Table of Contents

1. Software

1.1. Arduino’s

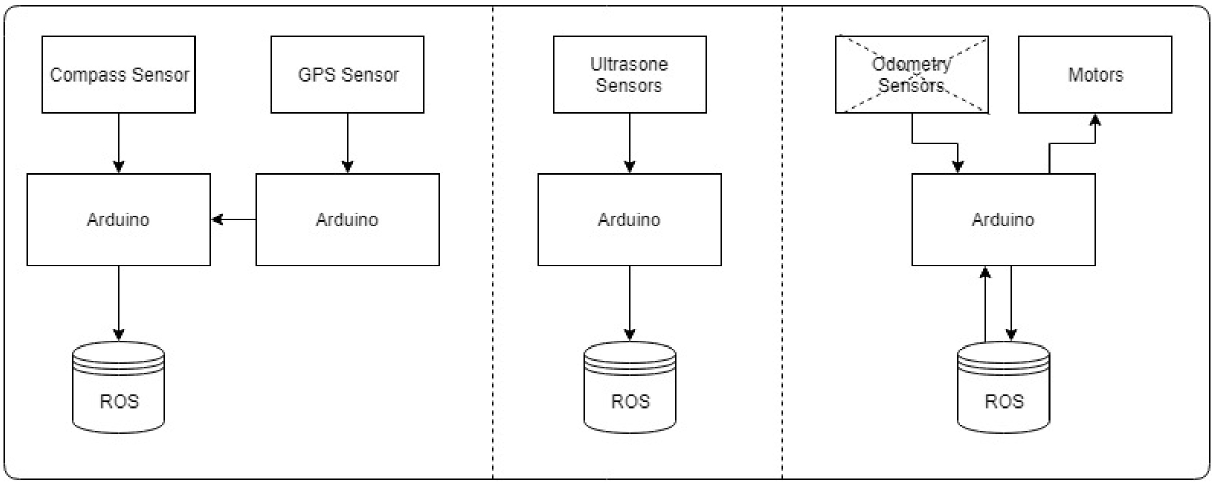
This chapter contains a overview of the code on the different Arduino’s in Willy.

The following definition list is to clear up confusion about certain terms used in this chapter.

Term	*Definition*
Code conventions	These are rules that are used for programming and drawn up by the developers of Wake 'Em.
UpperCamelCase	This means that each word is capitalized. Example: SetAlarmOfAndroid.
LowerCamelCase	This means that each word is capitalized except for the first word. Example: setAlarmOfAndroid.
Odometry	The technique of measuring the amount of movement of the wheels with special sensors.
ROS	Robot Operating System, the framework Willy has been built on.

1.1.1. System Overview

The following schematic shows an overview of how the Arduino’s communicate with ROS.

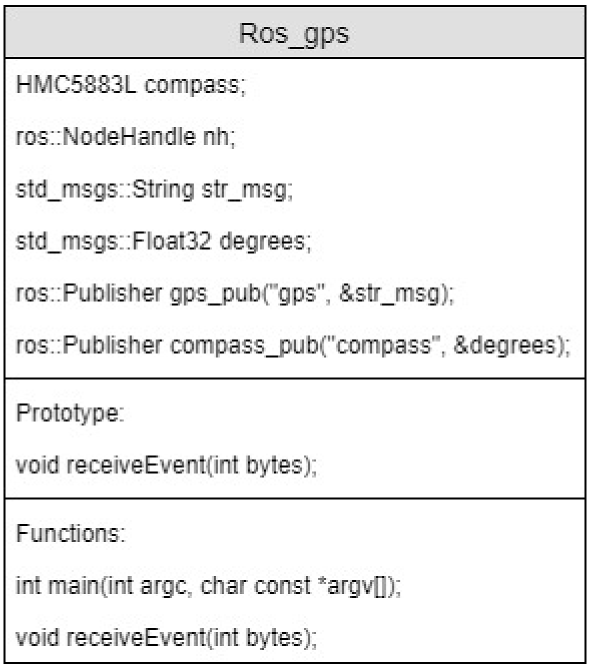


ros_gps
gps sonar odometry

It should be stated that the code for the odometry is currently removed. The reason for this can be found in the Technical Hardware document.

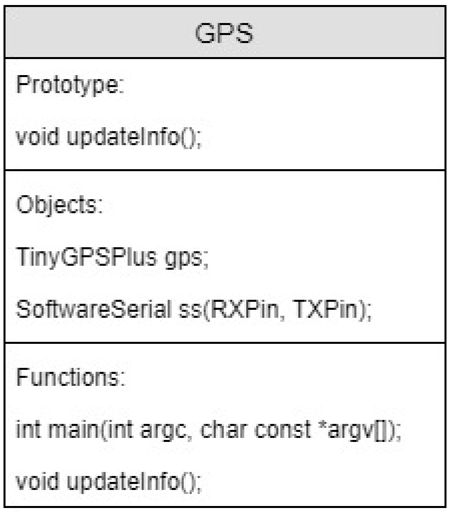
Code Overview

In this chapter the structure of the C code on the Arduino’s can be found. These schemes are not class diagrams, since C is not object-oriented.



```
Ros_gps

HMC5883L compass;
ros::NodeHandle nh;
std_msgs::String str_msg;
std_msgs::Float32 degrees;
ros::Publisher gps_pub("gps", &str_msg);
ros::Publisher compass_pub("compass", &degrees);

Prototype:
void receiveEvent(int bytes);

Functions:
int main(int argc, char const *argv[]);
void receiveEvent(int bytes);
```

This is the code of the file in ros_gps. It is running on the Arduino connected to ros. The compass sensor is attached to this Arduino.


GPS

Prototype:

void updateInfo();

Objects:

TinyGPSPlus gps;

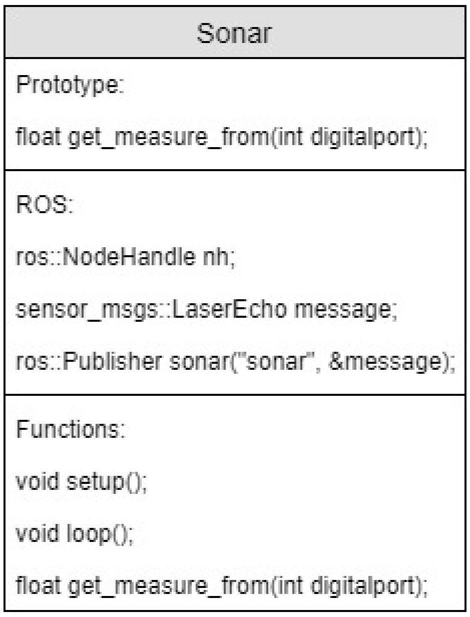
SoftwareSerial ss(RXPin, TXPin);

Functions:

int main(int argc, char const *argv[]);

void updateInfo();

This code reads out the GPS sensor and sends it with a serial connection to the other Arduino (the one running ros_gps).


Sonar

Prototype:

float get_measure_from(int digitalport);

ROS:

ros::NodeHandle nh;

sensor_msgs::LaserEcho message;

ros::Publisher sonar("sonar", &message);

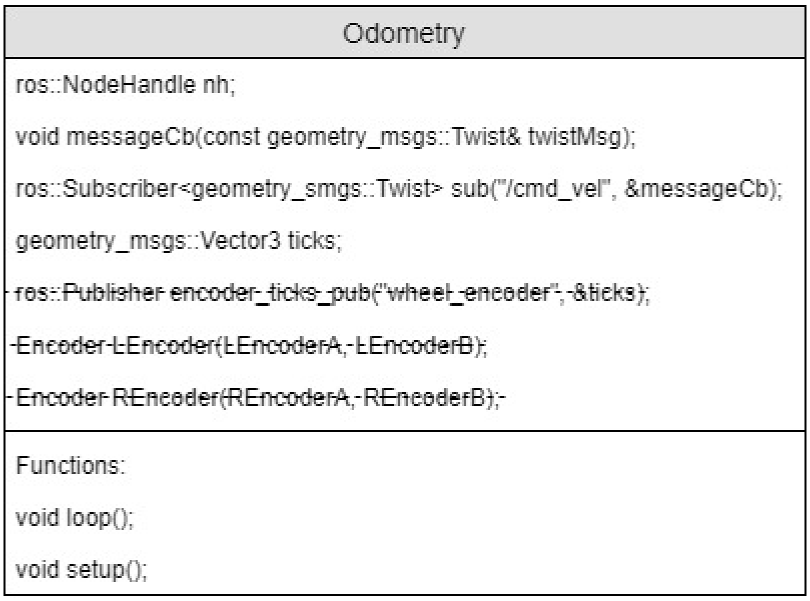
Functions:

void setup();

void loop();

float get_measure_from(int digitalport);

This is the code running the ultrasonic sensors. All the sensors have a digital pin.


Odometry

ros::NodeHandle nh;

void messageCb(const geometry_msgs::Twist& twistMsg);

ros::Subscriber<geometry_smgs::Twist> sub("/cmd_vel", &messageCb);

geometry_msgs::Vector3 ticks;

~~ros::Publisher encoder_ticks_pub("wheel_encoder", &ticks);~~

~~Encoder LEncoder(LEncoderA, LEncoderB);~~

~~Encoder REncoder(REncoderA, REncoderB);~~

Functions:

void loop();

void setup();

This is the code of the motor controller. It is called 'Odometry', but this part is removed from the code. The code receives data from ROS and controls the motors of the wheelchair by sending serial data to the built in wheelchair controller.

1.1.2. Design Decisions

In the past the decision has been made to make the hardware modular. Unfortunately the reason behind this decision is not documented.

The motor controller (odometry) is made by the group of the second semester of 2016/17. The odometry code subscribes from the topic “/cmd_vel”. The code writes data it gets from ROS to the motors.

The sonar code was reading the 10 sonar sensors and publishes it to ROS on the topic “sonar”. The code has been written by the group of the second semester of 2016/17.

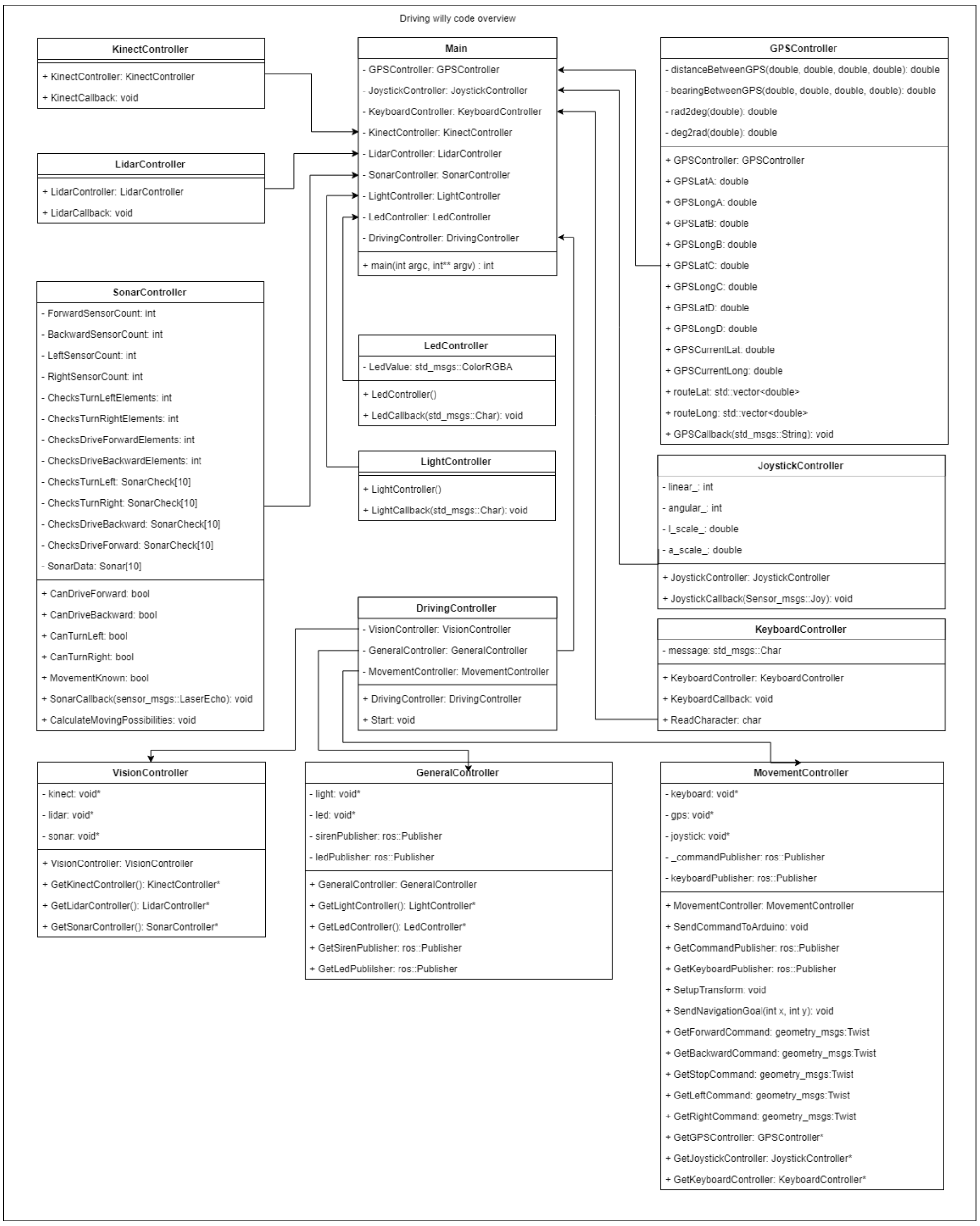
We updated the code so it uses 16 sensors, 6 for the front, 6 for the back, and 4 for facing down to the ground to detect if Willy is driving above a stair.

The GPS and compass code is written by the group of the first semester of 2017/18. The setup is made ambiguous. The compass has a Arduino and the GPS has a Arduino. The data from the GPS is sent to the Arduino with the compass. From that Arduino the data from the Compass and the data from the GPS is both being published to two separate ROS topics. The topic with compass data is “compass” and the topic with GPS data is “gps”. This setup has not been fixed yet.

1.2. Software

Here goes information about the code of the DrivingWilly ROS package

1.2.1. Overview

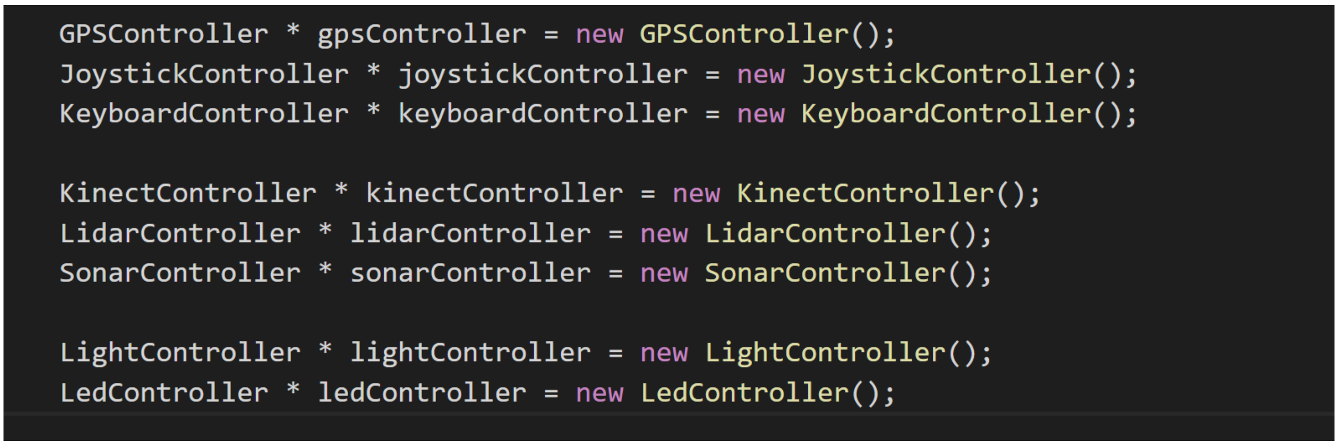

Driving willy code overview
KinectController
+ KinectController: KinectController
+ KinectCallback: void
Main
- GPSController: GPSController
- JoystickController: JoystickController
- KeyboardController: KeyboardController
- KinectController: KinectController
- LidarController: LidarController
- SonarController: SonarController
- LightController: LightController
- LedController: LedController
- DrivingController: DrivingController
+ main(int argc, int** argv) : int
GPSController
- distanceBetweenGPS(double, double, double, double): double
- bearingBetweenGPS(double, double, double, double): double
- rad2deg(double): double
- deg2rad(double): double
+ GPSController: GPSController
+ GPSLatA: double
+ GPSLongA: double
+ GPSLatB: double
+ GPSLongB: double
+ GPSLatC: double
+ GPSLongC: double
+ GPSLatD: double
+ GPSLongD: double
+ GPSCurrentLat: double
+ GPSCurrentLong: double
+ routeLat: std::vector<double>
+ routeLong: std::vector<double>
+ GPSCallback(std_msgs::String): void
LidarController
+ LidarController: LidarController
+ LidarCallback: void
SonarController
- ForwardSensorCount: int
- BackwardSensorCount: int
- LeftSensorCount: int
- RightSensorCount: int
- ChecksTurnLeftElements: int
- ChecksTurnRightElements: int
- ChecksDriveForwardElements: int
- ChecksDriveBackwardElements: int
- ChecksTurnLeft: SonarCheck[10]
- ChecksTurnRight: SonarCheck[10]
- ChecksDriveBackward: SonarCheck[10]
- ChecksDriveForward: SonarCheck[10]
- SonarData: Sonar[10]
+ CanDriveForward: bool
+ CanDriveBackward: bool
+ CanTurnLeft: bool
+ CanTurnRight: bool
+ MovementKnown: bool
+ SonarCallback(sensor_msgs::LaserEcho): void
+ CalculateMovingPossibilities: void
LedController
- LedValue: std_msgs::ColorRGBA
+ LedController()
+ LedCallback(std_msgs::Char): void
LightController
+ LightController()
+ LightCallback(std_msgs::Char): void
JoystickController
- linear_: int
- angular_: int
- l_scale_: double
- a_scale_: double
+ JoystickController: JoystickController
+ JoystickCallback(Sensor_msgs::Joy): void
KeyboardController
- message: std_msgs::Char
+ KeyboardController: KeyboardController
+ KeyboardCallback: void
+ ReadCharacter: char
DrivingController
- VisionController: VisionController
- GeneralController: GeneralController
- MovementController: MovementController
+ DrivingController: DrivingController
+ Start: void
VisionController
- kinect: void*
- lidar: void*
- sonar: void*
+ VisionController: VisionController
+ GetKinectController(): KinectController*
+ GetLidarController(): LidarController*
+ GetSonarController(): SonarController*
GeneralController
- light: void*
- led: void*
- sirenPublisher: ros::Publisher
- ledPublisher: ros::Publisher
+ GeneralController: GeneralController
+ GetLightController(): LightController*
+ GetLedController(): LedController*
+ GetSirenPublisher: ros::Publisher
+ GetLedPublilsher: ros::Publisher
MovementController
- keyboard: void*
- gps: void*
- joystick: void*
- _commandPublisher: ros::Publisher
- keyboardPublisher: ros::Publisher
+ MovementController: MovementController
+ SendCommandToArduino: void
+ GetCommandPublisher: ros::Publisher
+ GetKeyboardPublisher: ros::Publisher
+ SetupTransform: void
+ SendNavigationGoal(int x, int y): void
+ GetForwardCommand: geometry_msgs:Twist
+ GetBackwardCommand: geometry_msgs:Twist
+ GetStopCommand: geometry_msgs:Twist
+ GetLeftCommand: geometry_msgs:Twist
+ GetRightCommand: geometry_msgs:Twist
+ GetGPSController: GPSController*
+ GetJoystickController: JoystickController*
+ GetKeyboardController: KeyboardController*

Our driving willy code consists of modulair components named controllers. These controllers controls the functionality of willy. Every piece of hardware has it’s own controller which controls the methods and stores the variables. A global class diagram can be found in the image above.

By using this way of coding, functionality of willy can be easily extended. Just create a new controller and add your methods to it. A detailed explaination of this can be found in the "Extending functionality" paragraph on this page below.

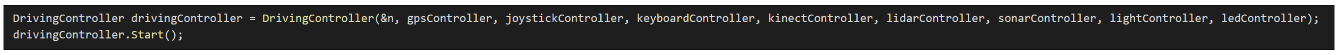
1.2.2. Controller setup

Every controller is created and declared in the main. The main creates the class object and returns a pointer to it.



```
    GPSController * gpsController = new GPSController();
    JoystickController * joystickController = new JoystickController();
    KeyboardController * keyboardController = new KeyboardController();

    KinectController * kinectController = new KinectController();
    LidarController * lidarController = new LidarController();
    SonarController * sonarController = new SonarController();

    LightController * lightController = new LightController();
    LedController * ledController = new LedController();
```

This pointer is forwarded to the controller of willy named "DrivingController".

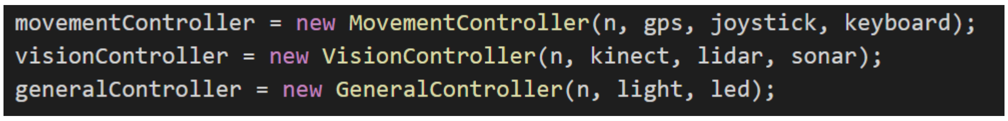


```
DrivingController drivingController = DrivingController(&n, gpsController, joystickController, keyboardController, kinectController, lidarController, sonarController, lightController, ledController);
drivingController.Start();
```

In this class the subcontrollers are declared. The current subcontrollers of willy are:

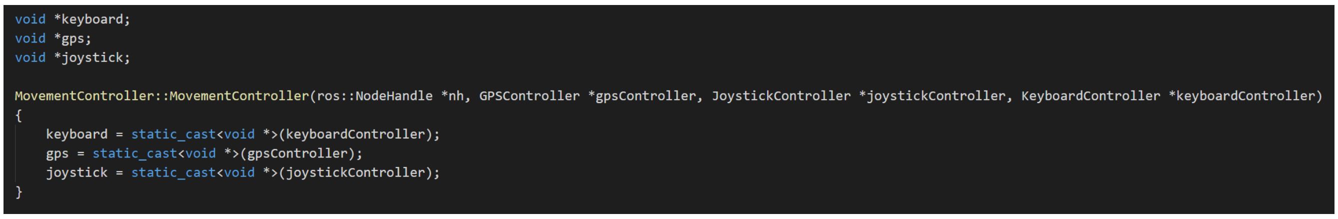
1. VisionController
2. GeneralController
3. VisionController

The pointers of the controllers are forwarded to the subcontrollers above in the "DrivingController"



```
movementController = new MovementController(n, gps, joystick, keyboard);
visionController = new VisionController(n, kinect, lidar, sonar);
generalController = new GeneralController(n, light, led);
```

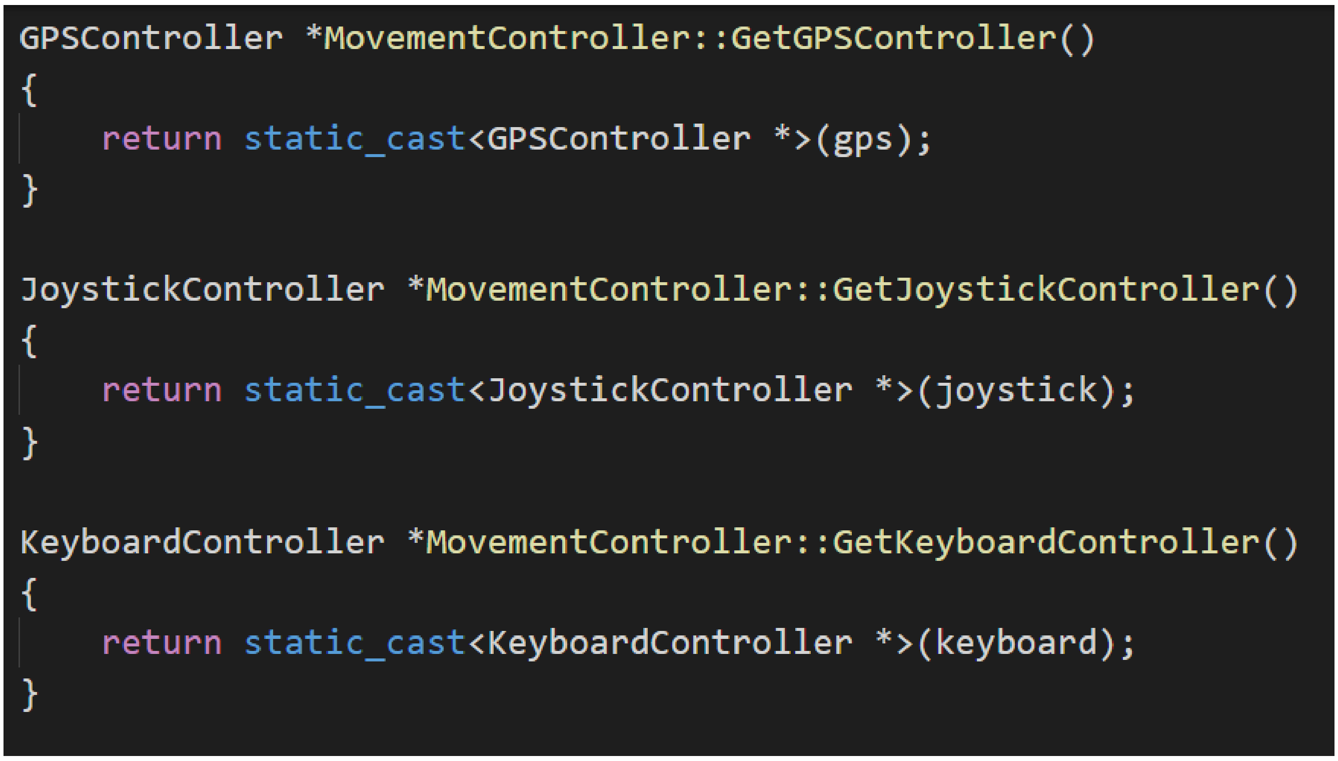
Now every controller has acces to their subscontroller and the subcontroller has acces to the main where the other controllers are declared. By using this method we can always access all data and methods of the controllers and subcontrollers. To make that possible, we have to send class object pointers between functions. Thats done by using the "static_cast" in C++



```
void *keyboard;
void *gps;
void *joystick;

MovementController::MovementController(ros::NodeHandle *nh, GPSController *gpsController, JoystickController *joystickController, KeyboardController *keyboardController)
{
    keyboard = static_cast<void *>(keyboardController);
    gps = static_cast<void *>(gpsController);
    joystick = static_cast<void *>(joystickController);
}
```

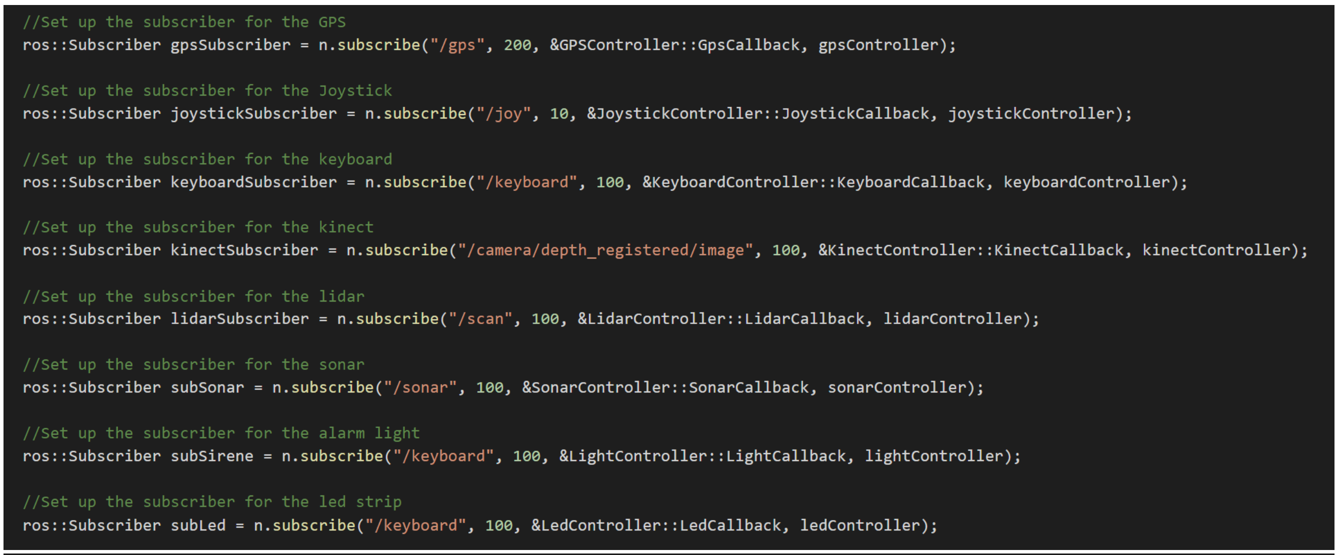
Above is an example of the "static_cast" in the "MovementController"

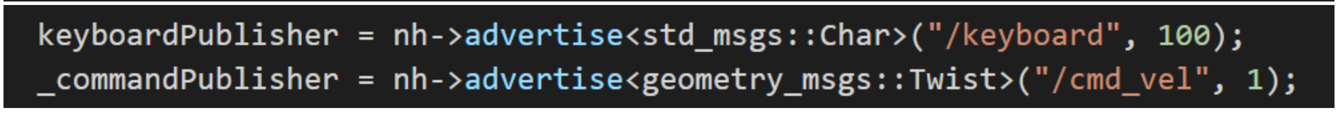
The "static_cast" make it possible to convert a void pointer back into a class object pointer without declaring the class again.


```
GPSController *MovementController::GetGPSController()
{
    return static_cast<GPSController *>(gps);
}

JoystickController *MovementController::GetJoystickController()
{
    return static_cast<JoystickController *>(joystick);
}

KeyboardController *MovementController::GetKeyboardController()
{
    return static_cast<KeyboardController *>(keyboard);
}
```

1.2.3. ROS setup

To make the communication between nodes easy accessibly, we created some advertisers and subscribers in the ROS cloud.


```
//Set up the subscriber for the GPS
ros::Subscriber gpsSubscriber = n.subscribe("/gps", 200, &GPSController::GpsCallback, gpsController);

//Set up the subscriber for the Joystick
ros::Subscriber joystickSubscriber = n.subscribe("/joy", 10, &JoystickController::JoystickCallback, joystickController);

//Set up the subscriber for the keyboard
ros::Subscriber keyboardSubscriber = n.subscribe("/keyboard", 100, &KeyboardController::KeyboardCallback, keyboardController);

//Set up the subscriber for the kinect
ros::Subscriber kinectSubscriber = n.subscribe("/camera/depth_registered/image", 100, &KinectController::KinectCallback, kinectController);

//Set up the subscriber for the lidar
ros::Subscriber lidarSubscriber = n.subscribe("/scan", 100, &LidarController::LidarCallback, lidarController);

//Set up the subscriber for the sonar
ros::Subscriber subSonar = n.subscribe("/sonar", 100, &SonarController::SonarCallback, sonarController);

//Set up the subscriber for the alarm light
ros::Subscriber subSirene = n.subscribe("/keyboard", 100, &LightController::LightCallback, lightController);

//Set up the subscriber for the led strip
ros::Subscriber subLed = n.subscribe("/keyboard", 100, &LedController::LedCallback, ledController);
```

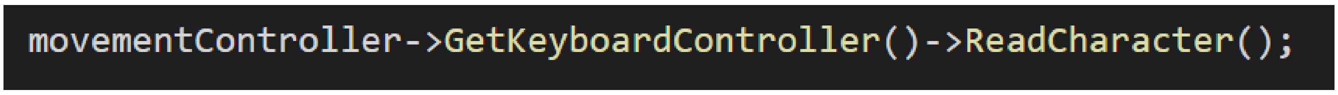

```
keyboardPublisher = nh->advertise<std_msgs::Char>("/keyboard", 100);
_commandPublisher = nh->advertise<geometry_msgs::Twist>("/cmd_vel", 1);
```

To see the basics of ROS and the general purpose, please visit our ROS generic wiki page on: //LINK TO ROS GENERAL page

These publishers and advertisers make it possible to push data generated by hardware to a rostopic, as example the sonar topic, and read that data all over the code of willy. Because you can echo the rostopic data anywhere and anytime.

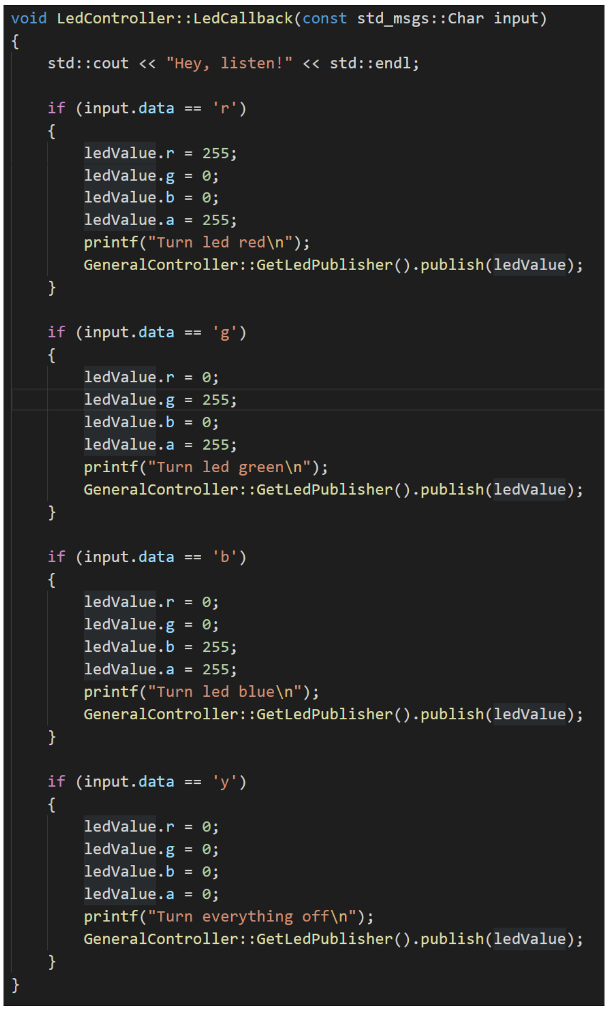
Using ROS we're able to push keyboard characters on the 'keyboard' topic and subscribe on the keyboard topic. This means that when the code is running and you pressed a key, the function that you gave to the keyboard subscriber will be launched. In our code this means that the led lighting will turn red if you press the 'r' button on the keyboard. The flowchart is shown on this page below.

Always running (DrivingController) MovementController → GetKeyboardController() → ReadCharacter()



```
movementController->GetKeyboardController()->ReadCharacter();
```

Running when char received subLed() → LedCallback() → Advertise ColorRGBA on 'led' topic()



```
void LedController::LedCallback(const std_msgs::Char input)
{
    std::cout << "Hey, listen!" << std::endl;

    if (input.data == 'r')
    {
        ledValue.r = 255;
        ledValue.g = 0;
        ledValue.b = 0;
        ledValue.a = 255;
        printf("Turn led red\n");
        GeneralController::GetLedPublisher().publish(ledValue);
    }

    if (input.data == 'g')
    {
        ledValue.r = 0;
        ledValue.g = 255;
        ledValue.b = 0;
        ledValue.a = 255;
        printf("Turn led green\n");
        GeneralController::GetLedPublisher().publish(ledValue);
    }

    if (input.data == 'b')
    {
        ledValue.r = 0;
        ledValue.g = 0;
        ledValue.b = 255;
        ledValue.a = 255;
        printf("Turn led blue\n");
        GeneralController::GetLedPublisher().publish(ledValue);
    }

    if (input.data == 'y')
    {
        ledValue.r = 0;
        ledValue.g = 0;
        ledValue.b = 0;
        ledValue.a = 0;
        printf("Turn everything off\n");
        GeneralController::GetLedPublisher().publish(ledValue);
    }
}
```

The following paragraph is about the ros navigation stack. To learn the basics about ros_navigation, see our ros navigation wiki page.

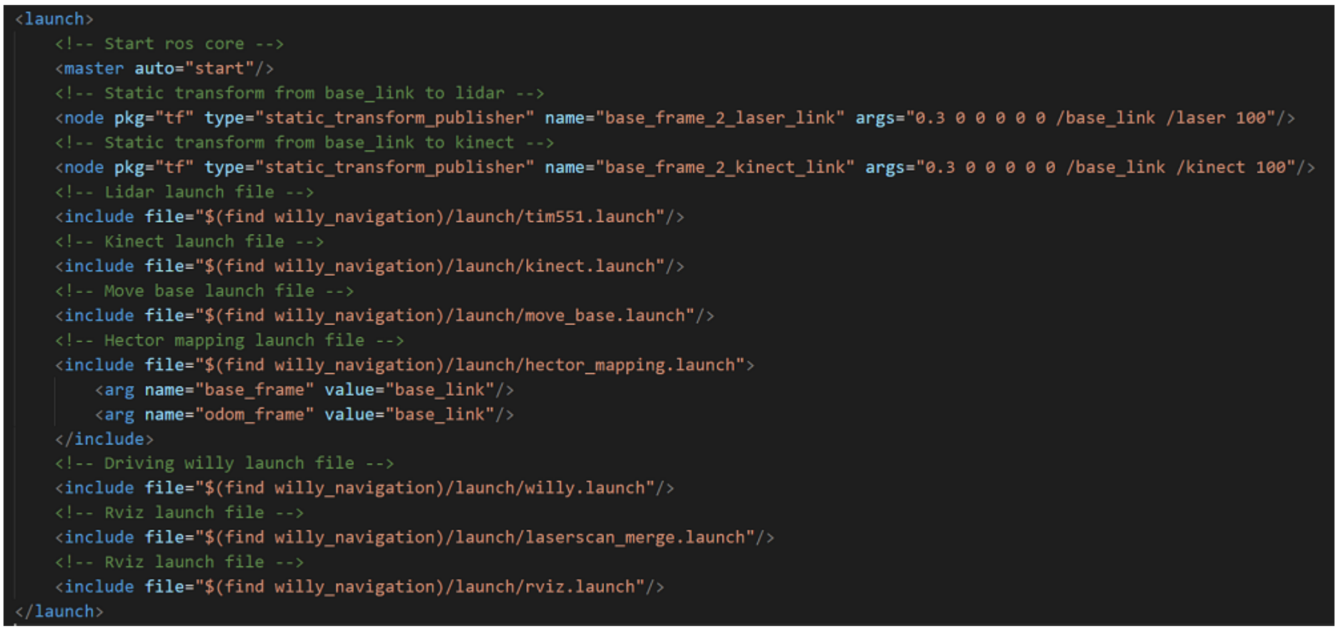
1.2.4. autonomous driving

The autonomous driving of Willy is controlled by external ROS plugins. These plugins are:

Move_base → Used for sending Twist messages to Willy out of goals send to the ros navigation stack.
Hector_Mapping → Used for creating a map based on the LiDaR and transforms.
Sick_tim551 → Used for initilization of the LiDaR and creating the /scan topic for the map.
Transform → Used the send the transformation and rotations from devices on the robot to the

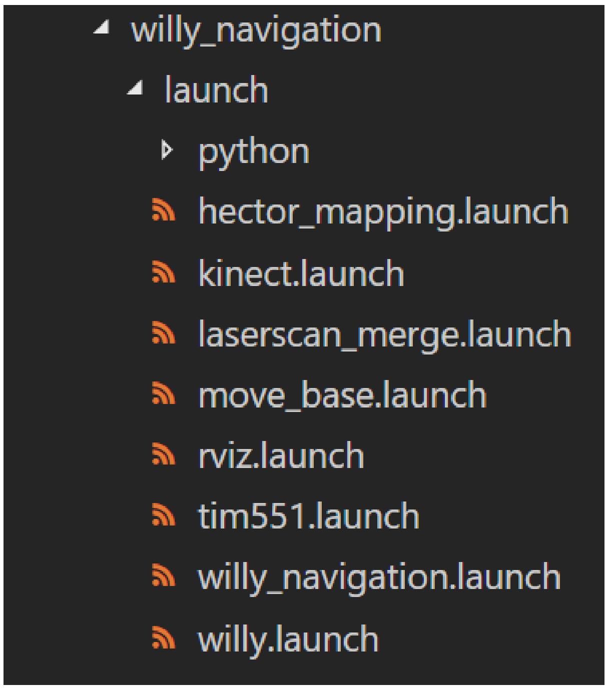
rotation point of the robot.
Kinect → Used for the recognition of people and creating the /camera topic.
Rviz → Used for visualization of the map and sensors on the base frame.

All of these external plugin are started using the 'willy_navigation.launch'.



```
<launch>
    <!-- Start ros core -->
    <master auto="start"/>
    <!-- Static transform from base_link to lidar -->
    <node pkg="tf" type="static_transform_publisher" name="base_frame_2_laser_link" args="0.3 0 0 0 0 0 /base_link /laser 100"/>
    <!-- Static transform from base_link to kinect -->
    <node pkg="tf" type="static_transform_publisher" name="base_frame_2_kinect_link" args="0.3 0 0 0 0 0 /base_link /kinect 100"/>
    <!-- Lidar launch file -->
    <include file="$(find willy_navigation)/launch/tim551.launch"/>
    <!-- Kinect launch file -->
    <include file="$(find willy_navigation)/launch/kinect.launch"/>
    <!-- Move base launch file -->
    <include file="$(find willy_navigation)/launch/move_base.launch"/>
    <!-- Hector mapping launch file -->
    <include file="$(find willy_navigation)/launch/hector_mapping.launch">
        <arg name="base_frame" value="base_link"/>
        <arg name="odom_frame" value="base_link"/>
    </include>
    <!-- Driving willy launch file -->
    <include file="$(find willy_navigation)/launch/willy.launch"/>
    <!-- Rviz launch file -->
    <include file="$(find willy_navigation)/launch/laserscan_merge.launch"/>
    <!-- Rviz launch file -->
    <include file="$(find willy_navigation)/launch/rviz.launch"/>
</launch>
```

This file starts all of the external plugin launch files. All of these external launch files can be find in our git repository.



```
willy_navigation
    launch
        python
        hector_mapping.launch
        kinect.launch
        laserscan_merge.launch
        move_base.launch
        rviz.launch
        tim551.launch
        willy_navigation.launch
        willy.launch
```

To learn more about the parameters in the launch files. See our 'parameter' wiki page.