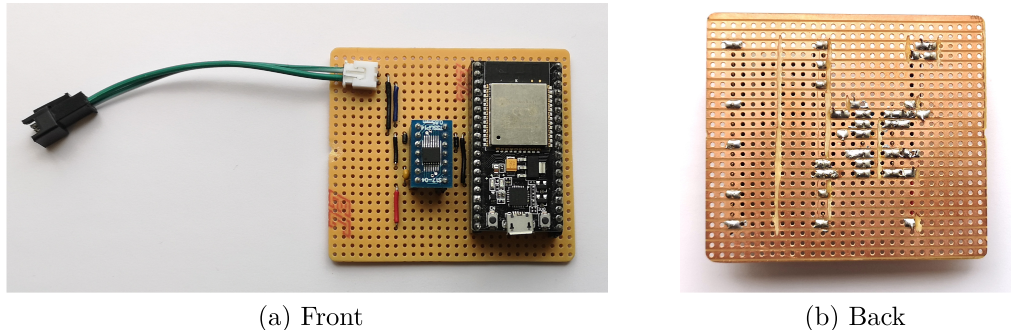


Figure 11.1: A proposed implementation of the LED-controller on a stripboard. Black wires indicate ground connections, red power, and blue data. Notice where the copper tracks are cut, in addition all tracks under the ESP32 and buffer are cut, except for the track between pin 7-8 on the buffer.

11.3 Redesigning and Implementing the LED Dome Software

11.3.1 Technical Considerations and Requirements for the LED Controller Interface

In accordance with the changes to the LED controller interface, the following considerations and requirements are stated:



(a) Front (b) Back

Figure 11.2: The LED-controller, realized on a stripboard. Notice which copper tracks are cut.

- The interface must be implemented using FreeRTOS tasks for the main functions.
- FreeRTOS queues must be used when passing visualization data between tasks.
- The interface must be able to detect the start and end of a serial message, and in addition correctly identify the message content.
- The interface shall support text visualization on the whole LED dome.
- The visualizations must be executed correctly and fast enough to look "aesthetically pleasing".

11.3.2 Interface for the LED Controller

In order to initialize the library for text animations, a defined rectangular area of pixels is needed. The interface that was presented in the authors specialization project [1] was only using seven of widest pieces of led strip on the LED-dome, where the first three strips only are one pixel wider than the next four, which in turn makes defining a rectangular area trivial. In reality, when laid out, the led strips take on the form of an egg, defining rectangles inside an egg shape means loosing big portions of the available pixels, which obviously is not a good solution.

Another challenge that needs to be resolved in order to have proper text animations, is the fact that there is no system in how the individual portions of led strips are aligned on the LED-dome, meaning that when text animations are displayed, there is nothing that guarantees that the animations will be aligned throughout the strips, potentially rendering the animations so skewed that they are useless. To tackle this problem, a new remapping function, remapping the whole LED-dome is designed and implemented. The function enables us to emulate a rectangle enclosing the physical egg-shape of the led strips, and aligns the output on the led strips. Pixels that fall outside of the physical egg shape the led strips take on, are discarded.

Arduino libraries *FastLED*, *LEDMatrix*, *LEDText*, and *FontMatrise* are used for controlling the leds and creating text animations [36][37]. The new interface implements two FreeRTROS tasks, `ReceiveSerialDataTask` and `VisualizationTask`, one for serial communication and one for visualizations. The serial messages that are sent from the LED dome ROS module are on a special format where values 252-255 are reserved, their definition is given in table 11.1.

VALUE	**FUNCTION**
252	Indicates if text should be vertical
253	Indicates if message is text
254	Start of message
255	End of message

Table 11.1: Definitions for reserved values over serial.

When a message is received by the LED controller, the data is parsed by the serial task, before it is put into a buffer and sent to the visualization task. The buffer is a data structure on the following form:

```
typedef struct{
    bool textmode;
```

```
    bool verticalMode;
    uint8_t length;
    uint8_t data[NUM_LEDS*3];
} MessageStruct;
```

The two boolean variables indicate if the data is text and if the text should be displayed vertical or horizontal. The data is stored in the *data* array, and the length of the array in *length*. FreeRTOS queues are used for passing pointers to the structures containing visualization data between the two tasks. When an item on the queue is consumed by the visualization task, the visualization task executes accordingly, before the address of the buffer is returned to the serial task on a separate queue. The following functions are implemented in the interface for the LED controller:

setup - The first function to execute, initializes visualization and FreeRTOS queues, and starts the two tasks.

ReceiveSerialDataTask - Checks for available buffers on the appropriate queue, and intercepts serial data if one is found.

receiveMessageInfo - Executed by the serial task when a serial message arrives. Takes in a pointer to a buffer, parses the beginning of a serial message (before the actual visualization data) and sets the first two variables of the buffer.

VisualizationTask - Receives visualization data on a queue from the serial task and executes accordingly, before returning the consumed buffer back to the serial task.

remapLeds - Remaps the visualization data for text animations, called by the visualization task.

The new interface for the LED-controller supports visualizations of text and other animations, both in the same version.

11.3.3 The LED Dome ROS Module

The context diagram for the LED dome ROS module is presented in Figure 11.3. In order to facilitate the changes done in the interface for the LED controller, the text state in the LED dome ROS module is modified slightly. When text commands are sent to the module, the text state now handles commands for both vertical and horizontal text. This is done by checking if the first word after the text command is "vertical", indicating vertical text. Because of the reserved bit for the new command, the `loop` function is modified to round down all values above 251 to 251, before messages are sent to the LED controller over serial.

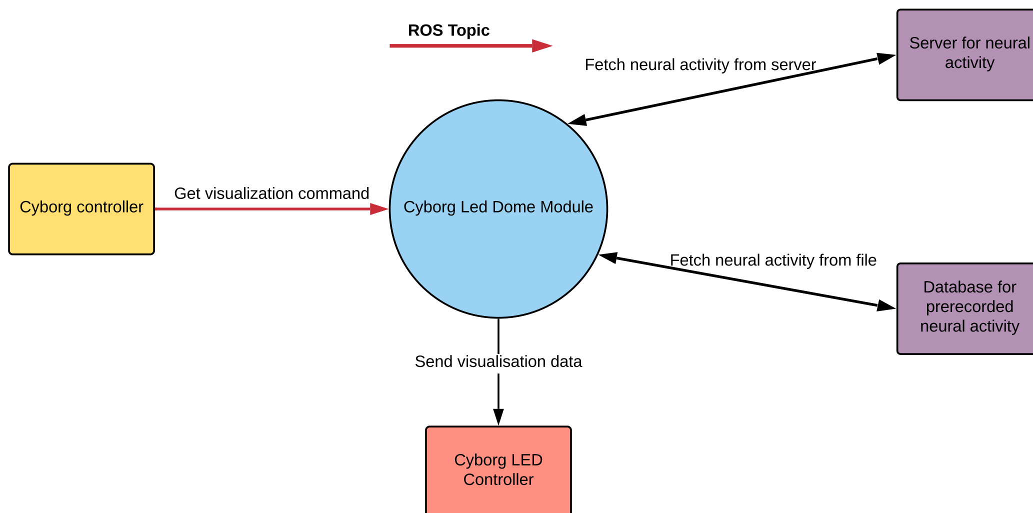


Figure 11.3: Context diagram for the Cyborg LED dome ROS module.

How to add new Animations

- Add the new interpreter and the name of the interpreter to function `return_interpreter`.
- Add the name of the new interpreter to function `updatevisualization_mode`.
- Add the name of the new interpreter to function `set_visualization_`

```
mode_callback
```

11.4 Preparing the LED Dome for Integration into the Cyborg

11.4.1 Casing

In order to facilitate the goal of proper integration of components, an enclosure for the LED-controller is designed and 3D printed. The casing is designed using Rhino 3D, and printed on a private 3D printer. It features a snap-fit cover with venting holes, and snap fits for the circuit board. Figure 11.4 show the drawing of the case. A photo of the finished LED-controller hardware, complete with casing, is shown in Figure 11.5.

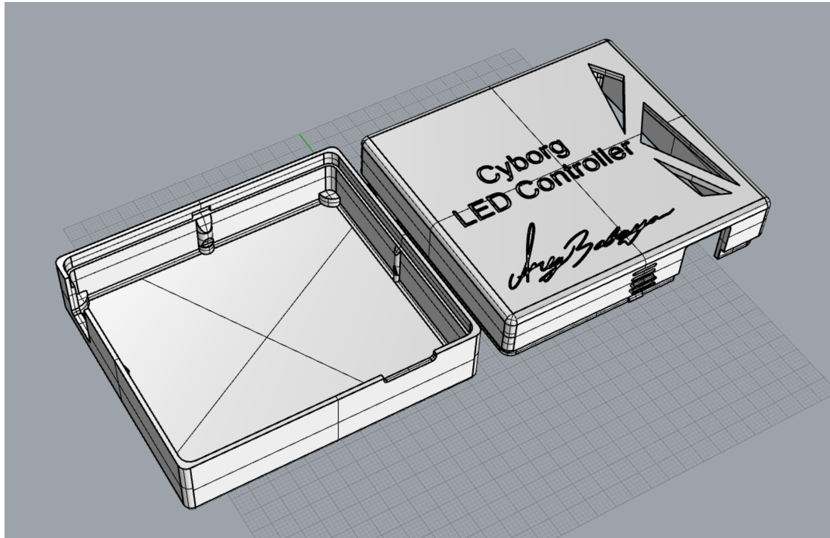


Figure 11.4: The casing for the LED controller, drawn in Rhino 3D.

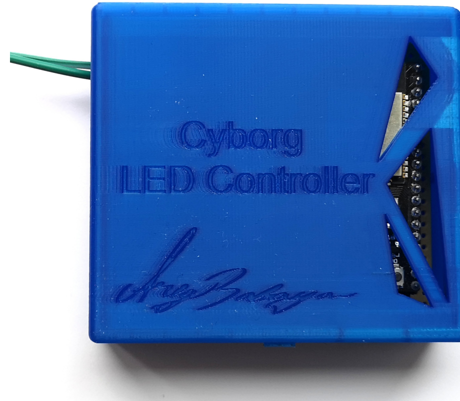


Figure 11.5: The LED-controller, mounted inside its casing.

11.4.2 LED dome ROS module

The LED dome ROS module has been tested and is ready for integration, except for installing the Module on the Cyborg, no further work is needed.

11.4.3 LED-dome

The author assisted group 3 from Experts in Team with their work on the LED dome, this work is presented in Chapter 4.

11.5 Discussion

The LED controller has been tested for extended periods of time on the authors computer, both hardware and software works as intended. No increase in framerate was achieved with the new interface, after some investigation the culprit is found to be the blocking time of binary semaphores, which are used in the implementation of queues. By changing to direct notification, one can expect a decrease in blocking time of up to 45% [38]. The 3D printed case looks good, and makes for a complete solution. Implementing new animations

was not prioritized, but would make a great addition to the LED dome.

11.6 Conclusion

In accordance with the evaluation presented in Chapter 3, the interface for the LED controller has been redesigned and implemented, and the LED Dome ROS node has been modified to accommodate for more advanced text commands. The new dual-core interface for the LED controller supports text commands in addition to visualization data sent from the LED Dome ROS node, utilizing the whole LED Dome for both.

The LED Controller circuit proposed in the authors specialization project[1] has been modified and realized, and an enclosure has been designed and 3D printed, making the LED controller ready for integration into the Cyborg. The framerate of the new LED controller interface is 15 Hz, 5 slower than the 20 that was achieved before. The lower performance than was achieved with the old dual-core interface is due to FreeRTOS semaphores being less effective than direct notifications.

Chapter 12

The Finishing Touches

12.1 Introduction

This chapter presents the last steps in making the Cyborg ready for demonstration. The body needs to be finished and the LED Dome properly integrated. The Cyborg also needs a working state machine, and all implementation must be properly tested on the Cyborg.

12.2 Cyborg Body and LED Dome

12.2.1 Protective Fan Cover

The cooling fan mounted by EiT group 3 is exposed on both sides. In order to prevent damage caused by the fan, protective covers are installed on both sides. Small rubber bits are added between the fan and the fan guards, the rubber bits dampen fan vibrations, and provide some distance between the fan and the fan guards in order to avoid scratching.

Figure 12.1: Protective fan covers installed on both sides of the exhaust fan.

The reassembled fan with added fan guards and rubber bits is seen in Figure 12.1.

12.2.2 Covering the Gap Between the LED Dome and Cyborg Body

A rubber sealing strip is installed on the LED dome, covering the gap between the LED dome and the body. The sealing strip features tape on the backside, making it easy to glue the strip onto the LED dome shell. The edge of the dome shell is uneven and should not be used as a reference when mounting the sealing strip. In order to make the sealing strip line up perfectly with the body edge, the LED dome is installed and the sealing strip pushed into the gap. When pleased with the orientation of the sealing strip, masking tape is applied to mark where the sealing strip should be glued on, seen in Figure 12.2 The LED dome is taken out of the Cyborg body, and the sealing strip glued on. The end result is seen in Figure 12.3.

Figure 12.2: Masking tape is used to line up the sealing strip on the LED dome.

12.2.3 Aligning and Fastening the Body Panels

Special parts have been designed and 3D printed in order to properly align the 3D printed body panels where they meet. The parts are mounted with strong double sided adhesive, seen in Figure 12.4.

(a) Before (b) After

Figure 12.3: Sealing strip mounted on the LED dome.

Care must also be taken in order to stop the backside body panel to slide off while operating the Cyborg. This challenge has been resolved by installing pieces of hook velcro tape on the body panel and Cyborg base right under the exhaust fan. A consecutive piece of loop velcro is placed over the hook pieces when the backside needs to be fastened.

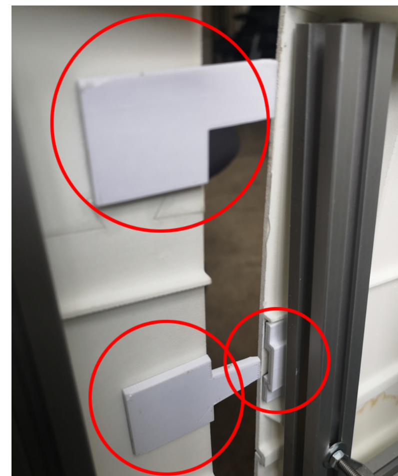


Figure 12.4: The body aligners installed on the 3D printed body.

12.2.4 Mounting the LED Controller

Velcro tape is used to mount the LED controller inside the LED dome. The velcro makes it easy to take out the controller if needed, and requires no modifications or permanent changes to the LED dome. Loose wires inside the LED dome are fastened with a strong type of duct tape. A picture of the mounted LED controller is seen in Figure 12.5.

12.3 Configuring the State Machine

A new SMACH state machine is implemented in the controller module in order to test all implementation and make the Cyborg ready for demonstration. A sequential state is implemented in `statemachines.py`, providing the main state machine with the navigation_go_to state for *navigation_emotional* events. State diagram for the Cyborg is presented in Figure 12.6. With the exception of states `navigation_planning` and `wandering_emotional`, all states are provided through behavioral presets in the behavior module. Beware that the state machine is configured with implemented docking behavior in mind, even though docking this is not fully implemented in the navigation module yet. States `exhausted` or `sleepy` lead to the docking state `sleeping`. The exhausted state is triggered by a "power_low" event published by the event scheduler module, while the sleepy state is triggered by a scheduled event published by the same module.

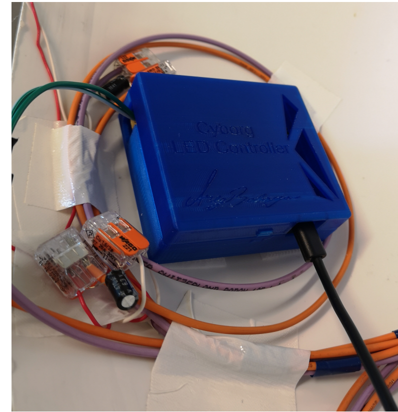


Figure 12.5: The LED controller mounted inside the LED dome with velcro tape.

12.4 Preparing the Cyborg for Testing

The setup script is updated to accommodate for the libraries used by the LED Dome module and text to speech, the following lines have been appended:

```
#Speech Output Module requirements:
pip2 install PyTTSX3

# LED Dome Module requirements:
pip2 install pyopengl
```

```
pip2 install pyopengl-accelerate
pip2 install numpy
pip2 install pandas
pip2 install colour
pip2 install pyserial
```

All new and changed modules are downloaded onto the Cyborg, and the system is built with the `catkin_make` command.

12.5 Testing the Cyborg v3.0

The Cyborg v3.0 is tested by activating the implemented state machine and letting it roam autonomously in Glassgården. The following acceptance criteria have been selected in order to satisfy the specifications for the Cyborg v3.0:

- All integrated modules must work together in a consistent manner. Canceled actions shall not lead to undetermined behavior.
- Visualizations and sounds must be activated correctly.
- The Cyborg must navigate properly.
- The Cyborg must avoid driving into people.
- Body and mounted hardware shall not be moving while traveling.

The cyborg state machine is activated either by using the Mode Selector or manually through a ssh connection. Two videos of the testing is included, a description is given below:

Video 336:

- The Cyborg is activated through a remote ssh connection. State `music_horror` is activated after the initial `idle` state and the Cyborg tries to move at 0:35, but the state fails and the Cyborg changes to the idle state, while music is continuing to play.
- At 0:51 the `wandering_emotional` state is activated, playback is still active.

- At 1:52 the `idle` state is activated, before `navigation` is activated at 02:10.
- At 3:00 and 3:23 the Cyborg reacts to the tile floor and stops shortly.
- The Cyborg reaches the intended location at 04:09, and the rest of the sequential states in the `navigation_go_to_emotional` are activated.
- At 04:49 the idle state is activated once again, `show_off_mea` is activated shortly after, lasting for exactly sixty seconds as configured in the behavioral preset.
- Idle state is activated at 06:03, before `astrolanguage` is activated at 06:16.
- At 06:35, the Cyborg is seen dodging a person.

Video 339:

- The Cyborg is activated with the Mode Selector box. `Music_horror` is activated and the Cyborg tries to move at 00:35, but navigation is aborted.
- At 0.50 `wandering_emotional` is activated, playback from `astrolanguage` is still active.
- The state finishes and `idle` is active at 01:51, before `navigation` goes active shortly thereafter. The Cyborg navigates to the commanded location and executes the proper arrival sequence.
- Idle state is active at 03:29, and `show_off_mea` at 02:42.
- At 04:20 the Cyborg dodges a gap in the floor.
- Idle state is active at 04:44 and `astrolanguage` shortly thereafter.
- At 06:30, the Cyborg drives into a gap in the floor, but manages to get out on its own.
- Idle state is activated at 06:50, before `show_off_mea` is activated at 07:01.

The videos show us a couple of important points regarding the behavior of the Cyborg, and in addition highlights some issues:

- The state machine works, the Cyborg operates autonomously.

- The Cyborg navigate successfully, humans and foreign objects are avoided.
- Gaps in the floor are sometimes avoided, but not always.
- Body panels and hardware does not move while the Cyborg is traveling.
- When a state is aborted, playback continues into the next state.
- The first event in the second video is a result of the state machine trying to activate navigation before all modules are properly inititated.
- All modules work as intended together, except for the issues stated in the point above.

In order to prevent the Cyborg from getting stuck in gaps in the floor, forbidden areas should be added where these gaps are on the map. The issue regarding playback continuing into the next state when the current state is aborted has since been fixed, by adding a preemption of playback when the behavior module aborts. The issue with navigation when the Cyborg is just initiated has also been addressed. A delay is added after *ARNL* is started in the startup sequences, giving navigation some time to properly initiate. In addition, the launch file in the controller module is updated, the controller module is now the last module to be activated. The Cyborg v3.0 has been tested again without issues after the stated problems were fixed. The map has not been updated. It was also registered that the Cyborg struggled with visualizing neural activity from a file at the same framerate as other animations, the framerate has therefore been set to 3 frames per second for this kind of animations. To summarize, all modules work as intended, visualizations and sounds are activated correctly, and navigation performs adequately.

12.6 Discussion

Testing the Cyborg in Glassgården was at times quite challenging, many people displayed great interest and wanted to talk, which made it hard to

actually get any testing done. Nevertheless, attention was always part of the plan for the Cyborg, and this just proves that the Cyborg is on the right track to become a proper mascot.

Testing shows that the Cyborg covers the stated acceptance criteria. The implemented state machine is not the most advanced one, but it works well for demonstrations and should be a good starting point for further development.

It can be hard to balance the emotional feedback parameters for different states and events, for future participants to the project, a tuning or configuration guide of some sort would be greatly beneficial.

12.6.1 Navigation

A lot of time was spent on getting navigation to work properly, the localization task is still a bit slow, leading to suffering localization when the Cyborg is moving. Nevertheless, localization is held successfully, and the Cyborg is able to navigate autonomously. Sonar parameters have been tuned by trial and error in order to mitigate the issues with the sonar and the slate tiled floor, the Cyborg does not abruptly stop as often as before. Although navigation works to a satisfactory degree, there is probably still room for improvement by tuning. In order to properly solve the problems regarding localization, more powerful or distributed computers for different tasks would be beneficial, more processing power would also aid neural visualizations.

12.6.2 Cyborg Body and Hardware

The Cyborg body looks good and is ready for demonstration. The gap between the body and the LED dome looks better than expected, and active leds beneath the translucent sealing strip makes for a nice overall effect. The velcro used to prevent the backside body panel from sliding was originally planned as a temporary solution, but it works well enough to be permanent. As EiT group 3 used decals for the details on the Cyborg body, the upper

ring has not been painted blue as in the proposed vision, and should be done in the future. The finished product is seen in Figure 12.7.

12.7 Conclusion

The last tasks deemed necessary in order to make the Cyborg body and hardware ready for demonstration is completed. The gap between the LED Dome and Cyborg body is covered, protective covers have been mounted on each side of the exhaust fan, and the body panels aligned and fastened. Navigation works and the Cyborg is able to keep localization, but the lack of computing power makes localization and neural visualizations suffer. A state machine has been implemented, and the Cyborg has been tested in Glassgården. Videos of testing have been presented and commented on. Testing unearthed some issues which have been resolved, and the Cyborg now covers the stated acceptance criteria.

Figure 12.6: State diagram for the Cyborg controller.

(a) Front

(b) Back

Figure 12.7: The Cyborg v3.0.

Chapter 13

Discussion

The focus of the presented work has been on finalizing a proper foundation and getting the Cyborg to a state where it is ready for demonstration, instead of adding new complex states or other features. The configured state machine works, but some more variety on the state machine would be ideal when the Cyborg is going to be active for longer periods of time. I am confident that the software evaluation and the restructuring was the right choice. All software modules now feature a more concentrated set of tasks to handle than before, enhancing modularity. I believe that the new structure and the behavior module will make the Cyborg easier to work with and aid in configuring new behaviors for the Cyborg. I was amazed by how much attention the Cyborg attracts when tested. The current platform works very well, and it is without doubt that given some active time the Cyborg will turn into a proper Mascot. At the end of this thesis the Cyborg is ready for demonstration, and there is not much left in order to make the Cyborg ready for full time operation.

13.1 Proposed Future Work

A list of tasks tasks that was either not completed, or tasks aiding future development is presented below:

Recovery behavior - The Cyborg will get stuck or lose localization at some point, and without human intervention it will not move when this happens. As discussed in Section 3.5, some form of recovery behavior or notification to an administrator, would be preferable to have.

Docking behavior - Docking behavior is needed in order for the Cyborg state machine to select when to dock and undock the Cyborg. An almost finished docking function is implemented by the author in the navigation module, with only a few lines of code missing where commented.

Emotion configuration guide - Selecting appropriate emotional feedback values for new behaviors is hard and frustrating. A set of guidelines of some sort would be beneficial.

Change navigation stack - Although an evaluation of the navigation stack was not prioritized, I would still recommend changing the current navigation with the one presented on the ROS Wiki [33]. Although the current stack is provided as a complete solution, it has several limitations and consists almost entirely of legacy code.

New Visualizations - New visualizations will make the Cyborg more attractive and can be exploited in order to convey emotional states.

More advanced state machine - The current state machine works and presents the current features of the Cyborg. But for full time operation it might be too repeating.

Change to ROS 2 - An evaluation of ROS 1 vs ROS 2 was not prioritized, but ROS 2 offers several benefits over ROS 1 for the Cyborg. In addition, the software for object detection requires ROS 2 or a ROS bridge for communication between nodes of different ROS versions.

Integrate object detection - Object detection is implemented, but not integrated into the Cyborg. The stereoscopic camera is already bought,

and the computer for it is readily available at the university campus. Object detection requires ROS 2, which means that the system must be converted to ROS 2, or that a ROS bridge must be used between the ROS 2 node for the object detection and the ROS 1 controller node.

Computer upgrade - The computer on the Cyborg base struggles with handling navigation and visualization of neural data. A more powerful or distributed computers would greatly benefit both.

Refrost the LED dome - The frosting on the LED dome achieves diffusing the leds to some degree, but it would be preferable if the frosting was good enough to properly hide the inside of the LED dome.

Chapter 14

Conclusion

The work in this thesis brings the Cyborg to a state where it is ready for demonstration, and almost ready for full time operation. Motivation for a reworked software structure and an evaluation of the current Cyborg software has been presented. The point of the evaluation is to obtain a less complex software structure, and make the Cyborg easier to work with. The current Cyborg ROS modules have been reimplemented in accordance with the evaluation, and a behavior module has been designed and implemented. The behavior module provides a way to configure presets for commonly used output modules, and the presets can be used as states in the Cyborg state machine.

The LED controller proposed in the authors specialization project has been realized, and the interface reimplemented in order to add better memory protection and provide text animations on the whole LED dome. All corresponding software and hardware has been integrated into the Cyborg. The author has assisted groups from EiT with their work on the Cyborg, and with their collaboration the Cyborg body and LED dome has been made ready and integrated into the Cyborg. In order to make the Cyborg easier to operate, the software for the Start-up box has been upgraded and the component renamed to Mode Selector, adding the option to shut down and start new

modes. A SMACH state machine has been configured for the Cyborg, and the Cyborg has been tested on university campus. Lack of processing power on the Cyborg base makes navigation and neural visualizations suffer, and an upgrade would be beneficial. Nevertheless, the Cyborg still manages to perform adequately and the specifications for the Cyborg v3.0 are satisfied. Tasks deemed beneficial for the overall goal of the Cyborg by the author have been presented as proposed future work.

The new design of the Cyborg is finished, the new software structure features modules with more condensed task responsibility while states have been broken down into simpler entities, making the Cyborg software more modular and less complex. At the end of this thesis the Cyborg is ready for demonstration, testing shows that the current design platform works very well and the Cyborg attracts a lot of attention.

Appendices

Appendix A

Diagrams

Figure A.1: Simplified activity diagram for the Cyborg behavior module.

Appendix B

Notes and Documentation for Navigation Tuning

B.1 Tuning Guides

For navigation tuning, I recommend the guide on the ROS wiki [39], in combination with the ROS Navigation Tuning Guide [40]. Keep in mind that these both presume a navigation stack that is set ut with RViz.

B.2 Parameters in MobileEyes

In order to find out how the different localization parameters available in *MobileEyes* affected accuracy and computational load, I activated only the `ros_arnl` node, and commanded the Cyborg base between the registered locations *entrance* and *hallway* in glassgården, while changing one and one parameter. My notes for the tested parameters are presented below:

Num samples - Default value is 2000 samples. As expected, decreasing this value reduces the computational load, when decreased to 1000 samples, almost no warnings were received. I did not notice any problems

regarding accuracy or loosing localization with this value.

Grid resolution - Default value is 100. Decreasing this value by half, increases the computational load by four times, without offering any benefits for our use of the Cyborg. Increasing the value to around 200 helps with localization, but might cause problems with accuracy. More testing is needed in order to determine if an increase might be beneficial.

MaxSpeed while following path - Default value 750. This is much lower than the maximum translational speed of 1800 mm/s, after proper navigation tuning, this value can be increased if deemed necessary.

NoLocalPlanLookAhead - Default value 200. An increase in this value caused the Cyborg to behave slightly less erratic while wandering with bystanders around.

B.3 Errors Encountered

The following list presents errors I encountered while working with the `ros_arnl` module:

Duration out of dual 32-bit range - Probably caused by an incomplete update to ROS packages. Fixed by deleting the `devel` and `build` folders after updating packages, before building with catkin.

Localization task took to loong - Different versions of this message exists, all caused by a lack of computational power. A quick fix is to decrease parameter *Num samples*.

B.4 Relevant Excerpts

From [41]: "Some values we've used are a rotVelMax of 250, rotAccel of 300, rotDecel of 300, transAccel of 600, transDecel of 600. This isn't appropriate for all robots or situations, but should give you an idea of the range of values, and that the higher you set them the better and faster ARNL will drive. You

should obviously find values appropriate to your robot, your robot's payload, your robot's environment, and your robot's task. Note that for some of these you may need to change the *Top value with the configuration program for the microcontroller (some old robots shipped with low Top values). Also note that you that the gyros that we use will only give accurate readings until 300 degrees/sec, so your rotVelMax should never go over that value, and probably shouldn't come close. It is okay if the rotAccel or rotDecel is higher."

"For example, if the environment has few unmapped obstacles and is relatively fixed, the user can get away with a lower CollisionDistance parameter which will reduce the collision computation. If the robot has to navigate through narrow passages like doors, the PlanRes can be decreased to allow for more resolution. Changing some of these parameters can result in unintended consequences due to the limited resources such as the speed of the on board computer and memory. For example reducing the PlanRes by a factor will cause path planning computations to increase by the square of the factor. Path planning and obstacle avoidance can also fail to operate as expected due to parameter values."

From [42]:

One of the important considerations with any localization system is to know when it is lost. With the laser localization, we have one primary measure called PassThreshold which is the ratio of the laser readings that match with the map points to the total number of laser points. If this score falls below the PassThreshold, the Monte Carlo localization will stop trying to correct the robot pose from its sensor update step. Once this threshold is crossed under, the certainity of the robot pose will progressively get worse as the robot moves. This measure of uncertainity will be directly related to the amount of motion described by the robot kinematics and error parameters such as KmmPermm etc. This measure of uncertainity will be tracked in the Kalman filter during every cycle as a distance in the XY space. When this spread grows beyond a LostThresholdDistance the robot is considered

completely lost. The localization thread will call the fail callbacks as soon as this happens. "Note that the PassThreshold and the LostThresholdDistance are related. They come into play in sequence. If the PassThreshold is too low, say 0.1, it is unlikely that the robot will ever report itself lost using MCL. So a low PassThreshold like 0.2 and a LostThresholdDistance of 100mm is useful only for situations where the environment is fairly unchanged from the map used. In situations such as in a warehouse where a significant area of the map during its travel can be different from the map, it is better to use a higher PassThreshold such as 0.5 and a LostThresholdDistance of about 1000mm."

References

[1] A. Babayan, "Consolidating and Visualizing Biological Neural Network Activity on the NTNU Cyborg," 2018.

[2] "Pioneer LX from Adept MobileRobotics." [Online]. Available: https://www.researchgate.net/figure/Pioneer-LX-from-Adept-MobileRobotics{_}fig1{_}305986386

[3] Omron Adept MobileRobots, "PIONEER LX User's Guide." [Online]. Available: http://robots.mobilerobots.com/docs/all{_}docs/PioneerLX{_}Manual{_}revD.pdf

[4] "Arduino wiki."

[5] "NodeMCU ESP-32 Image." [Online]. Available: https://www.smart-prototyping.com/NodeMCU-32S-Lua-WiFi-ESP32-module

[6] "NodeMCU ESP-32S." [Online]. Available: https://nodemcu.readthedocs.io/en/dev-esp32/

[7] "ESP-WROOM-32." [Online]. Available: https://www.espressif.com/en/products/hardware/esp-wroom-32/overview

[8] "Installing the ESP32 Board in Arduino IDE." [Online]. Available: https://randomnerdtutorials.com/installing-the-esp32-board-in-arduino-ide-mac-and-linux-instructions/

[9] J. Waløen, "The NTNU Cyborg v2.0: The Presentable Cyborg," 2017.

[10] "Jetson TX2 Image." [Online]. Available: https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge/

[11] "NVIDIA Jetson TX2." [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules/

[12] "Zed Camera image." [Online]. Available: https://www.stereolabs.com/zed/

[13] "ZED Stereo Camera - Stereolabs Worlds Fastest." [Online]. Available: https://www.stereolabs.com/zed/

[14] "Lec 19: Signals and Signal Handling." [Online]. Available: https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/19/lec.html

[15] "Rhino 6 for Windows." [Online]. Available: https://www.rhino3d.com/

[16] "Fritzing Webpage." [Online]. Available: http://fritzing.org/home/

[17] "FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions." [Online]. Available: https://www.freertos.org/

[18] "MIT License." [Online]. Available: https://en.wikipedia.org/wiki/MIT{_}License

[19] "ROS Wiki." [Online]. Available: http://wiki.ros.org/

[20] "ROS Best Practices Mantra." [Online]. Available: http://wiki.ros.org/BestPractices

[21] "ROS Wiki - Parameter Server." [Online]. Available: http://wiki.ros.org/ParameterServer

[22] "ROS FileSystem Concepts: msg." [Online]. Available: http://wiki.ros.org/msg

[23] "ROS CMakeLists." [Online]. Available: http://wiki.ros.org/catkin/CMakeLists.txt{#}msgs{_}srvs{_}actions

[24] "smach - ROS Wiki." [Online]. Available: http://wiki.ros.org/smach

[25] "ROS Wiki - SMACH Sequence Container." [Online]. Available: http://wiki.ros.org/smach/Tutorials/Sequencecontainer

[26] "Monte Carlo Localization." [Online]. Available: https://en.wikipedia.org/wiki/Monte{_}Carlo{_}localization

[27] Sebastian Thrun, Dieter Fox, Wolfram Burgar, and Frank Dellaert, "Monte Carlo Localization for Mobile Robots Frank," *Icra*, vol. 2, pp. 1322–1328, 1999. [Online]. Available: https://www.cc.gatech.edu/{~}dellaert/ftp/Dellaert99icra.pdf{%}0Ahttp://ac.els-cdn.com/S0004370201000698/1-s2.0-S0004370201000698-main.pdf?{_}tid=ee7a0eac-c419-11e3-ae22-00000aab0f02{&}acdnat=1397510320{_}c8498bb43627ba8e16db14ba87856cb0

[28] T. R. Andersen, *Controller Module for the NTNU Cyborg*, 2017.

[29] "Student projects NTNU Cyborg." [Online]. Available: https://www.ntnu.edu/cyborg/student-projects

[30] "Work in progress - NTNU Cyborg - NTNU Wiki." [Online]. Available: https://www.ntnu.no/wiki/display/cyborg/Work+in+progress

[31] J. Waløen, "The NTNU Cyborg: Robot Hardware Infrastructure," 2016.

[32] M. Mjelva, "Control System and Object Detection System for the NTNU Cyborg," 2018.

[33] "Setup and Configuration of the Navigation Stack on a Robot." [Online]. Available: http://wiki.ros.org/navigation/Tutorials/RobotSetup

[34] "Experts in Team - Project report - Simple Neural Response Interpreter - SiNRI," 2019.

[35] "Experts in Team - Project Report Group 3," 2019.

[36] "FastLED." [Online]. Available: https://github.com/FastLED/FastLED

[37] "AaronLiddiment Github Repo." [Online]. Available: https://github.com/AaronLiddiment

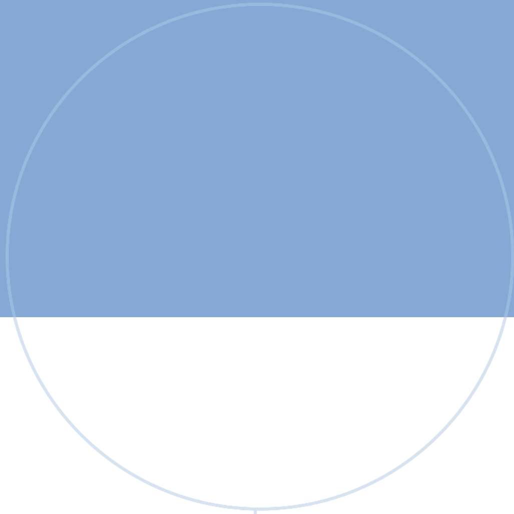
[38] "FreeRTOS task notifications, fast Real Time Operating System (RTOS) event mechanism." [Online]. Available: https://www.freertos.org/RTOS-task-notifications.html

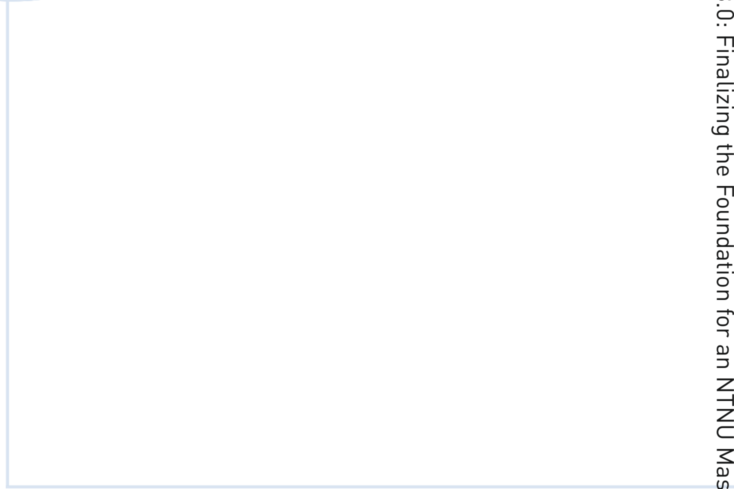
[39] "navigation/Tutorials/Navigation Tuning Guide - ROS Wiki." [Online]. Available: http://wiki.ros.org/navigation/Tutorials/NavigationTuningGuide

[40] K. Zheng, "Basic Navigation Tuning Guide," Tech. Rep. [Online]. Available: https://github.com/zkytony/ROSNavigationGuide

[41] "BaseArnl-Reference Documentation."

[42] "ARNL-Reference Documentation."





Areg Babayan

The Cyborg v3.0: Finalizing the Foundation for an NTNU Mascot

NTNU
Norwegian University of
Science and Technology