Project ROARy report
Project course in robotics 2015-2016

THE STUDENT PROJECT TEAM

Malardalens University
February 9, 2016

1 Content-PL

Contents

List of Figures

List of Tables

2 Summery-Daniel

A garbage collecting robot was built. It was able to find, lift and transport garbage bins. The robot can localize itself in a prebuilt map, it can do waypoint following aswell as detect and avoid obstacles.

3 Background and introduction-Daniel

ROARy is part of the Project ROAR(Robot Based Autonomous Refuse handling) which aims to automatize the garbage collection in the neighborhood. The project is done as a collaboration with Volvo Group, Renova, Chalmers, Penn State University. The purpose of ROAR is to show how technology in the future can use smart machines to help out with different activities within the society.

4 Goals-Daniel

The goal of the project was to develop a robot which can collect and bring garbage bins to the refuse truck for emptying. When the bins have been emptied the robot will bring the bins back. The robot was to be integrated with the complete ROAR ROS based system i.g. a trajectory planner which provides the robot with waypoints, a task manager which handles and verifies state transitions, a truck lift system for safe bin unloading, a quadcopter system for bin detection and a user interface. For loading garbage bins, a loader mechanism needed to be constructed.

4.1 Milestones

In order to complete these goals and get deliverables early in the project, the following milestones was established.

- Remote controlled ROARy with a loader system to lift bins.
- Autonomous localization and waypoint following.
- Autonomous Pickup and Unload of bins.

5 Requirements-Daniel

The robot shall be able to move and localize itself in the environment, avoiding obstacles and planning the path around them. The robot shall be safe to operate around humans or animals. It shall not be able to collide with objects nearby the robot.

The localization and navigation system work in an partly unknown environment with moving obstacles, this will make the operation area of the robot more general and realistic and is important to the overall success of the project.

The robot shall be able to detect and register garbage bins, the truck and the ROARy platform on the refuse truck. The robot shall be able to move to and pick up bins at unspecified area. The robot shall be able to move the bins to the truck and unload the bins. The robot shall be able to return the bins to the place where they were found.

6 Limitations-Daniel

The solution will assume to have a pre-made occupancy grid map obtained from other parties within the ROAR-project, additionally, the initial pose of the garbage bin is assumed known in the map within certain error mariginals.

The ground of the working space is limited to asphalt or similar material considering the density, friction and flatness of the surface. The robot will i.e. not consider course growl or cobblestone.

The robot will not be specifically designed to coop with moving objects within the loading and unloading systems, instead the goal is to focus on functionality and performance in ideal situations.

The maximum weight of garbage bins was limited to 50kg. This limitation was set to reduce implementation complexity, however the principle for bin lifting should work for weights higher than 50kg.

7 System description-Daniel

The diagram describes the initial design of ROARy. The system is divided into 6 parts, the PSU, OBS(On Board Software), communication system, Vision system,sensor system, Loader system and Emergency stop system.

We have a PSU which provides safe power distribution to the different system parts. The OBS system is where the main navigation, perception and control software is running. The OBS also includes the ROARy interface which describes how other systems or a manual operator controls ROARy. The communication system provides an internal network aswell as a wireless connection to the truck. The Vision system provides the functionality of getting images from the camera which can be used for both a manual operator and and the bin-detecting system which runs within the OBS system. The sensor system includes p arts centered around the sensors added to the Husky platform including IMU, laserscanner and laser scanner gimbal. The loader system includes the mechanical structure for loading bins. The system includes 2 motors with external driver and embedded system for motor control. The last subsystem is the emergency stop system which provides ROARy with an additional layer of safety which is needed when the robot operates in an environment with humans. The system utilizes E-stop through sensors, stop button and a remote.

8 Planning-Daniel

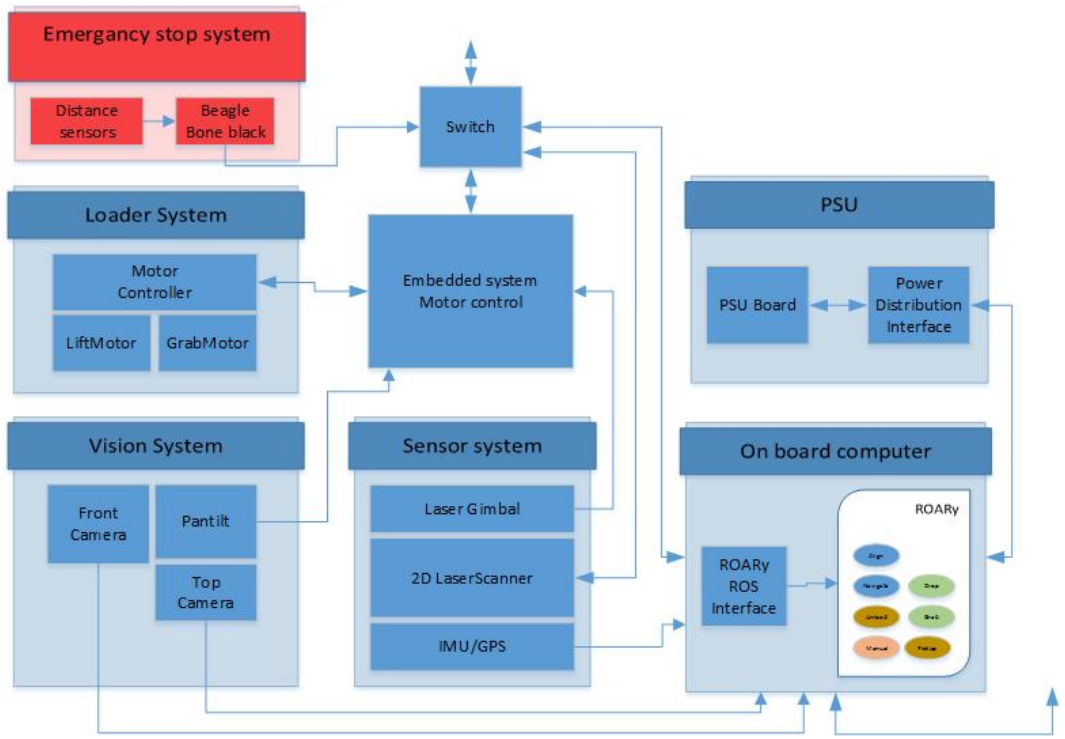
Figure **??** describes the planning of the project as well as the completion status. The missing parts is primarily in the unload program aswell as integration. The power supply software for ROS interface was not deployed and tested.

9 Emergency stop system and software-Ludvig Marcus

This part of the report will describe how the the Emergency stop system is implemented for the ROARy project.

9.1 Description

The purpose of the Emergency stop system (E-stop) is to prevent injury to people who are unfamiliar with the vehicle by shutting down its motors if they get too close. The system contains eight ultrasonic range finders to detect objects, and a Beaglebone black hardware which is used



to run all the necessary software to receive data from the ultrasonic sensors. There is also a ROS node that acts as a decision maker which decides based on the sensor readings if an object is too close or not. A second ROS node acts like a heartbeat. If no messages has been received within a certain amount of time the E-stop function will activate.

9.2 Requirements and limitations

There are two requirements for this system:

1. The robot must stop if an object with 30cm in diameter gets closer than 70-80cm to the robot.
2. It must be possible to allow the robot to get closer to the refuse bin without activating the E-stop.

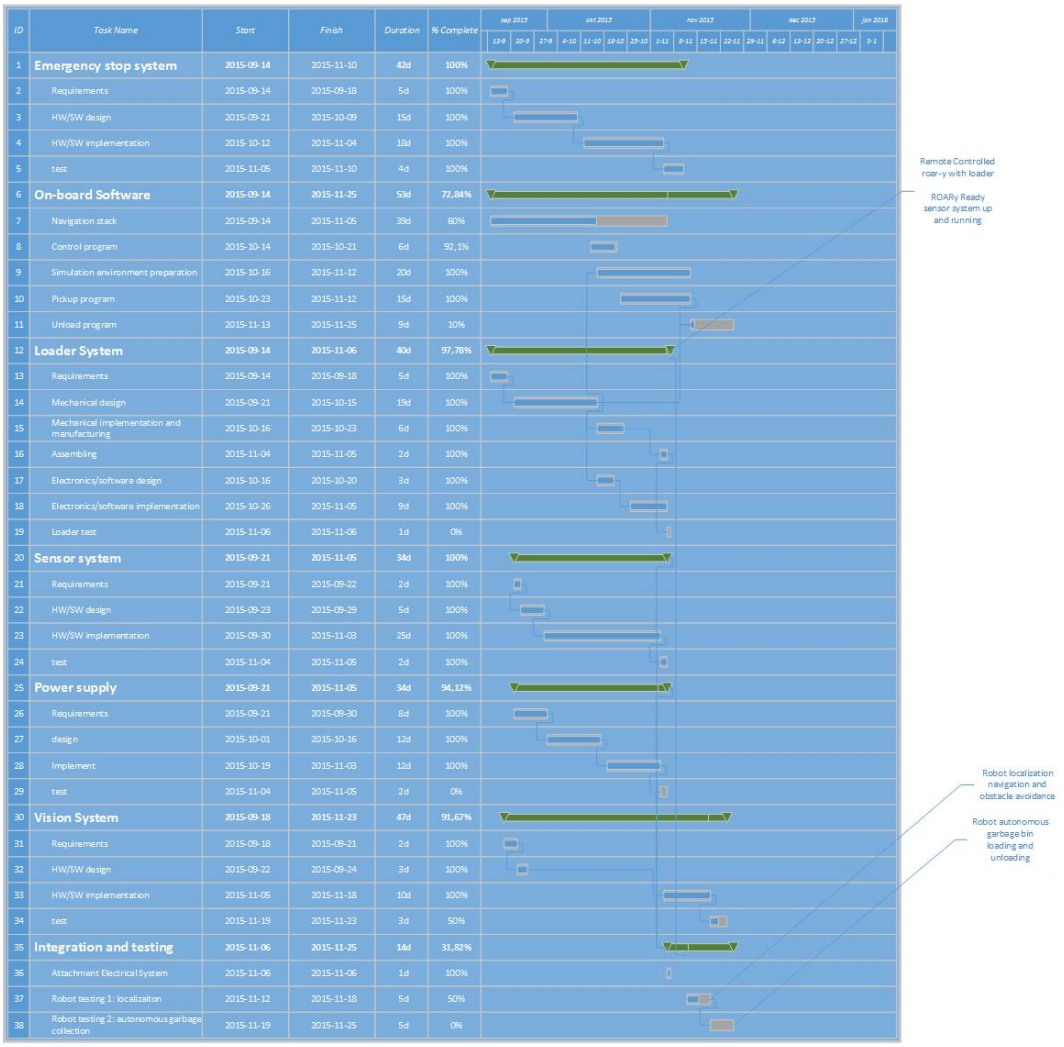
The system should have a low risk of failure since a failure could lead to injuries. In order to comply with requirement 2 it is necessary to be able to disable at least a part of the system.

To reduce the risk of failure for the whole system it is placed physically separate on its own hardware, a Beaglebone Black. The interaction between E-stop and the other systems on the robot is minimal with only a few ROS messages to:

- Disable all sensors (that would otherwise detect the refuse bin when picking it up).
- Stop the motors.
- Send heartbeat to let everyone know that E-stop is alive.

If the heartbeat signal stops the motors should also be stopped. When to disable the sensors is decided outside the E-stop system.

Currently there are a few limitations. It is at the moment not possible to connect more sensors to the system. Eight sensors is the maximum amount that the created hardware allows. There is a risk that some of the sensors may pick up the ultrasonic pulse produced by a another sensor



ID	Task Name	Start	Finish	Duration	% Complete
1	**Emergency stop system**	**2015-09-14**	2015-11-10	**42d**	**100%**
2	Requirements	2015-09-14	2015-09-18	5d	100%
3	HW/SW design	2015-09-21	2015-10-09	15d	100%
4	HW/SW implementation	2015-10-12	2015-11-04	18d	100%
5	test	2015-11-05	2015-11-10	4d	100%
6	**On-board Software**	**2015-09-14**	**2015-11-25**	**53d**	**72,84%**
7	Navigation stack	2015-09-14	2015-11-05	39d	60%
8	Control program	2015-10-14	2015-10-21	6d	92,1%
9	Simulation environment preparation	2015-10-16	2015-11-12	20d	100%
10	Pickup program	2015-10-23	2015-11-12	15d	100%
11	Unload program	2015-11-13	2015-11-25	9d	10%
12	**Loader System**	**2015-09-14**	**2015-11-06**	**40d**	**97,78%**
13	Requirements	2015-09-14	2015-09-18	5d	100%
14	Mechanical design	2015-09-21	2015-10-15	19d	100%
15	Mechanical implementation and manufacturing	2015-10-16	2015-10-23	6d	100%
16	Assembling	2015-11-04	2015-11-05	2d	100%
17	Electronics/software design	2015-10-16	2015-10-20	3d	100%
18	Electronics/software implementation	2015-10-26	2015-11-05	9d	100%
19	Loader test	2015-11-06	2015-11-06	1d	0%
20	**Sensor system**	**2015-09-21**	**2015-11-05**	**34d**	**100%**
21	Requirements	2015-09-21	2015-09-22	2d	100%
22	HW/SW design	2015-09-23	2015-09-29	5d	100%
23	HW/SW implementation	2015-09-30	2015-11-03	25d	100%
24	test	2015-11-04	2015-11-05	2d	100%
25	**Power supply**	**2015-09-21**	**2015-11-05**	**34d**	**94,12%**
26	Requirements	2015-09-21	2015-09-30	8d	100%
27	design	2015-10-01	2015-10-16	12d	100%
28	Implement	2015-10-19	2015-11-03	12d	100%
29	test	2015-11-04	2015-11-05	2d	0%
30	**Vision System**	**2015-09-18**	**2015-11-23**	**47d**	**91,67%**
31	Requirements	2015-09-18	2015-09-21	2d	100%
32	HW/SW design	2015-09-22	2015-09-24	3d	100%
33	HW/SW implementation	2015-11-05	2015-11-18	10d	100%
34	test	2015-11-19	2015-11-23	3d	50%
35	**Integration and testing**	**2015-11-06**	**2015-11-25**	**14d**	**31,82%**
36	Attachment Electrical System	2015-11-06	2015-11-06	1d	100%
37	Robot testing 1: localizaiton	2015-11-12	2015-11-18	5d	50%
38	Robot testing 2: autonomous garbage collection	2015-11-19	2015-11-25	5d	0%

Figure 1: Project planning

nearby leading to false ranges being measured. To reduce this limitation, the sensors will be divided into groups that can run at the same time, these groups are referred to as sets.

9.3 Design and interface

The system uses eight SRF05 (see Figure 2), ultrasonic ping sensors to detect objects. This is a cheap integrated circuit (IC) with a relatively large field of view. Its output is a wave, the width of which is relative to the distance. The specification for the sensor:

Figure 2: A picture of the SRF05 ultrasonic sensor

- It can detect object between 2cm and 4m.

- The frequency is 40kHz.
- Power: 5V, 4mA.

The sensors starts its ranging after it receives a trigger signal on the trigger pin. The output is a signal that is given on the echo pin. 5ms after being triggered the echo line goes high and remains high for a duration relative to the distance to the closest object. 58µs of echo line high time equals 1 cm distance to the target, the echo line will remain high for a minimum of 100µs however, hence about 2cm minimum range. If no object is detected the echo line will go low after 30ms.

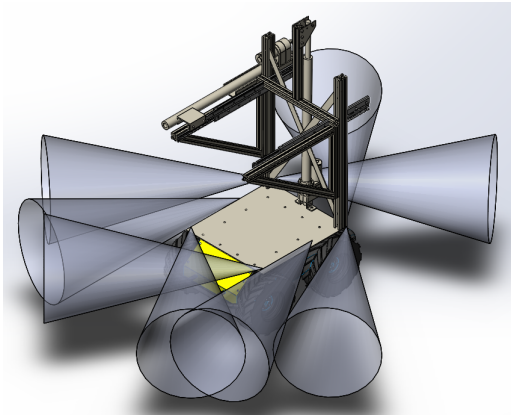


Figure 3: Sensor placement on the robot where the cones indicate the field of view of a sensor

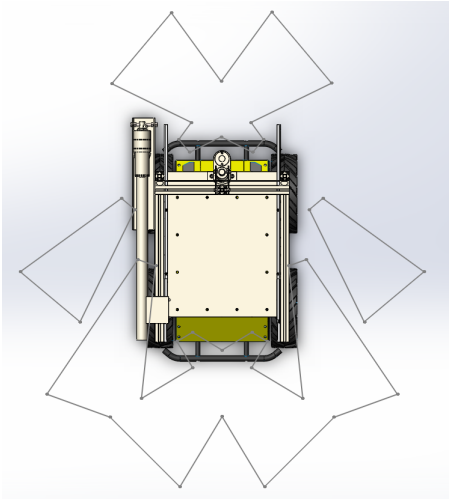


Figure 4: E-stop activation zones

In Figure 3 and 4 one can see the placement of the sensors and the E-stop activation zones. All sensor placement has been carefully thought out to cover as much area as possible. Some of the area in the rear is not fully covered since the E-stop system only consist of eight sensors. The most important area which is in front of the robot and on the sides are mostly covered. The placement and orientation of the sensors will be adjustable to some degree but no provisions have been made to allow more sensors to be connected.

The sensor consist of several connectors. In Figure 5 all the connections are listed. To send a pulse from a specific sensor the corresponding trigger pin must be set high. The echo from all of the sensors are MUXed together and connected to the interrupt pin. To listen to the echo of a

specific sensor its corresponding echo enable pin must be put high. Multiple sensors can be listen to at the same time simply by enabling their echo, however they must be triggered individually.

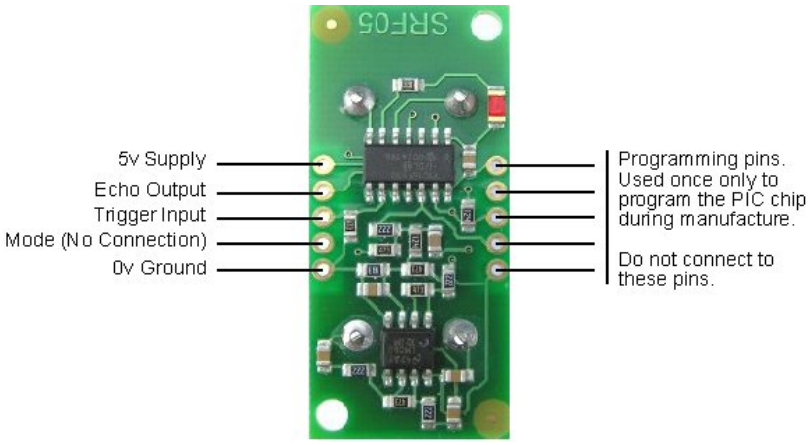


Figure 5: SRF05 ultrasonic range finder, connection schematic

9.4 Method

The software is divided into two parts. The low level, interrupt based reading of the sensor output (a kernel module) and the higher level decision making (two ROS node programs) for when to stop the motors. Sensor data is passed to the decision maker through text files where each sensor set has its own file. There are eight files available so that each sensor can be its own set and thereby be controlled individually. If the sensor reading software fails to get sensor data it will put a predefined error value (-1) in the data field.

9.4.1 Sensor reading software

This software is a Linux Kernel Module (LKM) created in the programming language C. The task of this kernel module is to detect interrupts from one GPIO pin of the Beaglebone Black and take time measurement of how long time the signal was high, (i.e. the time between a rising edge and a falling edge interrupt.) The module can be started by two different inputs: *debug* and *sets*. If *debug* is set to 1, some extra messages will be outputted into a log file stored in */var/log/kern.log*. The second input argument, *sets*, is an integer array where the user setup the sensor and set configuration. The first integer in the array is the amount of sensors that are going to be used. After that follows three integers for each sensor where the first one is which set the sensor will belong to, the second the GPIO pin for trigger, and the third one the GPIO pin for enable. Examples of how to start the kernel module can be seen in the section *Using the system* below. Down below is one example:

sudo insmod estop.ko debug=1 sets=3,1,49,47,2,115,23,2,51,45

The command above will have three sensors. The first one will be in set one, with the two GPIO pins for trigger and enable set to 49 and 47. After that comes sensor two and three that both are set to be in set two. Now, when the kernel module is up and running, it is time to describe how the module works.

First, the signal from the sensors will be read every 50ms, therefore a timer is needed. This timer will generate an interrupt, which will start a sub routine that will activate the next sensor set. It will first enable next set before disabling the current running set. The reason this is done is

so the signal from the echo pin never goes high during the switching process. This makes it easier to detect correct rising and falling edge interrupts. Once the next set has been enabled, a short trigger signal is sent to all sensors in that set. After that, everything is set up so a rising and falling edge interrupt can occur for that set.

Next, there is an interrupt routine which will run when a rising or a falling edge interrupt has occurred. During this interrupt, it will first identify if the current interrupt is a rising or a falling edge. If the interrupt type is a rising edge, the time which the interrupt occurred will be stored. After a short while, a falling edge interrupt should occur, and the time between the rising edge and falling edge will be printed out into a file. If no interrupt occur after a rising edge, an error value (-1) will be printed out into the text file. If no interrupt occur at all during one set, the same error value will be printed out.

9.4.2 ROS nodes with the decision making

The job of the decision maker is to read the sensor data from the set files and to determine if the motors should be stopped. The motors will be stopped if:

- The distance calculated from the data is shorter than the given threshold.
- The program fails to read the set files created by the kernel module.
- The data has not been updated since it was last checked.
- The data received is an error value (-1).

A flowchart can be seen in Figure 6 showing the decision maker. This decision maker is implemented in C++ in a ROS node. The ROS node is called *sensor_read* and its task is to run this decision maker every 60ms. Since the kernel module is updating every 50ms the ROS node is updating less frequently to avoid reading the same set file twice before the kernel module has updated with a new value from the sensors. As long the motors are running and the sensors

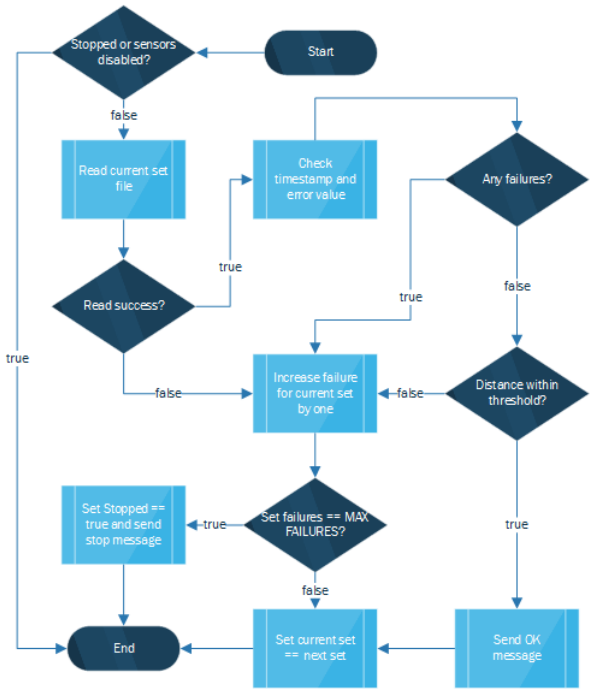


Figure 6: A flowchart over the decision maker used in a ROS program

are not disabled, then the ROS node will read a set file. If the reading is OK the program will

continue to check if the timestamp is the same as the timestamp from the previous reading of the same set file. It will also check if there is an error value inside the set file. Is everything OK so far the program will finally check if the distance is within the threshold. As long as the distance is withing the correct range an *OK* message will be sent as a ROS message to the ROS topic */ROARy/Estop/Status*. If any of the checks should fail then the current set will get a failure. Should a set receive two consecutive failures a *NOT OK* message will be sent to the same ROS topic. These two status messages will be read by a second ROS node called *estop_heartbeat*.

The *estop_heartbeat* node updates a timer every time a status message has been received. When the timer has passed 300ms then the *estop_heartbeat* will assume that something bad has happened and sends a stop message and activates the emergency stop built in the Husky. If a *NOT OK* message has been received from the *sensor_read* program the emergency stop will be activated as well. To reset the emergency stop, a ROS message containing *ROARyRestart* should be sent out to the ROS topic */ROARy/Input/Commands*.

ROS node *estop_heartbeat* is manipulating a file located in */sys/class/gpio/gpio27*. This file is called *values* and contains 0 if the GPIO pin is low and 1 if the GPIO is high. This GPIO is set as an input, and the *estop_heartbeat* node will set this GPIO to 0 when the emergency stop needs to be activated and to 1 when it needs to be deactivated. This file cannot be written to unless the owner to the file is changed from *root* to *ubuntu*. This is where the program *changeOwn* comes in. This small program change the ownership of that *value* file to *ubuntu* which allow *estop_heartbeat* to change the content of that file. Therefore, it is important to start the *changeOwn* program before starting the ROS node. More about this process is explained in the section *Using the system*.

9.5 Test and simulation results

To verify the functionality of the system a series of tests has been performed both in simulation and practically.

9.5.1 Simulated testing

Here we describe some of the simulated tests that have been done to verify the functionality and what the results of those tests have been.

For the decision making program (*sensor_read*) there are some tests with changing the variables done that are listed below:

1. Changing *i_nr_of_sets* to something other than the right amount:
    - -1: cannot compile
    - 0: crash
    - Less than the right amount: sets with higher number will get ignored.
    - More than the right amount: motors will be stopped because it will fail to read too many times.
2. Changing *i_failed_tries_threshold* to -1 gives the same results as setting it to 0, that is, the motors will be stopped when the maximum number of accepted failures is reached.
3. Changing *ix_distance_thresholds* to something not correct:
    - -1: nothing happens, just returns that an object is not too close
    - 0: same as above
4. Strange data in the set files: Letters instead of numbers: The motors will be stopped, either because timestamp will evaluate to 0 which is the initial value for *last_timestamp* or because distance will evaluate to 0 which is too close. If the distance threshold is also set to 0 or less

the motors will not be stopped but if the file still contains letters the next time it is read the timestamp will be the same so then it will be stopped. Any negative number will just result in it being interpreted as the error message.

For the second ROS node (*estop_heartbeat*) the following tests has been performed (note that it is assumed that the *sensor_read* always sends *OK* messages, these tests are only for checking the heartbeat function):

1. **Test:** Have *sensor_read* to continuously send *OK* messages that the *estop_heartbeat* receives. **Result:** As long an *OK* message is received, the E-stop function does not activate. This test has been running for over two hours with no problem. The heartbeat function seems to work.
2. **Test:** Same test as above but after a short while exit the ROS node *sensor_read*. **Result:** After *sensor_read* has been exited, it takes 300ms for the *estop_heartbeat* node to activate the E-stop function.
3. **Test:** When the E-stop is activated send the reset command *ROARyRestart*. **Result:** The heartbeat function starts up again when the *OK* messages starts to arrive again.
4. **Test:** Send in the command *ROARyDisableAllSets* to see if the heartbeat function stops. **Result:** The heartbeat function is stopped and no activation of the E-stop function is happening.
5. **Test:** Send in the command *ROARyEnableAllSets* after a *ROARyDisableAllSets* command has been sent. **Result:** The heartbeat function activates and *OK* messages are being received.

The conclusion from the tests above regarding the *estop_heartbeat* node is that the heartbeat function seems to work without any hitch.

9.5.2 Practical tests

Most of the functionality can be tested quite thoroughly using simulations but these tests aim to make sure that all parts work together. The practical tests are listed below together with their results.

1. **Test:** An object moving towards the threshold range in front of a sensor. **Result:** Once you got close, the E-stop function got activated, and a stop message is sent.
2. **Test:** Measure the distance between the sensor and the object closing in. A ruler was used to see the exactness of the read sensor data. **Result:** Between 4cm and up to the tested 70cm the accuracy was more or less perfect.
3. **Test:** Configure two sensors in two different sets and test if the E-stop function activates when closing in an object to any of them. **Result:** When closing in with an object to the first sensor, the distance decrease until the threshold is reached. The distance for the other sensor remains unchanged. When switching the object to move closer to the second object, the distance received from the second sensor got decreased while the first one increased.
4. **Test:** Configure two sensors in one set and test to move an object close to any of them. **Result:** The distance get decreased when an object moves closer to the first sensor when leaving the other sensor alone. When moving another object closer to the second sensor, the distance first decrease when the second object is closer to the second sensor than what the distance is between the first object and the first sensor.
5. **Test:** Send in a *ROARyDisableAllSets* command to disable the sensor readings. **Result:** Any object can move around freely without activating the E-stop function.
6. **Test:** Send in a *ROARyDisableAllSets* command to disable the sensor readings and move an object within the threshold range, then send in the command *ROARyEnableAllSets* to

activate the sensor reading again. **Result:** The E-stop function activates right away after the *ROARyEnableAllSets* has been sent.

7. **Test:** Move close to one sensor to activate the E-stop function, then move away before sending in the *ROARyRestart* command. **Result:** The E-stop gets reset without any problem.
8. **Test:** Move close to one sensor to activate the E-stop function, then send in the *ROARyRestart* command while standing too close to the sensors. **Result:** The E-stop gets reset without any problem but gets activated again within one second.
9. **Test:** Test all eight sensors on the robot one by one to see that all sensors can activate the E-stop function. **Result:** All sensors works great independently.
10. **Test:** Test sensors when at the same time twitch the cables that are connected at the end of the sensors. **Result:** Six of eight sensors caused no problems. Sensors that are connected to J6 and J8 can produce a couple or errors. However, the errors do not occur so often that the E-stop function activates unintentionally.

From the practical tests performed above, the conclusion is that the E-stop system works really well. The E-stop function activates when it should and when disabling the sensor reading you can move freely around the robot. It also reset when sending in the *ROARyRestart* command. There is, however, a small problem when twitching and pulling the cables at the end of the sensors. Some errors can be seen when running the *tail -f /var/log/kern.log* command. It should not cause any problem since the errors do not occur that often. It should be pointed out the the type of cables used together with the sensors has caused some problems and might in future be a good idea to change them to be 100% sure that no errors occurs.

9.6 Using the system

In this section there will be descriptions on how to use the E-stop system. The following section will cover how to run the sensor reading program (kernel module), and the ROS nodes. For the coming sections it is expected that everything is running on the Beaglebone Black that are inside the ROARy robot. To log in into the Beaglebone Black one first must connect a computer to the same network as the ROARy called ROARy-fi. The password for the wireless connection is *roary*. Once connected, use *ssh* to connect to the Beaglebone Black: *ssh ubuntu@192.168.0.111* where *ubuntu* is the user name. When inside, type the password *temppwd* and then you should be logged in.

9.6.1 Sensor reading program

Below in this section there will be step by step explanations for how to run the sensor reading program called *estop* and how to operate it.

How to compile and run the Linux Kernel Module (LKM)

1. Go to the folder where the program files are located (in Beaglebone black: */home/ubuntu/kernelModule/estop*).
2. Run the command *make* to start the compilation. When this is done a kernel object will be created (*estop.ko*).
3. The kernel module starts automatically at startup, so the kernel module needs to be removed first before it can be loaded again. To remove a kernel module run: *sudo rmmod estop*.
4. To load the kernel module run the command *sudo insmod estop.ko*.
5. A good idea is to check if the LKM is producing any error. To do so login into the Beaglebone Black in a new terminal window and type *tail -f /var/log/kern.log*. What will be displayed are all messages outputted by all the loaded kernel modules.

Run the LKM by manually setting up the sensor sets
In case one would want to change the default sensor setup that is configured in line 42 in *estop.c* there is a parameter one can use to override that: *sets*. In Table 1 a list of all connectors with their respective trigger and enable pins are displayed. Down below there will be some examples how to use the *sets* parameter.

1. Go to the folder where the kernel object exists.
2. If the kernel module is already running run *sudo rmmod estop* to remove it.
3. To activate a sensor connected to J2 run: *sudo insmod estop.ko sets=1,1,48,46*.
4. To activate sensors connected to J6 and J9 run: *sudo insmod estop.ko sets=2,1,50,69,1,49,47*. These two sensors will be in the same set.
5. To activate sensors connected to J2, J3, and J4 run: *sudo insmod estop.ko sets=3,1,48,46,2,31,26,2,30,44*. J2 will be alone in set one while J3 and J4 will be in set two.

Table 1: Showing what trigger and what enable pins are used for each connectors

Connector	Trigger pin	Enable pin
J2	48	46
J3	31	26
J4	30	44
J5	60	68
J6	50	69
J7	51	45
J8	115	23
J9	49	47

Run the LKM in debug mode

1. Go to the folder where the kernel object exists.
2. To run the LKM in debug mode run the following command: *sudo insmod estop.ko debug=1*.
3. Run the command *tail -f /var/log/kern.log* in a new terminal window. Using the *debug* parameter more messages will be printed out in the *kern.log* file.

How to run the LKM at startup

1. Go to the folder where the kernel object is located.
2. Copy the kernel object to another folder using the following command: *sudo cp estop.ko /lib/modules/4.x.x-linux_kernel/kernel/drivers* where *4.x.x-linux_kernel* is the folder with the kernel that is currently running. Usually it is the folder with the highest version number.
3. Edit the file *modules*: *sudo nano /etc/modules*. At the bottom write the module you want to load at startup. Write the module name without the *.ko* extension. To load kernel module *estop* write *estop* only.

9.6.2 The ROS nodes *sensor_read* and *estop_heartbeat*

Here the ROS nodes will be explained how to run them. There are two ROS nodes: *sensor_read* and *estop_heartbeat*. Both nodes are already started when the Beaglebone Black starts up. To quit the ROS nodes run the command *ps -e* to receive a list of running processes. Look up for ROS names *sensor_read* and *estop_heartbeat* and look for the PID to the left. Use the PID in the following command to kill the process: *sudo kill -9 PID* where PID is the ID of the process. Before you can run any ROS related commands one must run the following commands:

1. *export ROS_HOSTNAME=192.168.0.111*
2. *export ROSP_IP=192.168.0.111*
3. *export ROS_MASTER_URI=http://192.168.0.200:11311*
4. *source /home/ubuntu/catkin_ws/devel/setup.bash*

The commands above allow the Beaglebone Black to connect to the ROS master located on the Huskys onboard PC. The last command allows one to use the ROS commands in the current terminal. These four commands needs to be executed for each new terminal window you open.

How to compile and start the ROS nodes

1. Go to the folder */home/ubuntu/catkin_ws*.
2. After modifying the source files in the folder */home/ubuntu/catkin_ws/src/estop_system/src* run the command *catkin_make*.
3. Before running the ROS nodes both the kernel module and a small program *changeOwn* must be loaded and started. To start *changeOwn* go to the folder */home/ubuntu/software* and run the command *sudo ./changeOwn*. Note that the kernel module *estop* needs to be loaded since the file that *changeOwn* use is created by the kernel module.
4. After the compiling is done you can start the ROS nodes by running *rosrun estop_system sensor_read* and *rosrun estop_system estop_heartbeat*.
5. If the compiling did not happen, that could be because the date is not set, or that the date is too far off between the system time and the time which the files was modified. Since there is no battery on the Beaglebone Black, the date get reset after every shutdown. Set the date: *sudo date –set "yyyy-mm-dd hh:mm"* and do a small modification to the files you want to compile, and then try to run *catkin_make* again.

Run ROS node with the parameter nSets
The default value for number of sets is set to 3. This can be changed at startup using the *nSets* parameter.

1. Run the ROS node with the following command: *rosrun estop_system sensor_read _nSets:=2* to change the default value to 2.
2. It is important to set the *nSets* parameter to the same value as the number of set files that the kernel module is updating.

Using ROS messages

1. At startup the sensor reading is disabled. To enable it send the ROS message *ROARyEnableAllSets* to the ROS topic */ROARy/Input/Commands* by running this command: *rostopic pub /ROARy/Input/Command std_msgs/String -1 "ROARyEnableAllSets"*.
2. To disable all sets run: *rostopic pub /ROARy/Input/Command std_msgs/String -1 "ROARyDisableAllSets"*.
3. If the emergency system for the Husky has been activated run: *rostopic pub /ROARy/Input/Command std_msgs/String -1 "ROARyRestart"* to reset.

Explanation of the startup script
In location */home/ubuntu* there is a file called *startup.sh*. This file contains a number of commands that are executed at startup of the Beaglebone Black. The list below will go through them line by line.

- Lines 2-7 are needed to connect to the ROS master located on the Huskys onboard PC.

- Line 9 adds a device tree object to the cape manager. This object sets two enable GPIO pins to low. This is necessary because when the sensors that use these two pins are not used, they will disturb the whole E-stop system when they are high. The file *BB-GPIO-ROARY.dtbo* is located in */lib/firmware*.
- Line 10 load all ROS commands so they can be used later for line 13-14.
- Line 12 start the program *changeOwn*. This program change the ownership of one file that the ROS node *estop_heartbeat* use. This program must be started before the ROS node.
- Line 13 start the ROS node *estop_heartbeat*.
- Line 14 start the ROS node *sensor_read* with the parameter *nSets* set to 2.

The file above (*startup.sh*) is being executed at startup. To make this happen you edit the file */etc/rc.local*: *sudo nano /etc/rc.local*. There are three important lines:

- Line 1 is needed for all shell script files. Without this line, the script will not work.
- Line 13 is executing the *startup.sh* file in */home/ubuntu*
- Line 15 is also important and is needed like line 1.

***sensor_read* variables that are useful to know**

- *b_all_sensors_disabled*: if this boolean is set to true then the sensor reading will be disabled. To enable the sensor reading a ROS message *ROARyEnableAllSets* is needed. If the robot is moving close to a refuse bin the ROS message *ROARyDisableAllSets* can be used. This boolean is set to true as default.
- *b_stopped*: this boolean value will be set to true once too many failures has occurred. To reset this boolean to false again send in the ROS message *ROARyRestart*.
- *i_nr_of_sets*: this is the number of sets in use. If this value is lower than the actual number of sets in use, not all data will be read. If it is higher it will fail to read the data and stop the motors once the threshold for read fails has been reached. This value can also be set using a parameter during the startup of the ROS node. The value itself is initiated in the main function.
- *i_failed_tries_threshold*: The number of failed reads before stopping is controlled with this variable. The value should be determined with the number of sets in mind as that will affect the actual time elapsed between not getting information from the sensor and stopping.
- *ix_distance_thresholds[]*: The threshold for what distance is considered too short can be assigned with this array. The value at a given index is the distance in cm for the set with the same number as the index. The first index (0) in the array is not used.

9.7 Version

The files used for this E-stop system only have one version and their location is all on the Beaglebone Black hardware. The kernel module is located at */home/ubuntu/kernelModule/estop*. In this folder there are two files: *estop.c* and *Makefile*.

For the ROS nodes, there are two of them. These are located at */home/ubuntu/catkin_ws/src/estop_system/src*. In this folder there are two files: *sensor_read.cpp* and *estop_heartbeat.cpp*.

In the folder */home/ubuntu/software* you will find the program *changeOwn* as well as the code file *changeOwn.cpp*.

9.8 Future work

One possible future work is to extend the E-stop system to be able to handle more than eight sensors. By doing this, more area could be covered around the robot. Another is to change the cables to decrease the risk of getting errors when twitching the cables that are connected at the end of the sensors.

10 Emergency stop electronics-Roxanne emil

10.1 Description and Requirements-Roxanne

For the Beagle Bone Black [**?**] it has been decided to create our own cape or shield. A cape is made so it can be connected directly on the Beagle Bone without any cables. This particular cape is made for the Emergency Stop System which is a separate system on ROARy where we use distance sensor to detect if an object is to close. If that is the case system will stop ROARy completely so no accidents will occur. The cape must include connectors for all eight sensors and some kind of logic to select sensors and to only send the shortest distance to the Beagle Bone Black.

10.2 Design and Interface-Roxanne

For this system, there are eight ultra-sonic sensors [**?**] which all can be enabled and disable depending on which direction is relevant. On the cape there will be eight multiplexers, which is there for the enabling, one for each sensor. Since the logic level on the Beagle Bone Black is 3.3 volts and the logic level for the sensors is 5V voltage level translator are placed on the cape to convert the voltage levels. The only interesting distance is the shortest one and to receive the shortest distance we are using AND logic. The inputs of the AND logic gates are the enabled echo signals and the output is the shortest echo at that time. This signal is sent to the interrupt on the Beagle Bone Black and then converted into distance. For details about the components used, see table 4 on page 22.

10.3 Testing-Roxanne

When running the functions of the card, it needs to be connected directly to the Beagle Bone Black. There is two led lights, one for 3.3V and one for 5V that should lit up when the Beagle Bone Black is powered. You can also measure the voltage on pins 4 and 6 on the P9 connector. There is eight sensors connected to the cape and in ROARys case, the cable are long and in need of many connectors so it can be easily disabled. This means that one need to focus extra on the mapping of the cables and connectors and to make sure that no glitches appears. The current mapping of the connectors can be seen in figure 7 on page 22.

10.4 Using the system-Roxanne

All sensors will always be triggered, that is, the sensor will always send out a pulse. By using multiplexers it can be decided which of the sensor echo should be to listened to. Each sensor is connected to a separate multiplexer which the user can enable or disable through the GPIO pins on the Beagle Bone Black. When disabled, the signal will always send a high signal, and when enabled the signal will be high until an object is detected, then the signal will turn low. By using AND logic gates, the low signal can be detected and send to the interrupt pin on the Beagle Bone Black. The logic of the multiplexer and the AND gate can be seen in table 2 and 3.

S	I1	I2	Z
LOW	X	LOW	LOW
LOW	X	HIGH	HIGH
HIGH	LOW	X	LOW
HIGH	HIGH	X	HIGH

Table 2: Multiplexer Logic

U_N - Z	U_M - Z	$U_1$2 - Echo Out
HIGH	HIGH	HIGH
HIGH	LOW	LOW
LOW	HIGH	LOW
LOW	LOW	LOW

Table 3: AND Logic

10.5 Versions-Roxanne

The first version of the cape was called "Adapter UART RS 232". That version has only components for an adapter for UART to RS 232 and it was never manufactured. On the next version, which is named "BeagleBoneBlack_Cape_v2", there has been added everything for the sensorsystem. This version was manufactured in school and is fully working. The last and current version is called BeagleBoneBlack_Cape_v3. On this version, the adapter has been removed . The card was ordered from Würth, and the components was placed in school. Also fully working, except the leds are placed in the wrong direction.

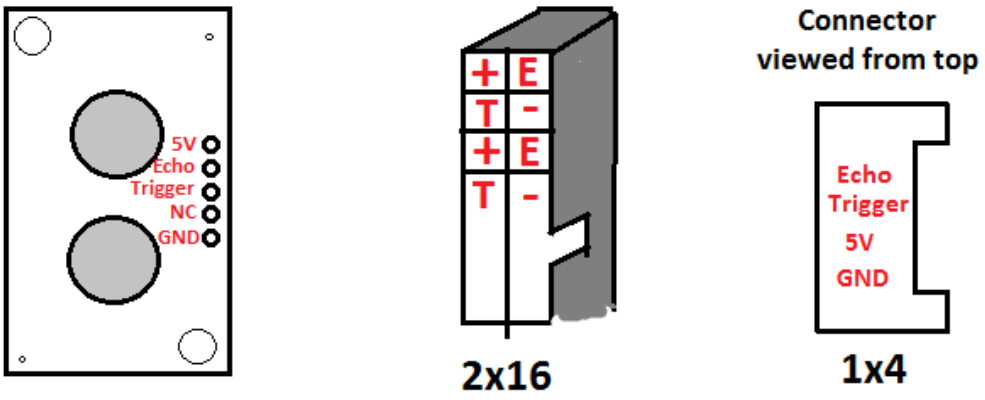
10.6 Future Work-Roxanne

The main function of the card is completed and the result was what we were hoping for but there is always room for improvement. Design and mapping can always be improved. For example, I would have liked to change the mapping of the sensor connectors so it would match the sensors mapping better. The echo and trigger signal sends data and the data was good but it could be better by adding for example capacitors.

10.7 Appendices-Roxanne

BOM					
Name	Vendor	Ordernr	RefDes	Value	Shape
Ceramic Capacitor	Würth	885012106006	C5	10uF	CAPC1608X95N
Ceramic Capacitor	Würth	885012106006	C6	10uF	CAPC1608X95N
Connector	Würth	691323100002	DIG_OUT	2 pinheader with flanges	2 Pin header with flanges
2.54mm Locking Header	Würth	61900411121	J2	HDR1X4	HDR1X4
2.54mm Locking Header	Würth	61900411121	J3	HDR1X4	HDR1X4
2.54mm Locking Header	Würth	61900411121	J4	HDR1X4	HDR1X4
2.54mm Locking Header	Würth	61900411121	J5	HDR1X4	HDR1X4
2.54mm Locking Header	Würth	61900411121	J6	HDR1X4	HDR1X4
2.54mm Locking Header	Würth	61900411121	J7	HDR1X4	HDR1X4
2.54mm Locking Header	Würth	61900411121	J8	HDR1X4	HDR1X4
2.54mm Locking Header	Würth	61900411121	J9	HDR1X4	HDR1X4
Diode	School	School	LED5V	LED_blue	CAPC1608X95N
Diode	School	School	LED33V	LED_blue	CAPC1608X95N
Connector	Würth	61306421121	P8_CON	HDR2X23	HDR2X23
Connector	Würth	61306421121	P9_CON	HDR2X23	HDR2X23
Resistor	School	School	R1	2.2kOhm	RESC1608X63N
Resistor	School	School	R2	2.2kOhm	RESC1608X63N
Multiplexer	Farnell	1467314	U3	NC7SZ157P6X	NC7SZ157P6X
Multiplexer	Farnell	1467314	U4	NC7SZ157P6X	NC7SZ157P6X
Multiplexer	Farnell	1467314	U5	NC7SZ157P6X	NC7SZ157P6X
Multiplexer	Farnell	1467314	U6	NC7SZ157P6X	NC7SZ157P6X
Multiplexer	Farnell	1467314	U7	NC7SZ157P6X	NC7SZ157P6X
Multiplexer	Farnell	1467314	U8	NC7SZ157P6X	NC7SZ157P6X
Multiplexer	Farnell	1467314	U9	NC7SZ157P6X	NC7SZ157P6X
Multiplexer	Farnell	1467314	U10	NC7SZ157P6X	NC7SZ157P6X
AND logic gate	Farnell	1607816	U11	SN74HC21DR	SOIC127P600X175-14N
AND logic gate	Farnell	1631684	U12	74LVC1G08GW	74LVC1G08GW
Leveltranslator	Farnell	2496497	U13	TXS0101DCKR	TXS0101DCKR
Leveltranslator	Farnell	1702548	U14	TXS0108EPWR	SOP65P640X110-20AN

Table 4: BOM



8

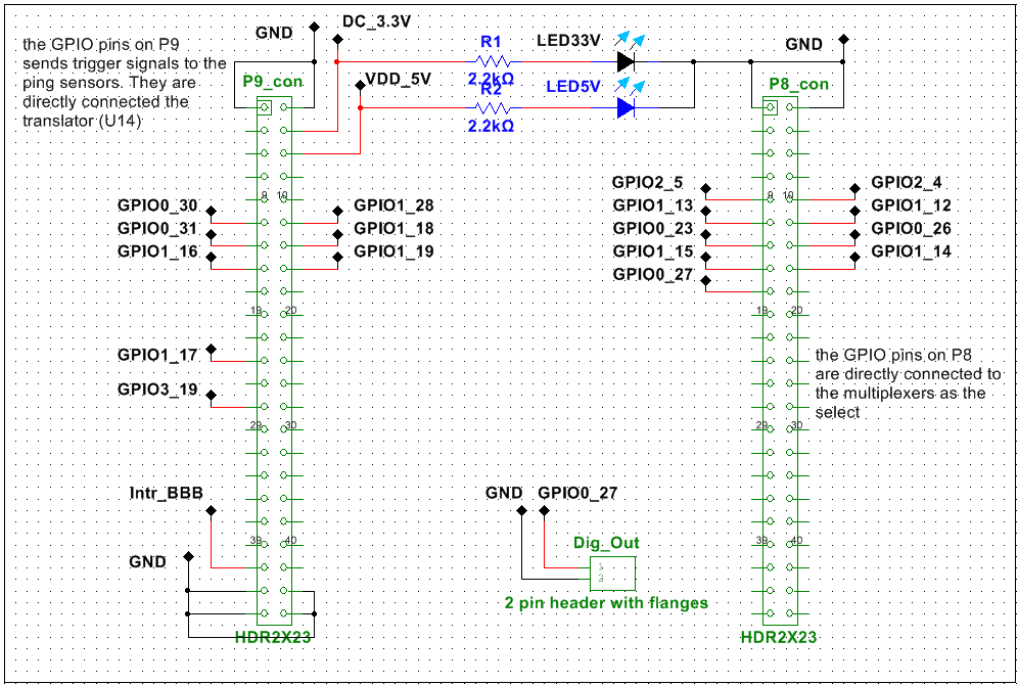
Figure 7: A image of the cablemapping, from the sensors to the cape

P9 Connector			
Pin	Name	Function / Connection	Comment
1,2	GND	Ground Plane	
4	DC 3.3V	Leveltranslator: U13, U14	
6	VDD 5V	J2-J9 and U3-U14	
11	GPIO0_30	LvlTransU14 – trigg – J4	Enable sensor trigger
12	GPIO1_28	LvlTransU14 – trigg – J5	Enable sensor trigger
13	GPIO0_31	LvlTransU14 – trigg – J3	Enable sensor trigger
14	GPIO1_18	LvlTransU14 – trigg – J6	Enable sensor trigger
15	GPIO1_16	LvlTransU14 – trigg – J2	Enable sensor trigger
16	GPIO1_19	LvlTransU14 – trigg – J7	Enable sensor trigger
23	GPIO1_17	LvlTransU14 – trigg – J9	Enable sensor trigger
26	GPIO3_19	LvlTransU14 – trigg – J8	Enable sensor trigger
41	Intr_BBB	LvlTransU13 - EchoOut	
43-46	GND	Ground Plane	
P8 Connector			
Pin	Name	Function / Connection	Comment
1,2	GND	Ground Plane	
9	GPIO2_5	Select – Multiplex U6	Set high to enable
10	GPIO2_4	Select – Multiplex U7	Set high to enable
11	GPIO1_13	Select – Multiplex U5	Set high to enable
12	GPIO1_12	Select – Multiplex U8	Set high to enable
13	PIO0_23	Select – Multiplex U4	Set high to enable
14	GPIO0_26	Select – Multiplex U9	Set high to enable
15	GPIO1_15	Select – Multiplex U3	Set high to enable
16	GPIO1_14	Select – Multiplex U10	Set high to enable
17	GPIO0_27	Digital Out to relaycard	

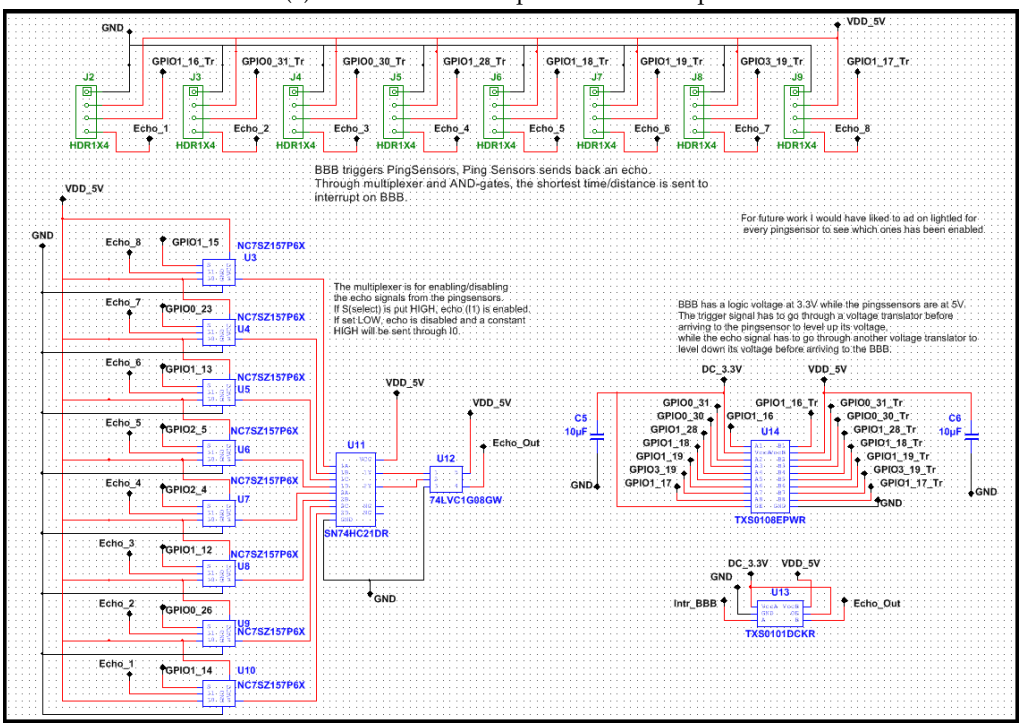
Table 5: Pin Specification for extension card

10.7.1 Schematics and CAD-Roxanne

The schematics have made in multisim and the CAD files have been made in ultiboard.

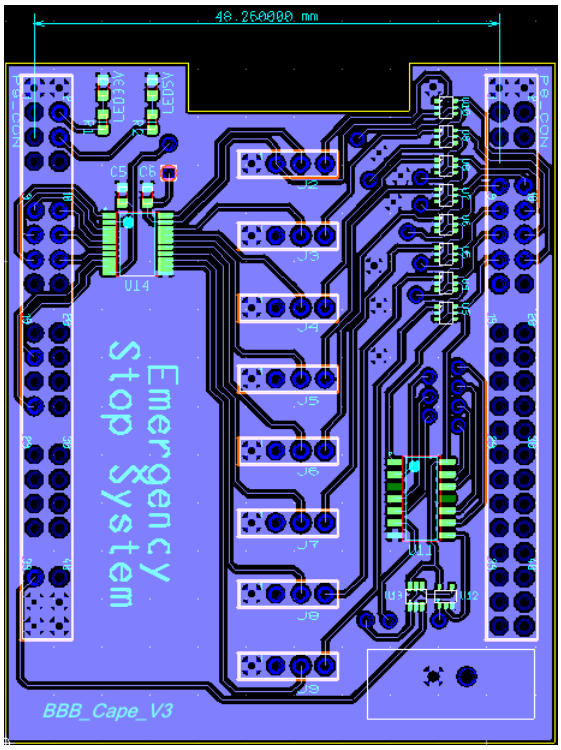


(a) Schematics for the pinouts on the cape

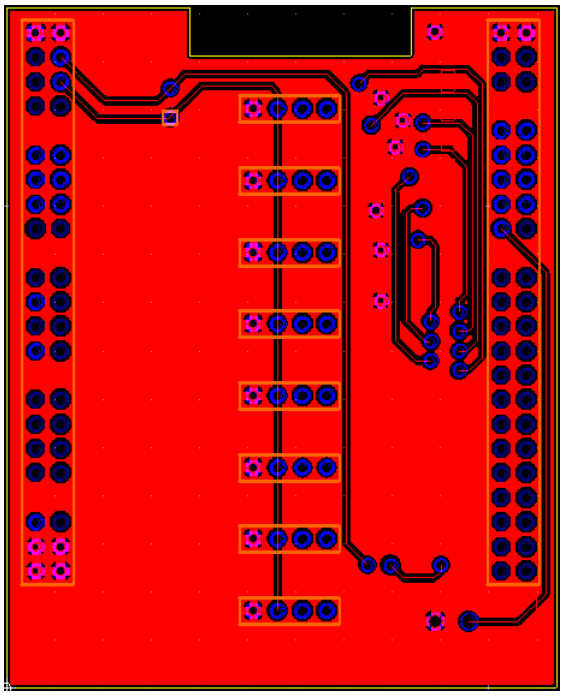


(b) Subcircuit schematics for the sensorsystem

Figure 8: Schematics



(a) Top Layer



(b) Bottom Layer

Figure 9: CAD model of the extension card

11 Emergency stop electronics-relay card-Emil

11.1 Description

This cards function is to connect all the different parts of the emergency stop system and if the signals are fine, relay the signals to the Husky control panel and ROARy PSU.

11.2 Requirements and Limitations

Output connectors:

Radio receiver
Power supply
BeagleBone Black
ROARy PSU killswitch
Channel 1 of the e-stop button
Channel 2 of the e-stop button
Husky control panel killswitch

Table 6: Connector list

11.3 Design and Interface

The relay card receives signals from the BBB and from the radio receiver, the signals are digital and they are 1 when everything is fine and 0 when something gets too close or someone presses

the remote stop button.

11.3.1 Radio receiver

Description:
Receiver and remote that can be used as a remote e-stop button.

Requirements:
A remote and receiver that could work at some distance and could be integrated as a part of the e-stop system.

Design and Interface:
A pack with both receiver and remote was ordered and in optimal conditions it is supposed to work at 2000m. It can be configured to work as a dead man switch, toggle and latch. The outputs/inputs are labeled as -, +, C, B and A. The relay is NC A and B and NO A and C. So A is the signal, B is GND and C is 5V

11.3.2 Relay card

Description:
Relay the signals from the PSU and Husky back to them when signals are received from the BBB and radio receiver

Requirements:
Signals is received from the BBB and relay card. The signals are 1 (3.3V and 5V) when the E-stop is disengaged, when one of the signals is 0 the E-stop shall be engaged. The E-stop is disengaged by creating a closed circuit from the PSU and Husky back to them.

Design and Interface:
The ROARy PSU and Husky E-stop works by sending a signal to a NC-button, which sends it back to the PSU and Husky unless the button is pressed. The emergency stop system works by taking this signal putting it through a relay, then through the button and back to the Husky and PSU, this way if the button is pressed or if the relay is open the signal won't go back to the PSU and Husky.

This is accomplished by having an AND gate that takes the signals from the BBB and radio receiver and uses the output to supply a relay. The AND gate is supplied by a DC-DC converter that is also used to send 5V to the relay on the Radio receiver, which gives a 5V signal. The card has a 12V-5V conversion because the Radio receiver needs 12V.

Notes:
The wrong footprint is used for the AND-gate.

11.4 Using the system

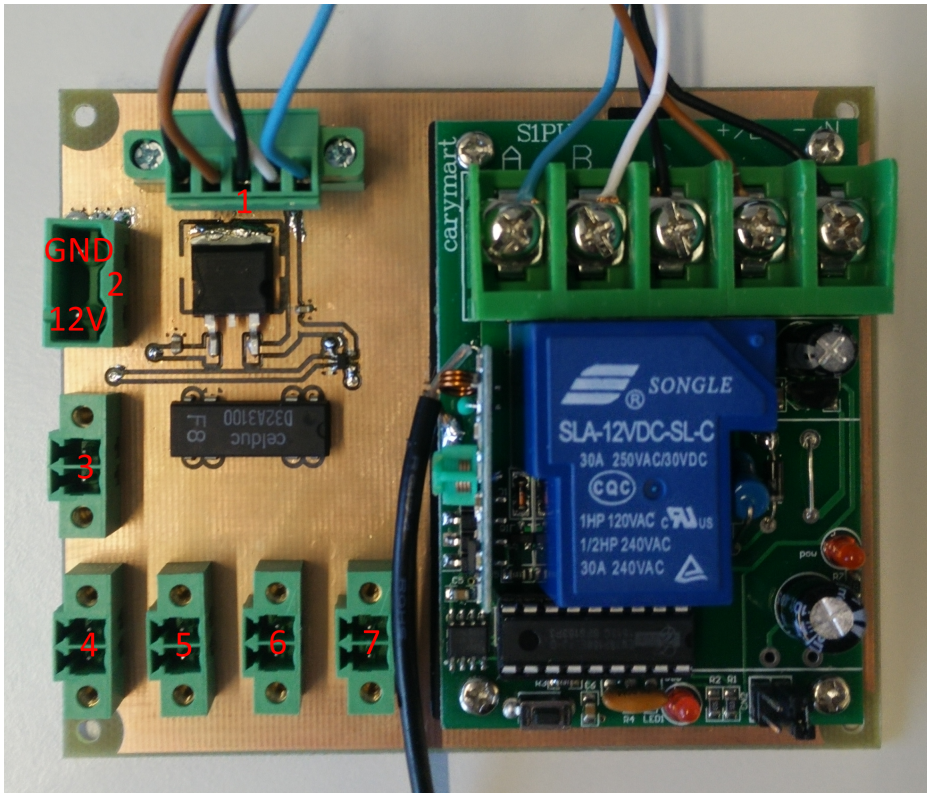


Figure 10: Power distribution sketch

1. Connect to the radio receiver as shown. The wires from left to right is GND, 12V, GND, 5V and signal. Pin 5 (signal/blue) is normally connected to pin 3 (GND/black) but when the remote button is pushed it connects to pin 4 (5V/white).
2. Connect to the PSU with GND and 12V as shown.
3. Connect to the BBB, top pin is signal which should be a 1 (3.3/5V) when everything is fine and 0 when an emergency occurs, bottom pin is GND.
4. Connect to the kill switch on the PSU we made.
5. Connect to one channel on the emergency stop button.
6. Connect to the other channel on the emergency stop button.
7. Connect to the PSU on the Husky, which was previously soldered to the emergency stop button.

Connector 4, 5 and 6, 7 is pairs, so they can be connected freely as long as each pair has one channel of the e-stop button. Pin order does not matter, since when the e-stop is disengaged it creates a closed circuit.

11.5 Troubleshooting

Check the DC-DC output and make sure it is 5V.

Check the inputs to the AND, they should be 3,3V from the BBB and 5V from the Radio receiver.

If there is an output from the AND-gate the relay should close, make sure it is closed.

12 Sensor System-Tobias Nabar

12.1 Description

The sensor system used in this project consist mainly of the LMS111 2D laser scanner. It can be used to find out where obstacles are within a 270 degree area infront of itself. The LMS111 has an operating range of 50 cm to 20m and has a 10% reflectivity on up to 18m. [1]

12.2 Requirements and limitations

In order to compile the program CMake is required, as well as some kind of ROS version as the code is also made to work on ROS. The versions used so in this project have been CMake 2.8.12.2 and ROS Indigo.

The LMS111 is needed for more than one thing. It needs to detect the environment to allow the system to create a map so that the ROARy can navigate through the area it finds itself in, but it also helps the camera. Since the camera will need a lot more processing power than the LMS111 for navigation, the LMS111 can also help with finding the refuse bin once the camera has identified it.

12.3 Design and interface

The code used for the LMS111 is designed to work on any lidar from the lms1xx series together with ROS. The data gathered by the LMS111 is published in the topic 'scan' as can be seen in figure 11.

12.4 Method

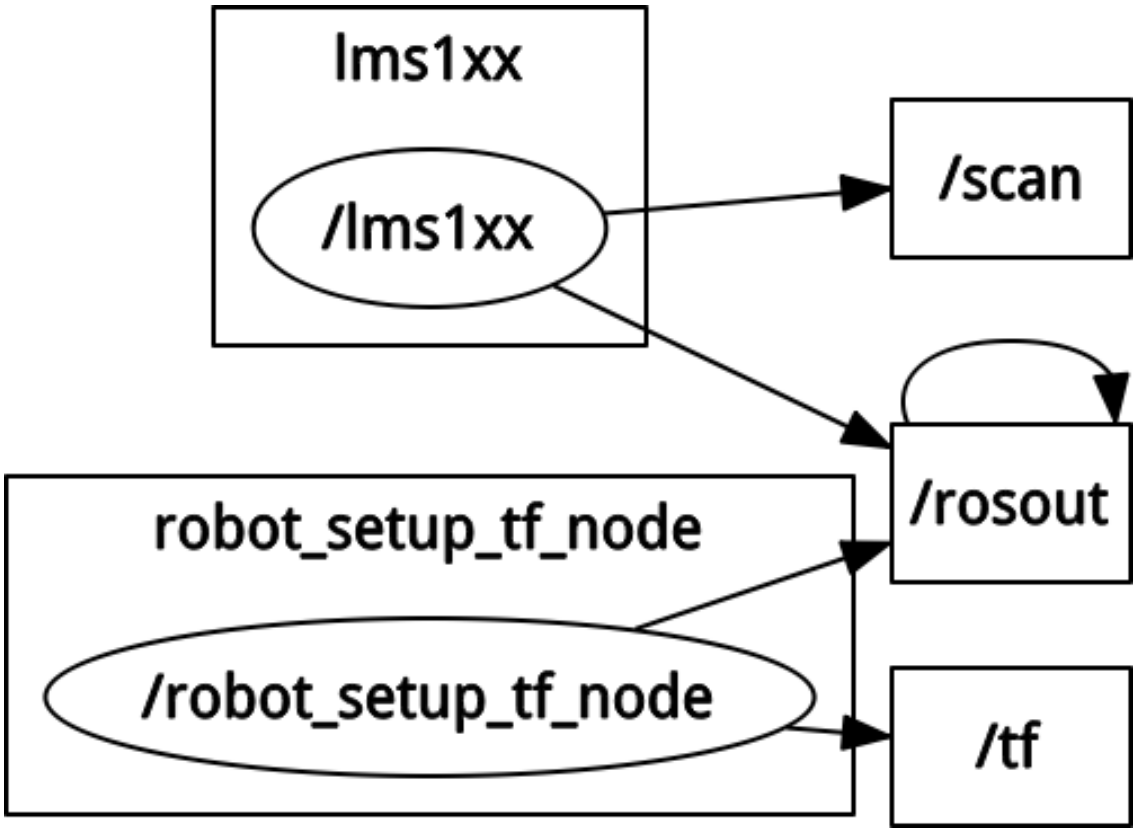
To get the LMS111 up and running, the first step was to find and download its drivers for ROS.[2] The drivers used in this project should work for all of the lidars in the LMS1XX series.

In order to make the LMS111's results have any meaning to the project as a whole, its position has to be represented in some kind of way. The way it is represented in this project is by using the tf package in ROS. The tf package allows a user to keep track of several coordinate frames that are all somehow connected in a rather simple way. To simplify how it works, one could say that the tf simply connects one item to another in a coordinate frame.[3] In this case it is the LMS111's position that is connected to the base of the Husky, then the tf will keep track of its position and angle wherever the Husky moves in the world coordinate system.

To connect the position from the tf package to what the actual sensor sees it has to be given the same frame ID as what the tf code uses. Once the position and the data had been connected it was fully possible to show a pointcloud of the detected objects together with the connected pieces of ROARy in rviz.

[1] https://www.sick.com/media/pdf/2/42/842/dataSheet_LMS111-10100_1041114_en.pdf
[2] https://github.com/RCPRG-ros-pkg/RCPRG_laser_drivers
[3] http://wiki.ros.org/tf

Figure 11: Shows what the lms1xx is connected to in the list of all the topics



The drivers only needed slight changes in order to work for the purpose of this project. An attempt to make the LMS111 only scan 180 degrees was made, however there was no success in this area. Since all information beyond the 180 degrees are unnecessary this problem had to be stepped around. Until a solution to make it only scan the 180 desired degrees, it is currently setting all unnecessary data to 0 values, which are not considered in the map. This means that it is still scanning all 270 degrees, but only 180 degrees are considered. Another slight change to the drivers is that the IP address was set to a more suitable value to make it easier to access.

A delay was added to the start of the LMS1XX code. This is required because the LMS111 starts slower than the code, so the code would otherwise try to run without the LMS111.

12.5 Test and simulation results

The sensor only needed the drivers in order to work. The testing around that was done by running the code and simulating the environment with RViz to make sure that it was still working, and it was moved around at the same time. The same simulation envinronment, RViz, was used to make sure that the 270 degrees were cut to 180 degrees in the correct fashion, and the results can be seen in figure 12.

Together with SLAM, the lidar was used to create a map of certain corridors in school. The results of these tests can be seen in figure 13 and figure 14. The maps were made by driving the

ROARy through the corridors then the data gathered from the lidar was used for the SLAM and this made it possible to make the maps.

Figure 12: Showing how the LMS111 views the world, picture taken from a simulation in RViz

12.6 Using the system (Dependencies, installation, configuration and execution)

In order to use the sensors for the ROARy, the code that has anything to do with the LMS111 directly has to be delayed on the ROARy startup. This delay is because the LMS111 is slower to start than the rest of the system, and because of this the code is delayed until the LMS111 is running.

The LMS111 requires a supply voltage DC of 11 to 30 V with a minimum power output at 20 W. When connecting the LMS111 to a power supply, the brown cable is the power cable, the blue or yellow cable is the ground. If the heating is used then the white cable is the power for the heating unit, and the black or green cable is the ground for the heating unit. In order to compile the code, the command catkin_make can be used as it will compile everything in ROS. If the other things should not be compiled, then the command is:
$ catkin_make -DCATKIN_WHITELIST_PACKAGES="liblms1xx;lms1xx"

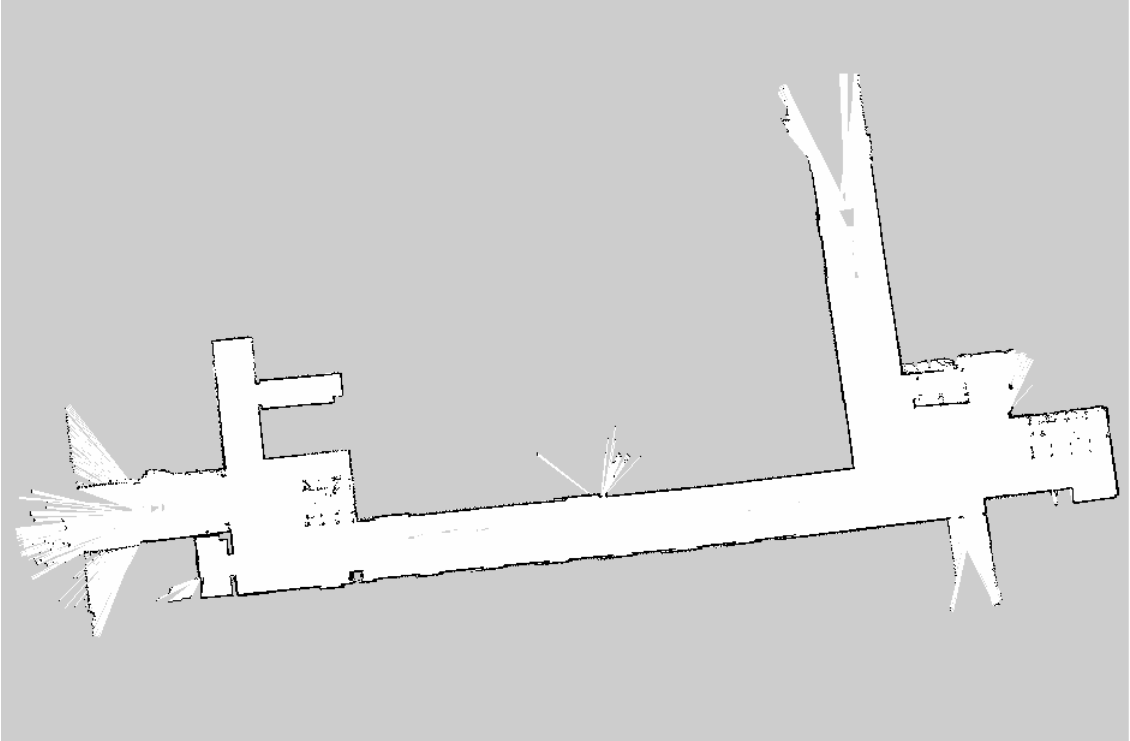
The drivers have been changed slightly in order to fit the needs of the ROARy project. The scanning radius is set to 180 degrees instead of 270 and the angle between each scan is the default value. At the start of the program a delay has been added in order to slow down its bootup time by 10 seconds because of how slow the actual lidar starts. The position of the lidar upon the ROARy has been measured out and is published in a tf.

The drivers for the sensor system will automatically start when the ROARy is powered on. However, if these do not start correctly for some reason or they just need to be started manually, run this command:
$ rosrun lms1xx LMS1xx_node

12.7 Versions

Currently there is only one version of the files that have something to do with the LMS111. They can be found at:
/opt/mdh/projects/roary/catkin_ws/src/lms1xx

Figure 13: Shows a map of the corridor made with help from LMS111 and SLAM, picture taken from a simulation at ISS's corridor



/opt/mdh/projects/roary/catkin_ws/src/liblms1xx
/opt/mdh/projects/roary/catkin_ws/src/robot_setup_tf

12.8 Future work

Currently the LMS111 works as intended, however something one must remember when customizing the code throughout the ROARy project is the bootup time added to the LMS111 code. The more that gets added to it, the slower it seems like the LMS111 starts up, which means that the bootup time may need to be increased in the future.

13 Vision System-Dennis Anders Nabar

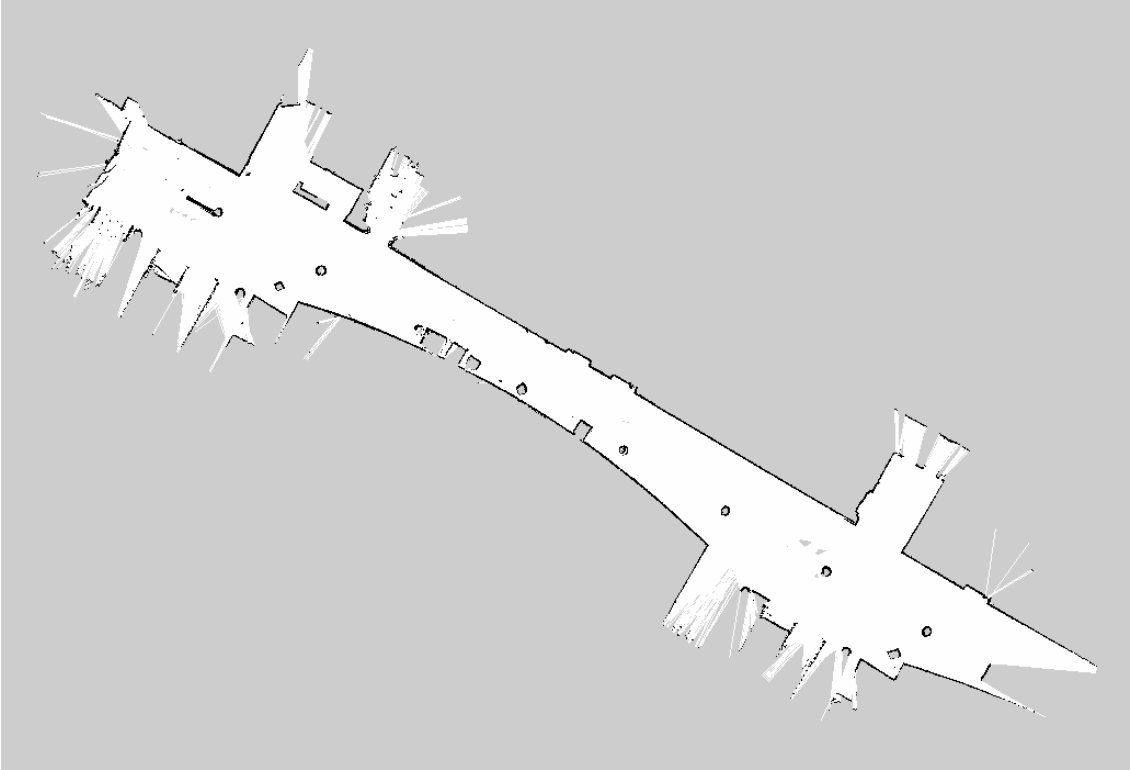
Missing text from Anders

13.1 The camera driver for ROS-Nabar

13.1.1 Description

There are two HD cameras, Logitech C270 HD, used in the vision system of the ROARy. One of them is used to find out the bin and to shoot an image of it. The other one is used for the manual driving by driver performing video streaming operation.

Figure 14: Shows a map of the corridor made with help from LMS111 and SLAM, picture taken from a simulation at Robotics Lab's corridor



13.1.2 Requirements and limitations

In order to compile the program, CMake and ROS are required. The versions used in this project are called CMake 2.8.12.2 and ROS Indigo. Transferring an image or a video needs a high bandwidth network and processing resource. Therefore the minimal resolution of the camera is recommended to be used regardless of its HD functionality.

13.1.3 Design and interface

The driver package used for the cameras in the vision system needs to be able to work with any camera which supports the Linux 14.04. The image/video obtained by the cameras will be published under namespace 'ROARy/Camera' as seen in figure 15.

13.1.4 Method

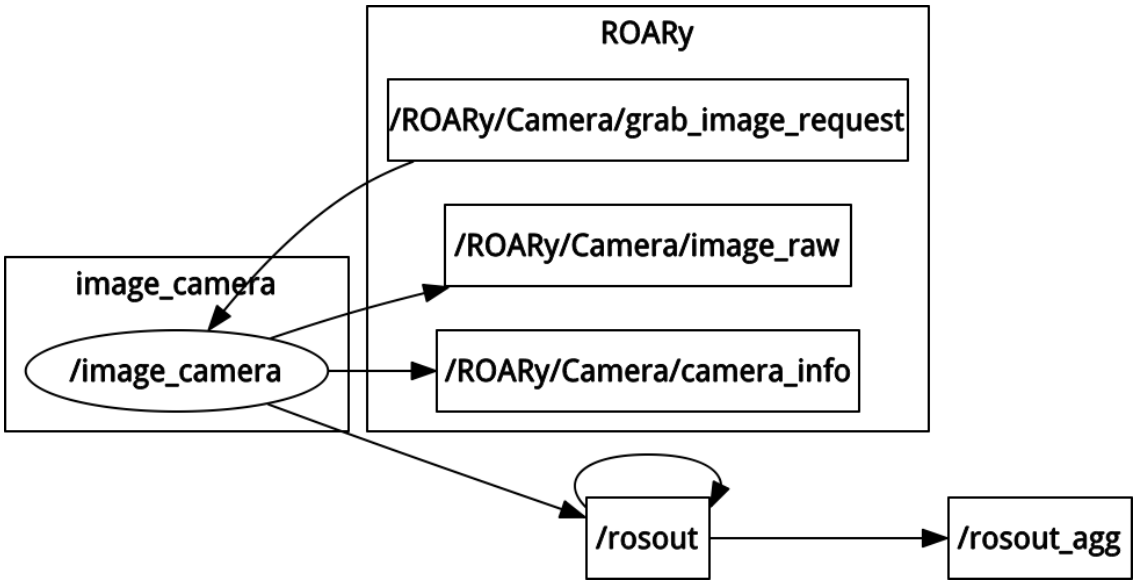
The driver for video streaming could be found from ROS repositories by using apt-get. In order to shoot an image, the driver for video streaming should be downloaded [4] and customized as roary_camera package which is designed to send an image only when it requested.

13.1.5 Test and simulation results

The streaming data from the video camera can be tested easily by using "image_view" package on ROS and the result can be seen in figure 16. The result of image test can be gathered/collected

[4]https://github.com/ktossell/camera_umd

Figure 15: Camera design



easily by performing the command, "rosrun echo ROARyCameraimage_raw", on terminal.

13.1.6 Using the system (Dependencies, installation, configuration and execution)

Dependencies: The cameras for both taking an image and streaming a video, should have the a+x permission which is set automatically by custom shell script.

Installation: The package for the video streaming is installed by "sudo apt-get install ros-indigo-uvc-camera".

Configuration/Execution: The camera's resolution can be customized by changing the variables of width and height in the code. The drivers for the cameras will automatically start when the ROARy is powered on. However, it is possible to run it manually for testing. Run the following commands:
For streaming:
$ rosrun uvc_camera uvc_camera_node
For shooting a image:
$ rosrun roary_camera grab_image_request

13.1.7 Versions

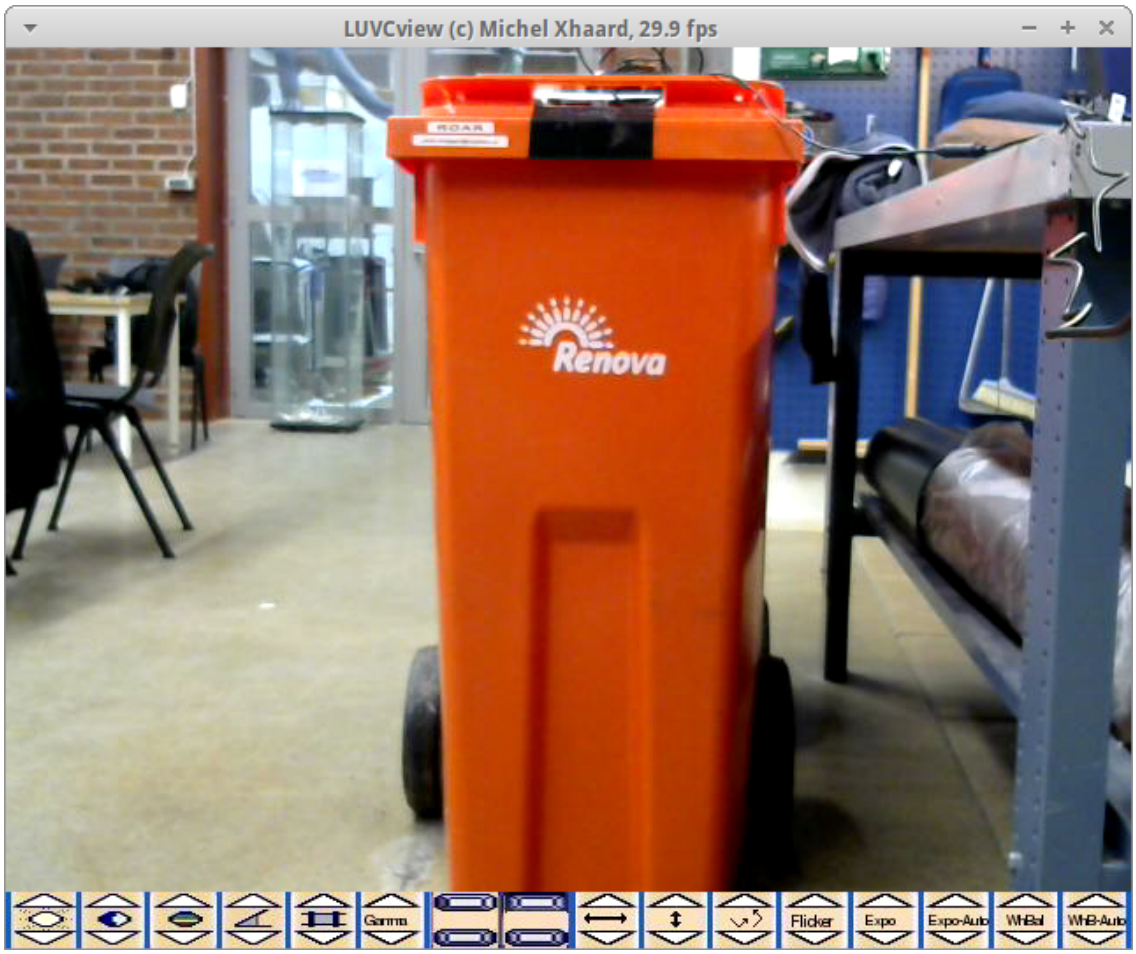
The source code for the taking an image can be found at:
/opt/mdh/projects/roary/catkin_ws/src/roary_camera

13.2 Manual vision control

13.2.1 Description

During the moments the ROARy are driven in manual mode a vision system is needed so the driver can compete its objective without any accidents. For it to be accomplished a controlled

Figure 16: Video streaming



camera are needed to increase the field of view of the driver.

13.2.2 Requirements and limitations

The limitation of the system is that the camera are only able to rotate 30°to the left and to the right from the starting point. the view is limited when the garbage bin is on the ROARy making it hard to see the pins on the garbage truck when

13.2.3 Design and interface

The node reads data that the joystick sends over the network. The value received are then transformed into a velocity for the servo controlling the angle of the camera. the speed can be adjusted by changing a K value in the code.

13.2.4 Method

The servo receive a PWM signal of 300ms and by having active range from 0.5ms to 3ms. Transforming the 0.5ms to 3ms range into a degree based range, so the servo is moved by degrees. using

the formula for degrees and taking the Δt and a K value the servo is now moving in a velocity.t being the difference between two iterations.

13.2.5 Using the system

to use this system Adafruid library and ROS where used. be able to use the pin on the beagle bone the node have to be started in sudo. The node is put into the sudo version of crontab to run a script that activates the node in sudo at startup.

13.2.6 Future work

Making it a pan-tilt instead of only a rotor , this can be simply done code wise. Better servos that can rotate 360°for full view. Getting feedback from the servos to have better orientation of the camera.

14 Loader System-Mechanics-Ragnar Oscar

14.1 Description

A lift mechanism to be mounted on the base platform (the husky), capable of loading and unloading the cargo on itself. The main principle is linear motors and linear rails, in an solution similar to that of a forklift.

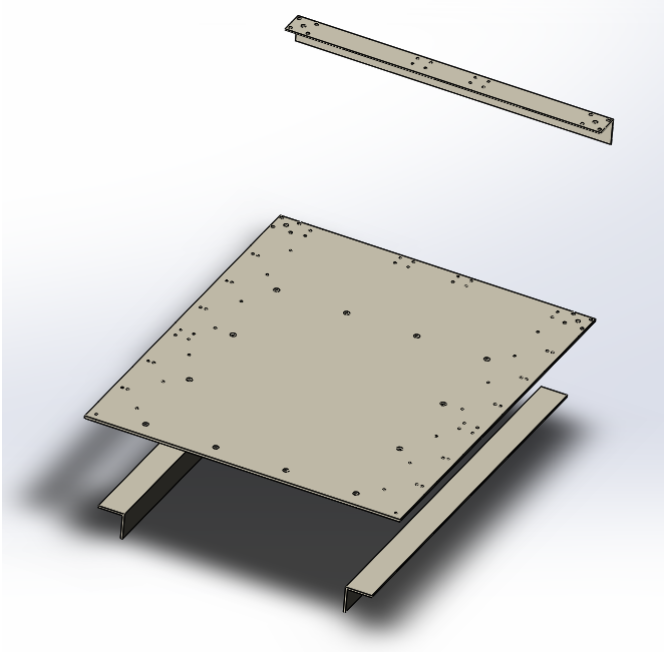
14.2 Requirements and limitations

The loader mechanism needs to be stable enough for the load and center of mass to stay inside ROARys footprint during all stages of loading, unloading and transportation. It also needs to be sturdy enough to handle heavy loads, since a fully burdened trash bin can weigh as much as 100 kg. The mechanism also needs to be designed to not extend too far outside of the base in the horizontal plane (in its collapsed state), since that would decrease ROARys ability to navigate narrow passages, and make it unsafe to operate around humans.
Another requirement of sort is that the robot as a whole must look its part. Both due to marketing and building trust within the industry, but also because the final product will be expected to operate “out on the streets”. The robot’s appearance is part of its function, since it will influence how people interact with it. Aesthetics is therefore important to keep in mind, although it is not the priority during this initial development.
As for limitations, there is a number of things that is (in the duration of this project at least) unchangeable. The base platform provides good mounting possibilities, but those are limited to the top of the robot, leaving a lack of mounting points on the rest of the platform. The bin is also defined as is. This limits the possible ways in which it can be repeatedly lifted safely.
The weight and load capability of the husky is also a limitation of sorts, as they are somewhat low for the task. The total weight of the husky is 50kg, and the defined maximum load is 70kg. This makes a limitation on the weight of the mechanism, since both cargo and lifter needs to clear the weight limit of 70kg. It also puts a higher demand on balancing, since it could be necessary for ROARy to lift cargo heavier than itself.

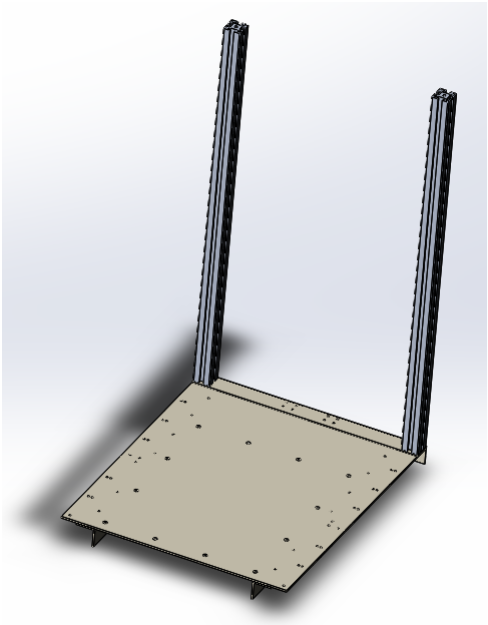
14.3 Method

The overall design choice is to put emphasis on simple solutions and as little custom made parts and materials as possible.
The base platform is a Husky from clearpath robotics. The gripper arm is 2-DOF and is based on

T-slot extrusions and driven by lead-screw actuators. Vertical movement is enabled by slide rails and robalon-blocks. Horizontal movement is enabled by telescopic rails.
The actuators are linear electric motors mounted directly in the frame. The strategy is once again to keep it simple. An solution such as this is less adjustable to the needs of the project, but offers more robustness. For example, the stroke of the vertical motor is longer than needed, and the stroke of the horizontal motor is somewhat shorter than what is optimal.
The connection between the base platform and the lift mechanism is a plate of aluminum, called the base plate. To save weight the plate is enforced with three L-profiles. The profiles is added to keep the base plate rigid, and is also made from aluminium to save weight. The profiles is bolted

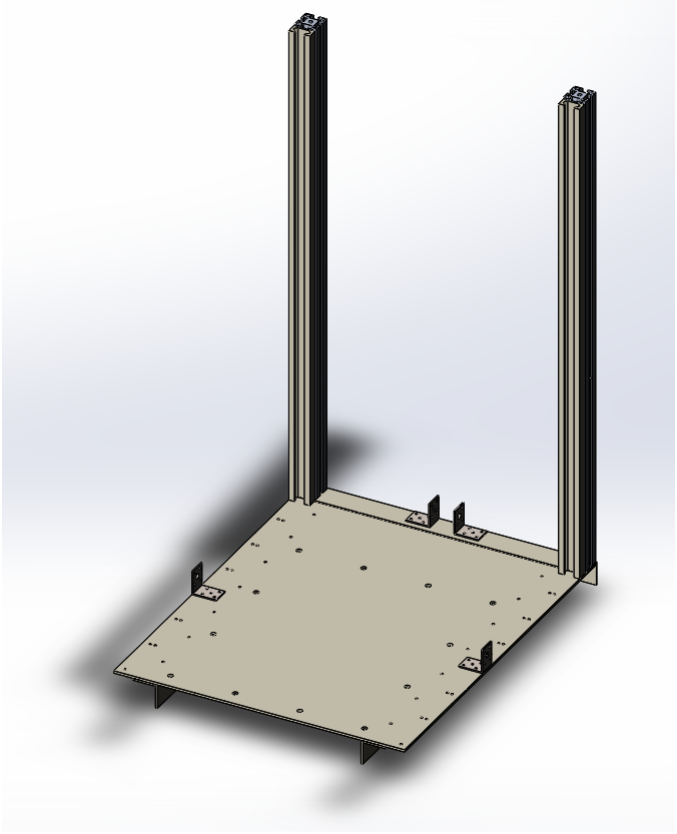


to the downside and edge of the plate.
The main frame is made with T-slot-profiles, which is connected using either angle brackets or with bolts in the ends of the profile. This makes the structure easy to assemble, and easy to

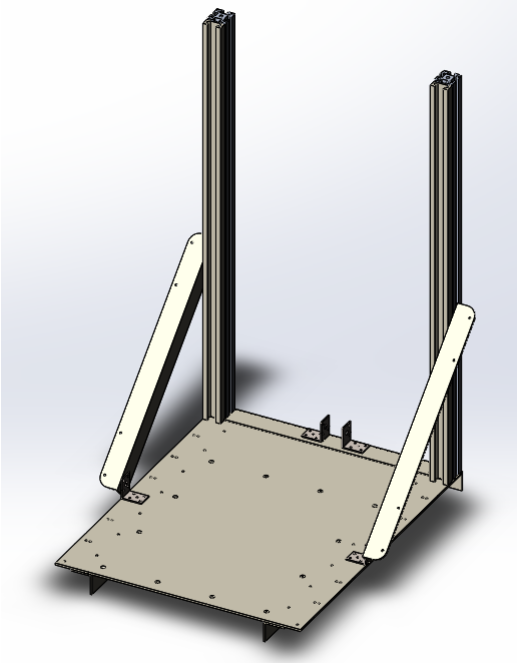


configure in future iterations.
The motion of the arms is guided by two pairs of rails. The horizontal motion is guided by heavy

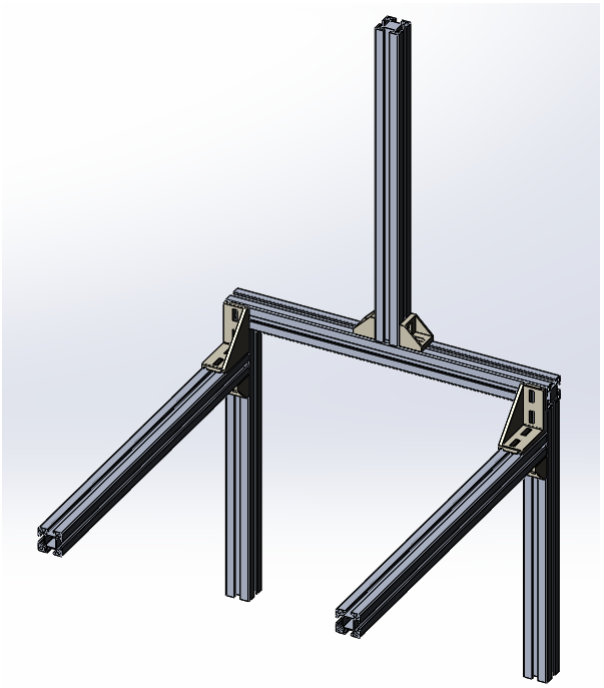
duty expandable rails. These are also a standard module solution from the industry, to provide robustness when dealing with heavy loads. The vertical rails are supported directly by the frame,

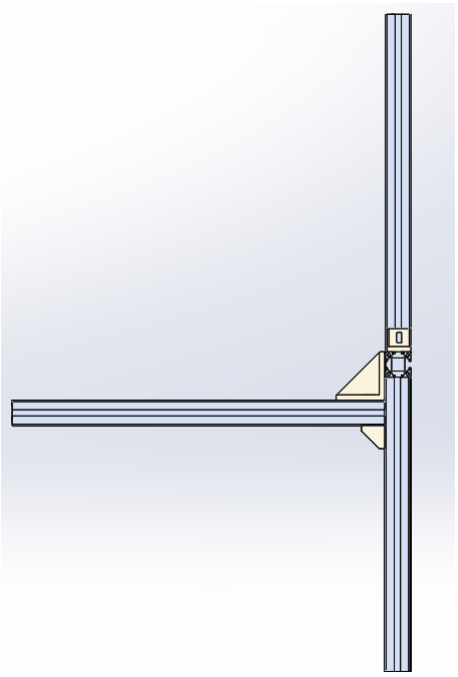


and are mounted at the corner posts
The support L-beams effects stability. One thing to keep in mind is that the t-nut slide slightly over time and needs to be readjusted, otherwise the entire structure can look a bit crocked.

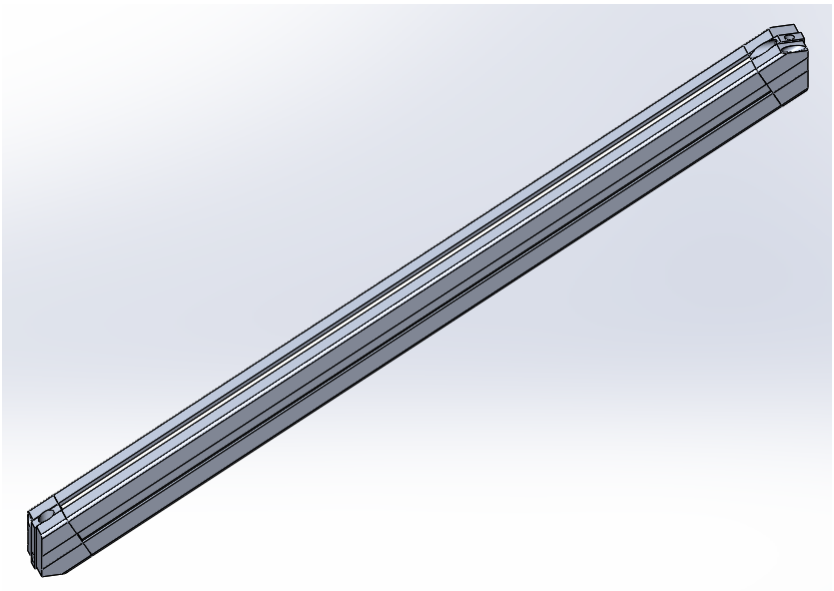


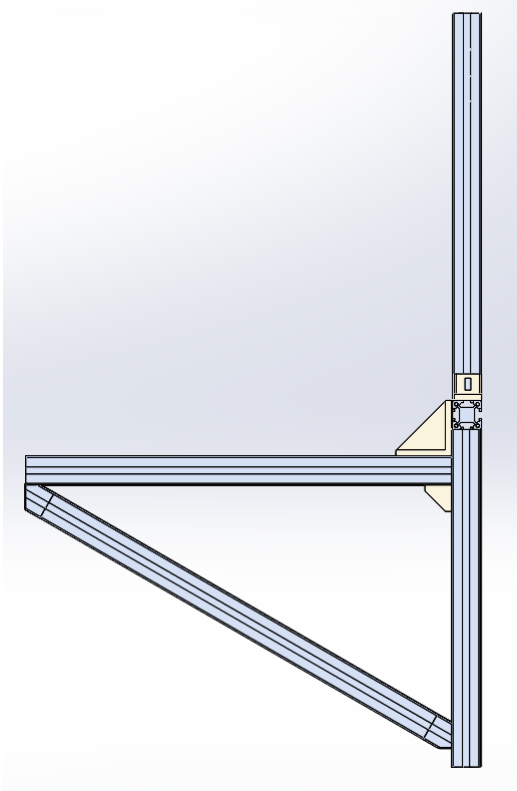
The moving part of the structure are made to hold up to 150kg. It is stabilized using a triangular

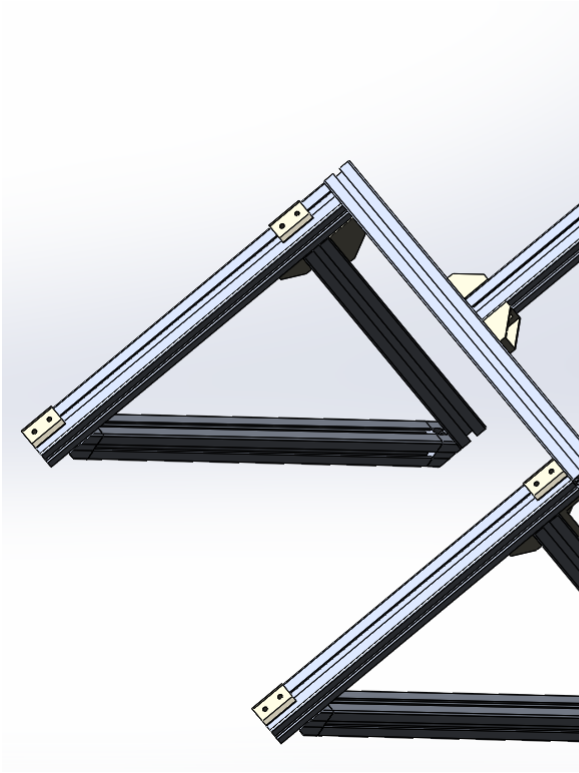




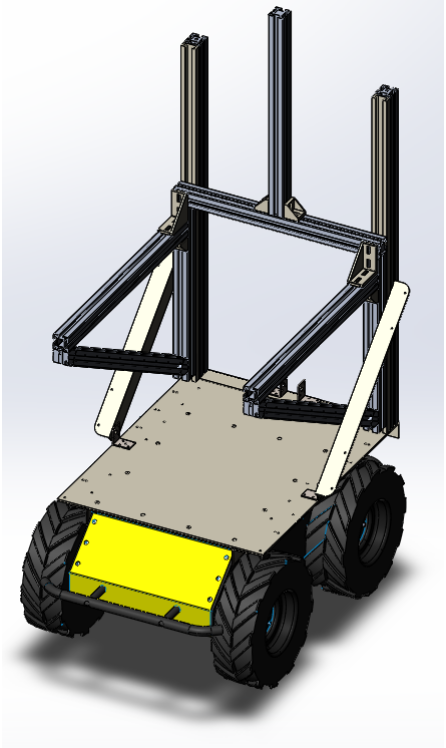
construction.

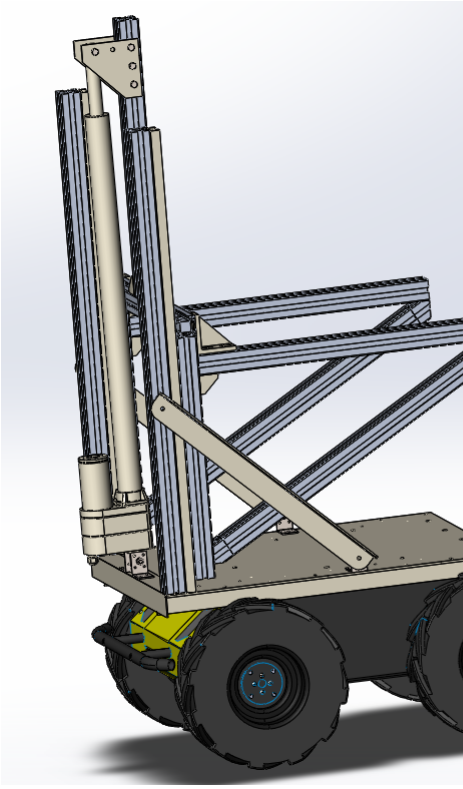




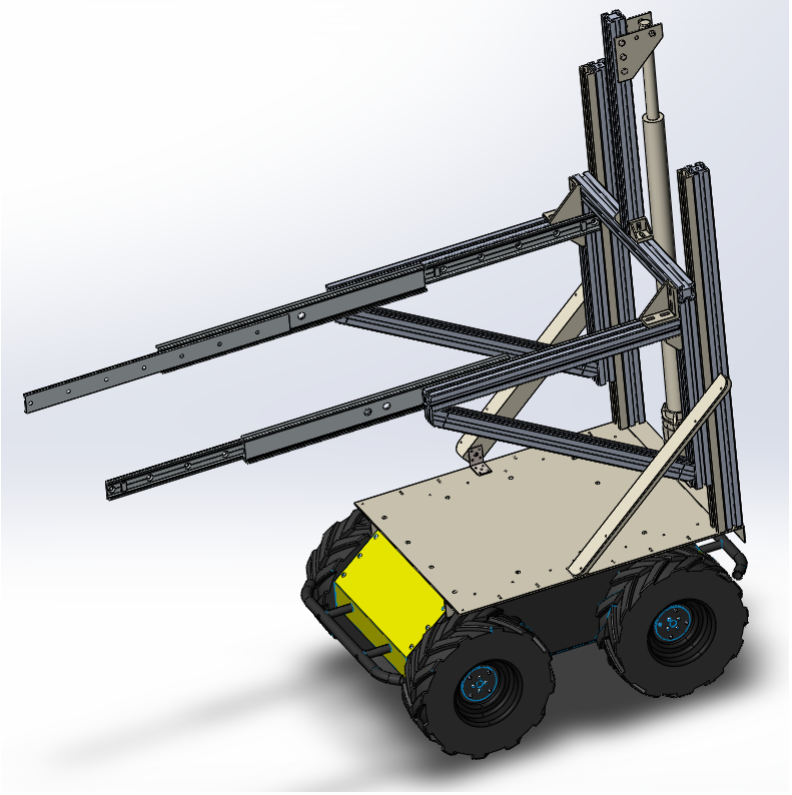


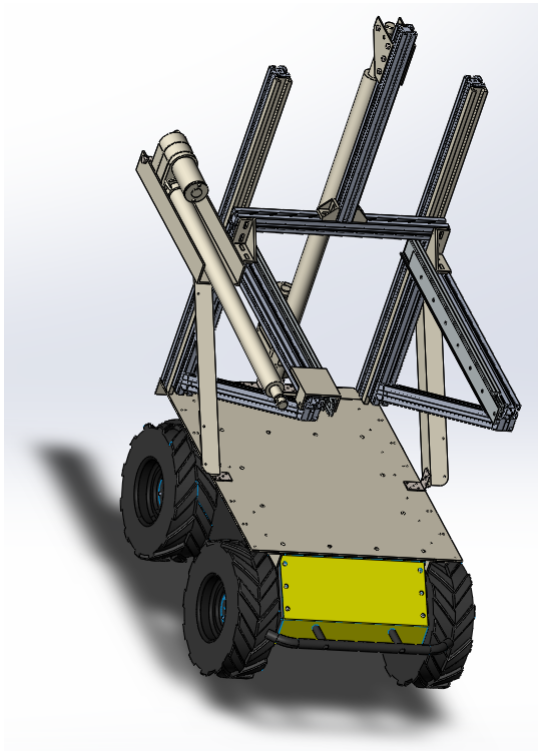
The blocks are made of a self-lubricating material called “Robalon”.





By moving the upper mount for the vertical actuator its offset can be changed.
The telescopic rails are sturdy enough for several hundred kilos and are graded for outdoor use.





Horizontal motor is mounted using U-shaped profiles.

14.3.1 Compromises

The horizontal actuator is not really optimal in its current state. The force is applied to one side, and then distributed to the other by the frame. The motor also rests outside of the mechanism, to one side. Given more time, a more integrated solution that applies equal force to both arms should be found.There were a couple of alternatives to the existing solution that vere disregarded for the sake of either simplicity and availability. One possibility is to use a so called rigid chain actuator. Another is to use a ball screw in each arm, and to connect the screws in each arm to a central rotary motor.
The materials i another compromise. Aluminum were favored over steel for the sake of weight. But, given a different platform with better load capacity, a steel frame would offer more a more robust, and possibly cheaper solution.

14.4 Test and simulation results

During simulation the main issue has been with displacement in the base plate. When two kilonewtons was applied to be edge of the baseplate it would first move several centimeters but after adding L-profiles that number is now 0.006mm where it moves the most.
As for testing, the lift has successfully lifted the intended bin (with light load) a number of times. When being manually operated ROARy have a tendency to slightly tilt so that it stands on its back wheels if not carefully operated. This is due to that center of gravity is farther back than originally. One unforeseen problem is that the upper mount for the vertical motor applies unwanted torque on the structure.

14.5 Assembly and manufacturing

Regarding assembly: Although the goal is to keep custom made parts to a minimum, at certain points it is more convenient to make the parts, than to find and buy a part that fits the needs.

Especially since some manufacturing processes are available on site. Parts made specifically for this project is made in one of two ways. Plastic parts is printed in an 3D-printer. Metal parts are made from sheet metal or standard profiles, and cut, drilled and bent into different shapes. Following are the parts custom made for ROARy
Baseplate is made from a 5mm thick sheet of 5754-H22 aluminum. The base plate is drilled out in a cnc router, to make mounting holes where they are needed. Some of the holes, namely the ones for the bolts connecting the base plate to the husky, are countersunk to level the bolt heads with the surface of the plate.
The profiles enforcing the base plate are made from 50X50X3mm L-shaped aluminum profiles. The profiles are sawn to length and bolt holes are drilled at repeated intervals using a drill press. They are then bolted to the base plate.
The main frame is assembled from T-slot profiles connected by corresponding angle brackets.
The motors are mounted to the mechanism using brackets and threaded rods. The vertical motor in the center of the main frame, and the horizontal on the side of the frame.
The grippers are made from 1,5mm steel sheets. They are cut out using a pair of electrical tin shears. The upper edge of the gripping plate are reinforced by folding the edge back on itself about 5mm. Holes are drilled, and the two parts are connected by a hinge, riveted to the lower edge of the plates.

14.6 Versions

The gripper comes in two different versions. The first is a simpler one, made from a single sheet of steel. It is simply an edge to grip the bin.
The second is made from two sheets of steel, connected by a hinge and a number of springs. This allows the gripper to compensate for the inaccuracies of the robot to some degree. The plate in contact with the bin is also slotted, to make sure that the bin locks in place at certain points.

14.7 Future work

The loading mechanism is to be fitted on a 2-wheel platform.
The gripper is limited in forward direction by its lead-screw actuator as it cannot extend more than its original length. On wikipedia we found another kind of actuator called rigid chain actuator that seems worth further investigation.
Adding balance support during loading/unloading. Extendable support legs or moving counterweight. There are benefits and drawbacks here, extendable legs makes a greater impact as the center of rotation is moved, a movable counterweight has the benefit of being useful during transportation for adjusting center of gravity which is in line of the aim to have a two-wheel platform.
Horizontal movement needs a longer stroke. This could either be done either with a new actuator or a solution yet to be discovered. If a change in actuator would take place then it is a good idea to make it well integrated and not putting excessive weight on one side like now. With current design it is quite a challenge from a control perspective to drive up to the garbage bin. It is easy to get correct angle but translation error is more difficult, hence it would be beneficial to widen this tolerance. As of now the gripper is spring loaded in order to offer a firmer grip and because of this a wider structure is a possible solution as any excess space could be filled out by the springs.
The bolts and screws tends to loosen up over time due to vibrations. A new structure where everything is welded together should be taken under consideration.

15 Loader System-Electronics and Software

15.1 Requirements-Suan

The loader system should be able to lift the refuse bin with a maximum specified weight of 70kg. The electronics should be designed to cope with power supply requirements along with over current protection for the motors used. The software must be able to quickly translate the commands from the main process to commands understandable by the motor controller. The motors should operate smooth to extent that there are no jerks while handling the refuse bin. The software needs to understand two different types of commands from the main process, which are, speed control commands and position control commands. These types of commands will be used in different situations. The software should know the limits of the mechanical system and always take care the structure is not damaged by not moving out of bounds.

15.2 Description-Suan

The software here is written for the motor controllers which actuates both the lift and grab motors. The lift and grab arms of the ROARy are designed to handle the refuse bin as explained in the Mechanics section. The software must be responsible enough to handle this mechanical structure smoothly along with the refuse bin. Three designs were made in this regard.
The first design failed while developing the software. The RoboClaw motor controller malfunctioned and the board was burnt partially due to an overloaded motor. The overload was caused because the motors were mechanically stopped at a non-end point by the ROARy's lift structure. The cause for the failure to stop the motor and protect itself from the burn could be either with a bad over-current protection circuit in RoboClaw or with a bad configuration settings in the control loop registers. The second design failed to work well as BBB's adc(analog to digital conversion) drivers stopped working at random instances and hence providing a very unstable system.
The third design involving an arduino is the current working solution. All the solutions are explained below.

15.3 Desired solution: Using RoboClaw Controller-Suan

This section describes the design with Beagle Bone Black and the RoboClaw motor controller. This motor controller has integrated position and speed controls for the arm of the motor. The required acceleration and maximum speed controls can be set as register parameters for every movement it needs to do. The RoboClaw controller has arm position data via a potentiometer interface from the motors via a 3 wire analog interface.

15.4 Design and interface using RoboClaw-Suan

The designed system consists of a BBB connected to the motor controller (a RoboClaw 2x30A Motor Controller) via USB cables. Figure 17 shows the schematic. The driver software is written as a ROS(Robot Operating System) node. The ROS node communicates with the ROARy-pc via the ethernet port connected to a switch. The motor controller is connected to both the lift and grab motors with polarities as shown in the table 7. The software is not completed for this implementation design, as it was abandoned during its development.

Table 7: Lift and grab motor wiring

Grab motor (ch2)	red(2) -> A	black(1) -> B
Lift motor (ch1)	red(2) -> A	black(1) -> B

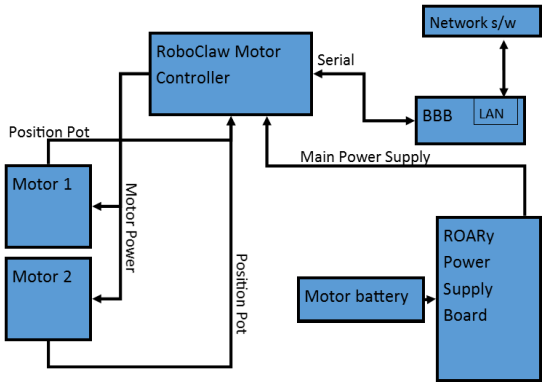


Figure 17: Design using RoboClaw

The ROS topic names used are tabulated. The 'Input' topic receives two types of commands. The command type is Vector 3 belonging to the geometry_msgs. The command values in X and Z are interpreted to be speed control commands when Y value is 0. The same is interpreted to be position control commands when y is greater then 0.
The publish topic types are also Vector3 belonging to the geometry_msgs. Here Z field is the position of the arm. The position values range from 0 to 509 and is converted suitably to motor-controller understandable values from its manual. The speed control values ranged from 0 to 127 for motor A and 127 to 255 for motor B. The psoition values were obtained digitally from the RoboClaw motor-controller interface.

Table 8: ROS topics

Topic	**Description**
ROARy/Husky/Lift/Input	Receiving command
ROARy/Husky/Lift/Output/GripperMotor	Publish Pos.
ROARy/Husky/Lift/Output/LiftMotor	Publish Pos.

This solution was not finally implemented, as the RoboClaw controller burned as explained in the Description Section. We continued and implemented using an alternative motor controller (The SaberTooth) which happened to be available at that time.

15.5 Desired solution: Using SaberTooth Controller-Suan

This implementation was same as that using the RoboClaw controller except for a few changes. The controller that was used is "Sabertooth 2x25 V2". This controller doesn't have the advanced motor controlling features. The sabertooth featured a simple speed control command interface, wherein it accepted the same range of values from 0 to 255. i.e. 0 to 127 for motor A and 128 to 255 for motor B. As this controller didn't accept position feedback from motors, there is no features like acceleration and maximum speed control, the ROS node had to include features for this purpose. Therefore the position feedback from the motor was directly fed to BBB's ADC port. Figure 18 show the connections made. The ROS topics remain same but included velocity values. Table 9 shows the ROS topic which includes the velocity values.

Every position command value is further fine tuned to obtain required smoothness for handling the refuse bin. This was done with the help of a control code which is also used in the next implementation.

The implementation using Sabertooth motor-controller was completed. The software works satisfactorily until the ADC fails. The observation that the ADC failed at random times made it

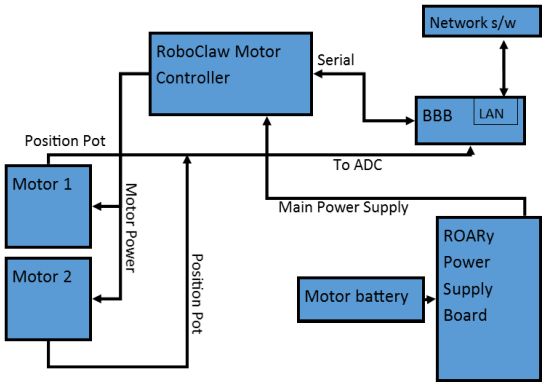


Figure 18: Design using SaberTooth

Table 9: ROS topics

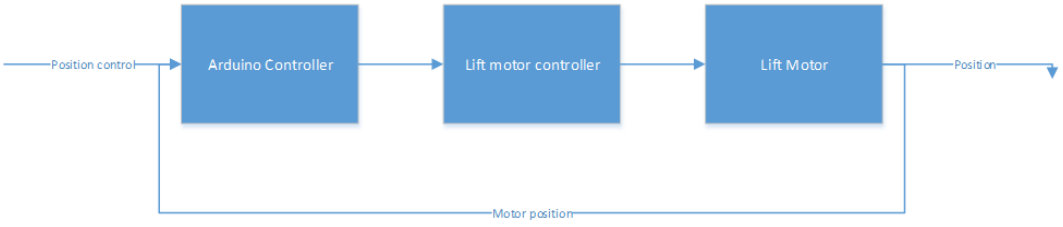
Topic	Description
ROARy/Husky/Lift/Input	Receiving command
ROARy/Husky/Lift/Output/GripperMotor	Publish Pos. & vel.
ROARy/Husky/Lift/Output/LiftMotor	Publish Pos. & vel.

difficult to work with. The ADC was tested extensively before this implementation was dropped. The straight forward solution would be to fix the ADC drivers of the BBB OS. A work around too couldn't be done for the lack of time.
The same software is used in the next design implementation with changes suitable to the hardware.

15.6 Current solution-Daniel

15.6.1 Design and interface

The current design uses an Arduino Uno to control the velocity or position of the loader motors. the Arduino is running ArduinoSerial to provide ROS interface to the Arduino. The control of position is done using a PID controller with analog voltage as feedback and UART for interfacing the "Sabertooth" which control the power to the motor. The lift can be controlled using the topics /ROARy/Lift/Input with the data type geometry message/Vector3. The actual position is published to the topics /ROARy/Lift/Output/GripperMotor and /ROARy/Lift/Output/LiftMotor.



15.6.2 method

15.6.3 Test and simulation results

The PID was tuned to provide feasible performance in terms of overshoot and settling time. The current solution fails to settle properly which can be heared when listening to the motor when the desired position is close to the reference position.

15.6.4 Using the system (Dependencies, Installation, configuration and execution)

The system is set to startup with a launch file. If the system doesn't start, please use the command "rosrun rosserial_python serial_node.py _port:=/dev/ttyACM1 _baud:=115200". Have in mind the port might differ from ttyAMC1, please check which port the Arduino connects to. The system is set to do velocity control when using vector3.y =0 and to position control when using value v such that $1 \geq v > 0$ Component vector3.x and vector3.z is set to desired to desired velocity or target position depending on the value v.

16 Software

16.1 Overview software design-Mattias

The overall software is designed for the Robot Operating System (ROS). The benefits of this is through the great infrastructure of ROS it is easy to work with subsystems in parallel. A (ROS)-node is a process that communicate with other nodes through streamed data on topics. Topics are data buses with specified names over which nodes can exchange messages through publishing and subscribing data. A robot system usually consists of several nodes and topics. Nodes that are dependent on input from other nodes can be simulated individually by creating dummy nodes that generate input data, thus making it easier to develop several dependent subsystems simultaneously. Most nodes for ROARy are written in C++ towards the ROS C++ API with some exceptions. These exceptions are Simulink models that were never built for ROS target, however for future work these are meant to be running in ROS as well.

The ROARy system uses a namespace standard for nodes and topics. Figure **??** describes the namespace standard. Table **??** describes the definition of the different namespaces. The software system was designed with robustness and modularity in focus. With ROS these attributes comes naturally through messages with standardized data types. Simulink was used for some nodes in order to achieve easy modularity through the Simulink Stateflow toolbox. An example of a node that uses this toolbox is the control program which will be further described in the next section.

Using the Clearpath Husky simplifies modular development. Clearpath provides both simulation and

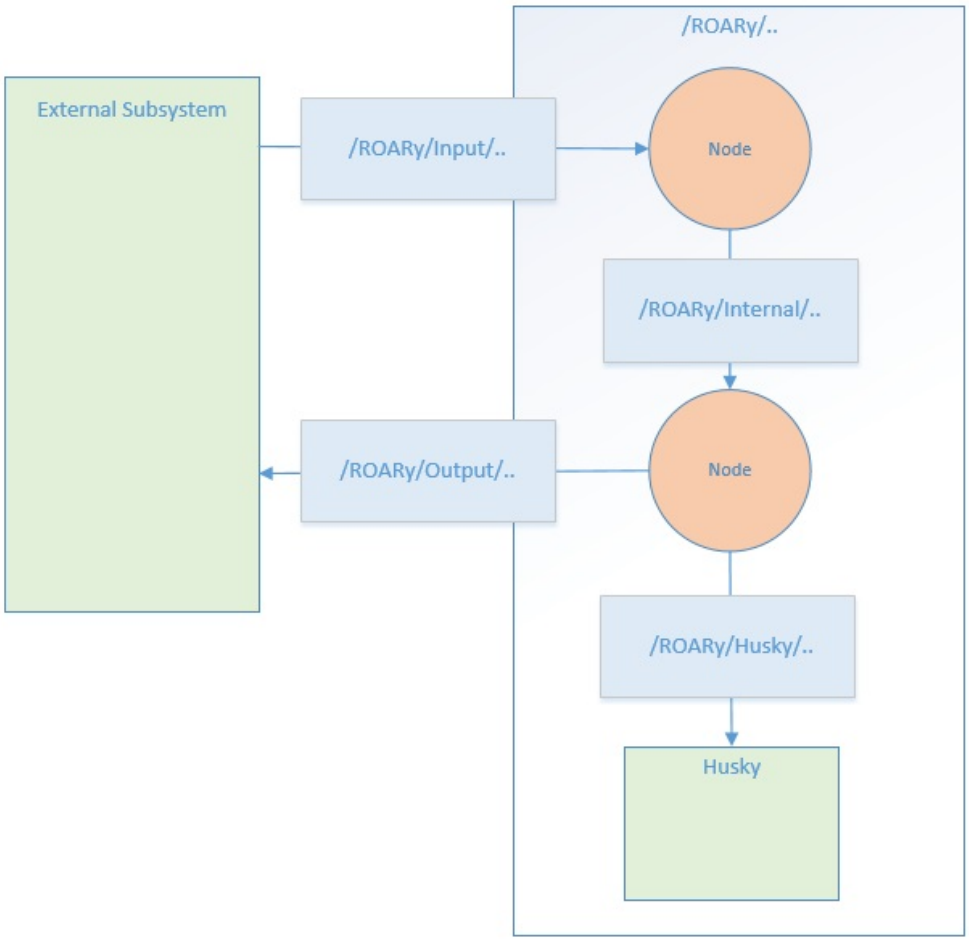
visualization packages for development with their robot platform. The Husky simulation package

starts a node that subscribes and publishes to all topics the actual, physical robot does. Because of

this everyone can test their nodes against the simulated Husky and when feasible simulated results

are reached the simulated Husky node is replaced with the real one. No change is needed for the

node that was running together with the simulation.

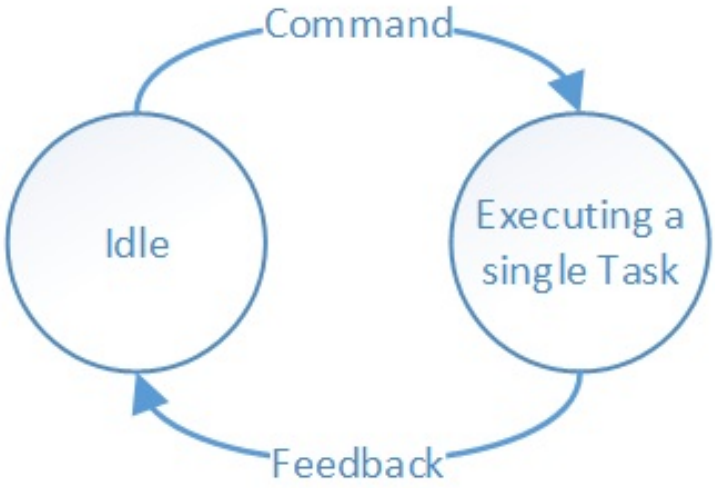


16.2 Interface-Mattias

The software design of ROARy is intended to work as a semi-autonomous slave with a set of functions called by a master (the task manager based on the truck). The master is responsible for the sequence of operations through these function calls, thus the decision on how to complete a mission is made on a higher level than ROARy itself. All functionality is called through published string commands to a specified ROS-topic and ROARy executes the corresponding functionality (task). There is a specific command that stop the current running tasks as well any tasks that may be running in the background. The ROARy will always return what state it's currently in, thus communicating to the whole system what task it is completing. Upon task completion ROARy will give feedback describing how the task execution went. All input and output are made through ROS publishes (with the exception of some output being a ROS service). Some tasks will call other tasks functionality within their own function call. After receiving a command the corresponding task is executed and once feedback is sent ROARy goes back to idle and waits for a new command. This behavior is repeated endlessly, which mean that never will two tasks execute simultaneously. Figure **??** gives a simplified view of the ROARy software workflow.

Table **??** presents all the tasks together with a very brief description of their functionality and which command to start them. Some tasks require additionally input to run. While the command will still start these tasks, the tasks will immediately return a feedback that not enough information has been given in order to fulfill the desired assignment. It's not ROARy's job to request the missing input data, but the higher level task manager's. Figure **??** shows all the ROARy software

Namespace	Description
/ROARy/..	The main namespace frame. All task nodes are found under this namespace.
/ROARy/Input/..	The namespace for all input from external subsystems.
/ROARy/Output/..	The namespace for all input to external subsystems.
/ROARy/Internal/..	The namespace for all communication between internal nodes and subsystems.
/ROARy/Husky/..	The namespace for all communication that leave the ROARy system, but stay on the husky. I.e. motor input.



interface inputs and outputs together with their data types and topics.

When a task leaves it's running state and returns to idle state it will output a feedback describing what made it transition back to the idle state. This can either be a successfully executed assignment or that something went wrong when trying to complete said assignment. Table **??** presents a list of all the tasks different feedbacks together with a brief description of each one.

Some tasks consists of a single node, while others may consist of several nodes. Some tasks may even use the same nodes as other tasks if they both share similar functionality. An example is the pickup task which will use the grab task in order to lift the refusal bin. In order to easier interfacing with the different tasks there is a control program that handles all the input commands. The control program is also responsible for outputting feedback and current state. Figure **??** show the interfacing between the truck and ROARy through the control program.

Table **??** show all possible ROARy states together with a brief description.

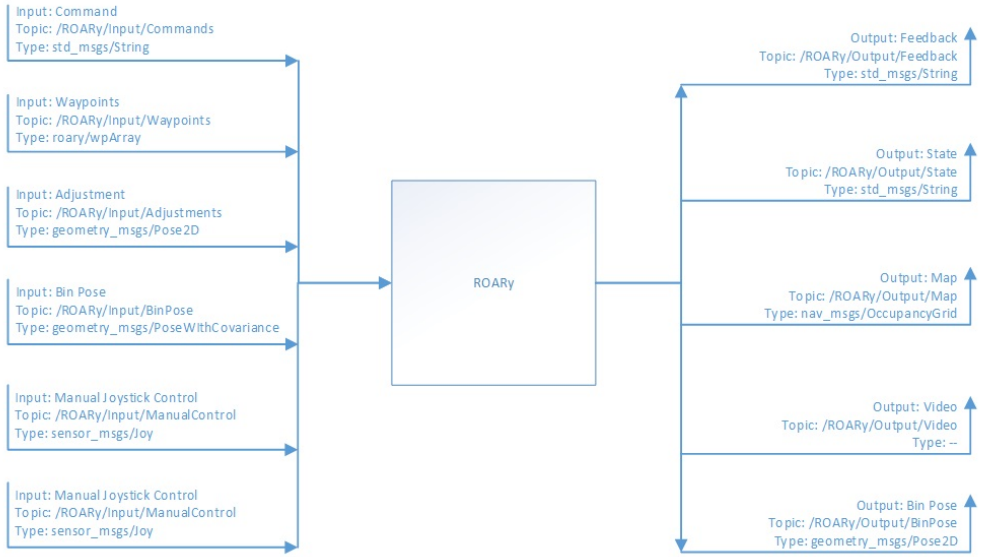
Apart from the output state topic it is possible to request ROARy's current state through the service GetState.

16.3 Control program-Mattias

16.3.1 Description

The control program is the central node of the ROARy software system. The main functionality of the control program is to read input from both external and internal subsystems and provide correct output to external subsystems. It give all the other nodes permission to execute, as well aborting execution when needed. Through a finite automata that tracks current task execution it

Task name	Description	Start Command
Navigate	Navigates through a set of waypoints in the world frame.	ROARyNav
Align	Relative alignment in robot coordinate frame.	ROARyAlign
Manual	Enables manual drive through joystick controller.	ROARyManual
Grab	Runs lift motor sequence to place refusal bin on ROARy from ground.	ROARyGrab
Drop	Runs lift motor sequence to place refusal bin from ROARy to ground.	ROARyDrop
Pickup	Finds refusal bin, drives in front of it and initializes grab task.	ROARyPickup
Unload	Finds refusal truck, drives in front of the drop off forks and places bin on forks.	ROARyUnload
Mount	Finds and drives upon truck mount platform.	ROARyMount
Unmount	Drives of mount platform.	ROARyUnmount



ensures only one task executes simultaneously.

16.3.2 Requirements and limitations

A running ROS master. Host PC with Matlab, Simulink, Simulink Stateflow and robotics toolbox. ROARy custom message for feedback. See this link for how to install custom ROS messages to Matlab and Simulink: http://se.mathworks.com/help/robotics/ug/ros-custom-message-support.html?refresh=true

16.3.3 Design and interface

The control program is written in Simulink to increase modularity and easier implementation of new states and functionality. If a new ROARy task is implemented, only the control program has to be modified and every subsystem interfacing with the control program have access to the new task. The input and output interface of the Control Program, their data types and topics is specified in figure **??**. The input feedback come from every individual task node and their topic correspond to their name. In the same manner the output command topic goes to every individual task node and their topic correspond to their name. The output feedback goes to an internal

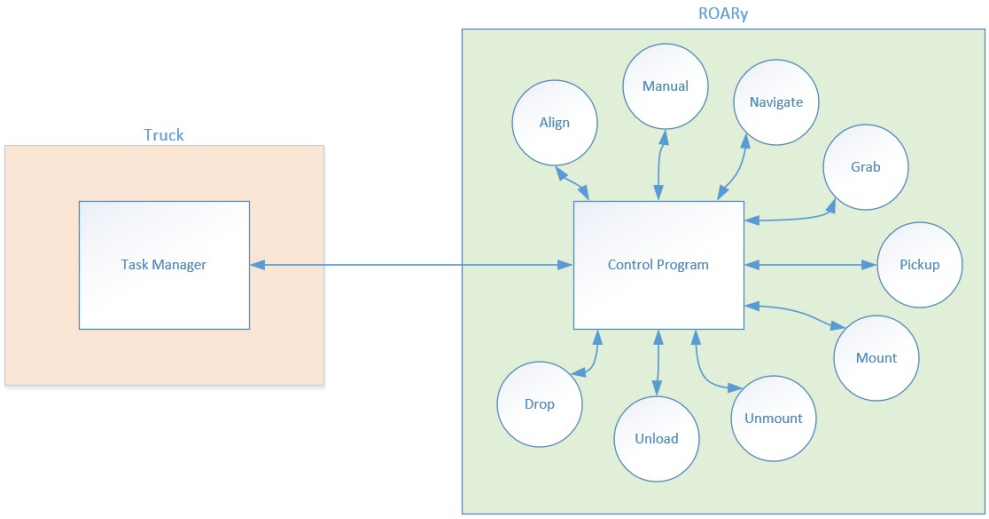
Task name	Feedback name	Description
Navigate	ROARyNavigationFeedback_comp	Navigate reached the last waypoint.
Navigate	ROARyNavigationFeedback_noWaypointOptained	Navigate cannot start due to no waypoints provided.
Navigate	ROARyNavigationFeedback_errorFollowingPath	Navigate couldn't reach the last waypoint.
Align	ROARyAlignFeedback_comp	Align reached its waypoint.
Align	ROARyAlignFeedback_noWaypointOptained	Align cannot start due to no waypoint provided.
Align	ROARyAlignFeedback_errorAligning	Align couldn't reach its waypoint.
Grab	ROARyGrabFeedback_comp	Bin was successfully placed on the ROARy.
Grab	ROARyGrabFeedback_errorGrabbing	Bin could not be lifted.
Drop	ROARyDropFeedback_comp	Bin is placed on the ground
Drop	ROARyDropFeedback_errorDropping	Bin could not be placed on the ground.
Pickup	ROARyPickupFeedback_comp	Bin has been found, approached and placed on ROARy.
Pickup	ROARyPickupFeedback_errorFindingBin	No bin could be found.
Pickup	ROARyPickupFeedback_errorAdjusting	Could not adjust position towards the bin.
Pickup	ROARyPickupFeedback_errorGrabbing	Bin could not be lifted.
Unload	ROARyUnloadFeedback_comp	Bin has been placed from ROARy on the truck.
Unload	ROARyUnloadFeedback_errorFindingTruck	No truck could be found
Unload	ROARyUnloadFeedback_errorAdjusting	Could not adjust position towards the truck.
Unload	ROARyUnloadFeedback_errorDropping	Bin could not be placed on the truck.
Mount	ROARyMountFeedback_comp	ROARy is placed on the lift platform.
Mount	ROARyMountFeedback_errorMounting	ROARy could not place itself on the lift platform.
Dismount	ROARyDismountFeedback_comp	ROARy moved off the lift platform.
Dismount	ROARyDismountFeedback_errorDismounting	ROARy could not move off the lift platform.

feedback translator that translates the custom ROARy feedback message to a string and finally publishes it to the output namespace feedback topic. Figure **??** show the design of the Simulink model. It can be broken into 5 parts (from left to right):

16.3.4 Method

The control program uses a finite automata to track state and current executing task. Between every task executed, the state machine returns to its initial idle state. Rising edge detection is used on state transitions in order to publish data (state, commands and feedback) only when ROARy makes state transitions. Figure 7 show the automata used for state space tracking.

The input data to the automata used for transitions are: cmd, feedbackResponse and transition. Cmd is the command integer converted from the input command string. When in idle, the cmd data will transition to another state.



State	Description
ROARyIdle	ROARy is idle and waiting for input command.
ROARyNavigating	ROARy is navigating through given waypoints.
ROARyAligning	ROARy is aligning towards given waypoint.
ROARyManualDrive	ROARy is operating through manual joystick commands.
ROARyGrabbing	ROARy is picking up garbage bin from ground to robot.
ROARyDropping	ROARy is putting garbage bin from robot to ground.
ROARyMountning	ROARy is mounting the truck lift platform.
ROARyDismounting	ROARy is backing off the truck lift platform.
ROARyPickup:FindingBin	ROARY is finding a bin for picking up.
ROARyPickup:Adjusting	ROARy is adjusting itself to found bin.
ROARyPickup:PickingUp	ROARy is ending pickup sequence with picking bin from ground to robot.
ROARyUnload:FindingTruck	ROARy is finding the truck.
ROARyUnload:Adjusting	ROARy is finding the truck for an unload.
ROARyUnload:DroppingOff	ROARy is ending the unload sequence with dropping the bin from robot to truck.

16.3.5 Test and simulation results

The control program has been tested towards the Navigation and Align simulations. Correct behaviour in sending data, receiving data as well changing state for these two simulation was recorded. The control program was also tested towards the truck task manager with no tasks running and manually giving the control program dummy feedback where it needed task feedback with correct result. The control program was built to a ROS target from Simulink.

16.3.6 Using the system-Dependencies, Installation, configuration and execution

16.3.7 Future work

Add e-stop state and init state.

State	Description
ROARyIdle	ROARy is idle and waiting for input command.
ROARyNavigating	ROARy is navigating through given waypoints.
ROARyAligning	ROARy is aligning towards given waypoint.
ROARyManualDrive	ROARy is operating through manual joystick commands.
ROARyGrabbing	ROARy is picking up garbage bin from ground to robot.
ROARyDropping	ROARy is putting garbage bin from robot to ground.
ROARyMountning	ROARy is mounting the truck lift platform.
ROARyDismounting	ROARy is backing off the truck lift platform.
ROARyPickup:FindingBin	ROARY is finding a bin for picking up.
ROARyPickup:Adjusting	ROARy is adjusting itself to found bin.
ROARyPickup:PickingUp	ROARy is ending pickup sequence with picking bin from ground to robot.
ROARyUnload:FindingTruck	ROARy is finding the truck.
ROARyUnload:Adjusting	ROARy is finding the truck for an unload.
ROARyUnload:DroppingOff	ROARy is ending the unload sequence with dropping the bin from robot to truck.

16.4 Navigation and localization-Mattias

16.4.1 Description

For localization, path planning and obstacle avoidance the predfined navigation stack that comes with the Husky is used. This includes Adaptive Monte-Carlo Localization (AMCL) for logalizing and D W A (DWA) for obstacle avoidance. More information regarding these algorithms as well the navigation stack can be found on the ROS wiki.

16.4.2 Using the system-Dependancies, Installation, configuration and execution

To run the localization, path planning and obstacle avoidance, run the test_map.launch file from the ROARy launch file folder. This launch file launches the map server for the static global map as well the AMCL launch file from the Husky navigation package (provided from Clearpath). To change the global static map, change the file name to the desired map file in the test_map.launch file (named map_file).

16.5 Simulation-Nabar

16.5.1 Description

The simulation system is implemented in Gazebo that helps to perform the functions of ROARy in the early stage of the robot building process.

16.5.2 Requirements and limitations

The simulation system requires the following basic operations of the ROARy:

- wheels: clockwise and anti-clockwise rotation
- arms: horizontal and vertical movement
- able to handle(find, transport) a refuse bin

16.5.3 Design and interface

The models of both ROARy and the refuse bin in the simulation system are designed as programmable models to handle their functionalities. The model of the ROARy is based on the husky's default model which is provided by Clearpath. The simulation system consists of a number of packages such as roarygazebo, roary_control, roary_io, roary_description, truck_description and refuse_bin_description. The interface of the ROARy's controller is designed to use the following commands:

- /joint1_position_controller/command
- /joint2_position_controller/command

16.5.4 Using the system (Dependencies, installation, configuration and execution)

Dependencies: In order to use controller in Gazebo, the simulation system requires effort-controllers package for ROS. To install the package run the following command:
$ sudo apt-get install ros-indigo-effort-controllers
To get the ROARy's model ready to use, husky's default model needs to be customized so that the model will be able to handle the ROARy's arm.

Installation: To get simulation system running, the above mentioned packages need to be compiled as following:
$ catkin_make -DCATKIN_WHITELIST_PACKAGES="roarygazebo;roary_control;roary_io;
roary_description;truck_description;refuse_bin_description"

Execution: The simulation system starts once the launch file runs:
$ roslaunch roarygazebo roary.launch

16.5.5 Versions

Current version of the simulation system can be found at the following location:
URDF models:
/opt/mdh/projects/roary/catkin_ws/src/
SDF models:
~/.gazebo/models/
Default husky models:
/opt/ros/indigo/share/husky_description and /opt/ros/indigo/share/husky_gazebo

16.6 Pickup and Unload-Anders Rickard Ludvig

The system to find the bin is first using camera to find the bin then the lidar to track the bin while moving towards it.

16.6.1 Uisng the system-Rickard

The bin localization function find bin consist of many sub folders, but it is only the file ”main_main” that needs to be run. At the top of the file the IP address to the host computer and the computer running ROS master must be specified.

When launching ”main_main” the function will be idle until it receives a command to start. ”main_main” should only be called when the garbage bin is expected to be in the camera frame.

When running ”main_main” 4 callback function will start: ”getROSLIDARCallback” Continiusly reads data from the lidar and saves it to the variable ”ROS_LIDAR” ”getROSImage-Callback” Continiusly Checks if new a new picture from the camera is available and saves it to the variable ”ROS_IMG_CHECK”. ”getROSROARyStartStopCallback” Checks on the topic ”/ROARy/Internal/FindBinCommand”, if the meassage is ‘‘ROARyFindBinStart” is recived the global varible ROS_RUN changes to 1. This variable decides when the program will advance and start to locate the garbage bin. If the command ‘‘ROARyFindBinStop” is recited the varible ROS_RUN will be set to 0 which in turn will return the function to idle. ”getROSROARyStopCall-back” Check for the command ”ROARyStop” on the topic ”ROARy/Commands/Stop” and sets ROS_RUN variable to 0.

When ROS_RUN = 1 the function ”SegmentBin” will continuously send the current position of the garbage bin.

Segment bin will start by locating the garbage bin with the camera function after that it will track and publish the pose of the bin until ROS_RUN = 0.
The function ”img_loc_bin” will Due to an sync error in the camera algorithm 5 pictures are taken in the function ”img_loc_bin” in order to avoid receiving old pictures.

The function “camera2lidar” transform an x coordinate in the camera frame to an angle in the lidar frame. The transformation is linear and if the camera or the lidar are moved or changed the parameters p1 and p2 inside needs to be updated.

The function ”ExtractBin” extracts and returns the index of all the scanned points that hit the garbage bin. Only a single point (that is hitting the garbage bin) is needed as input. The first time the function is run the point in the middle of the two points received from ”img_loc_bin” is used as input. After the first run the point that was in the middle of the previous iterations extracted points will be used.

Tuning functions The function ”ROAR_BIN.m” is used to calibrate the camera to only see the garbage bin. Adjust the sliders until only the garbage bin is silhouetted as best as possible. Pressing save will save the current settings to the file ”tresholds.mat” which in turn is read and used by ”img_loc_bi”.

17 Power supply-Albin Emil

17.1 Description

The power supply is responsible for distributing the power and to some degree protecting all the different systems inside the ROARy. The input power comes from a 6S LiPo battery that is used to power the lifting mechanism, and 3 different channels with 5, 12 and 24V from the battery that is included with the HUSKY.

17.2 Requirements and Limitations

Inputs:	22,2V @ Battery current	
	24V @ 5A	
	12V @ 5A	
	5V @ 5A	
Outputs:	22,2V @ 40A	
	24V @ 5A	
	12V @ 5A	
	5V @ 5A	
Battery:	6S LiPo	
	Husky included	
Protections:		
Short circuit:	YES	
Reverse polarity:	NO	
Over-voltage:	NO	
Reverse current:	YES	
Measurements:		
Current:	Partially	
Voltage:	YES	
Software operated:	YES	
Input connectors:		
LiPo battery	x1	
24V from Husky	x1	
12V from Husky	x1	
5V from Husky	x1	
Output connectors:		
Motors	22,2V	x 2
24V	24V	x 2
12V	12V	x 4
5V	5V	x 4
Computer:	12v	x 1
Servo:	12V	x 2

Table 10: Reqirements and Limitations

17.3 Design and Interface

The PSU was designed in Multisim and the PCB was later designed in Ultiboard. In Multisim it was designed to be modular and give a decent understanding of how it works at a glance, while not being overwhelming to look at for the first time. This was achieved by using subcircuits, which will be described below with the same name as in the Multisim file. The Ultiboard file is not as modular since it was designed to be space efficient, but it is still divided into subsystems so with small changes the layout should still be usable with some changes.

The PSU is divided into 4 main systems, where 3 is almost identical to one another. There is a system managing the LiPo battery and the 3 almost identical systems each manages one of the power outputs from the Husky. A CAN-card is docked onto the PSU to act as the intelligence. CAN-communication is not used, and the card was chosen because of availability and because they are easy to dock since putting pins or headers on them is up to the manufacturer.

17.3.1 Voltage measurement

Description:
Divide the voltage into the readable range (0-5V).

Requirements:
Take voltages of 24, 22, 12, 5 and divide them to as close to 3V as possible. Filter noise and clip voltages higher than 5V to protect the CAN-card.

Design and Interface:
The voltage is brought down to 3V by voltage division, the noise is filtered with a 0.1uF capacitor and a Zener diode is put in reverse to cut voltages higher than 5.1V, then they are read by the analog ports on the CAN-card.

17.3.2 Electronic relays and LEDs

Description:
There are LEDs indicating the status of the PSU, relays which switches the channels on/off and a chip that powers these things depending on the signals received.

Requirements:
The LED requirements was to have one LED that indicates that the PSU is at least partially powered, a LED for each power channel and an error LED (which can be used for other things) that can be turned on by putting a digital channel to high.

Design and Interface:
The LED named 5V indic in the Ultiboard file is always on when the 5V from the Husky is connected and the Husky is on. The chip A2982SLW was chosen since it can output enough current for the relays (4x24mA) and LEDs (6x30mA), it has 8 channels where each input corresponds to 1 of 8 output channels. The CAN-card puts a digital pin to high, which activates one or two outputs of the A2982SLW, which in turn powers a LED and for digital 0, 1 and 2 also a relay. The relays are apart from the fuses the only thing between input and output on the 5, 12 and 24V channels.

17.3.3 Motor relay

Description:
Creating the control signal for the safety supply, which powers the lifting mechanism.

Requirements:
Create a path to GND for the safety supply when a signal is received, and have a kill switch that overrides said signal.

Design and Interface:
There is a connector that connects to a NC-switch acting as kill switch, then a transistor switches on/off the supply to a relay which goes between GND and pin 2 on the BTS555.

Notes:
The transistor is currently a BJT, it would be prettier to have a MOSFET, that way no current would pass from the CAN-card.

17.3.4 Safety supply

Description:
This circuit is used to switch the LiPo channel ON/OFF, measure current and protect from reverse currents.

Requirements:
A chip that can switch a load of 40A, provide current feedback and protect from reverse currents.

Design and Interface:
The BTS555 is specified for 45A, but 2 were used to ensure that there is no heating issues. It has current feedback and the same circuit as in “Voltage measurement" was used to make it fit in the readable range. The control signal is an open/closed connection to GND which is supplied by “Motor relay", and applied to pin 2. The capacitors are there to avoid voltage drops.

Notes:
BTS555 is on when the control signal is shorted to ground and it does not protect against putting a battery in reverse.
The resistors in the Multisim and Ultiboard, for the current/voltage measurement, might not be the ones that actually ended up being used on the PSU

17.3.5 CAN connections

Description:
Dock for a CAN-card that send and read signals.

Requirements:
Analog, digital, power out and supply pins are needed.

Design and Interface:
Headers were put on the PSU for the CAN-card to dock into, and a small capacitor is used to try and avoid restarts due to voltage drops.

Notes:
CAN GND is connected to the PSU GND to get the same reference.

17.4 Using the system

A CAN-card us needed to use the system and the ports used are:

DIO0	5V
DIO1	12V
DIO2	24V
DIO3	LiPo
DIO4	Error LED
A0	LiPo voltage
A1	5V voltage
A2	12V voltage
A3	24V voltage
A4	LiPo current

Table 11: Pin mapping

Putting a 1 on one of the DIOs enables that channel. The voltage measurement have been scaled so all of them should measure about 3V when they have appropriate voltage.

17.5 Test and simulation results

There weren't any tests conducted apart from functionality, and then it was used in the ROARy and have been working fine.

17.6 General notes

17.6.1 V1.2 issues

GND return path for 5, 12 and 24V is too thin, this was resolved by adding a cable.

The GND from the LiPo and the battery included with the Husky is not connected, which means the current and voltage from the LiPo battery can't be measured. The fix to this was to solder a small bridge between the different GNDs.

Relay silkscreen is wrong, which was not a problem since there is enough space to fit them anyway.

All capacitor symbols on the silkscreen is wrong.

17.6.2 V1.3 fixes

GND return path has been made larger.

The different GNDs are connected by a Net bridge in Ultiboard.

Fixed the relay silkscreen.

Made some copper pads bigger to make it look nicer and make manufacturing easier.

The capacitor symbols have been corrected.

17.7 Troubleshooting

Make sure there is 5V between pin 9 and pin 12 of the A2982SLW.

If a DIO is enabled a LED should light up, and for DIO0-2 you should hear a relay tick. For DIO3 and 4 a LED should light up, if nothing of this happens make sure there is 5V on the 2982SLW output.

If the LiPo channel does not turn on with DIO3, make sure there is a closed connection over J25, then check the output of transistor Q2 and make sure the relay is closed

17.8 Future work

If the plans to use a Segway or some other platform than the Husky were to go through, the PSU would most likely need a complete redesign, since it is unlikely that the new platform provides 3 different voltage levels like the Husky.

There was talk about needing more current on 5V, so a new revision would probably include a DC-DC converter.

Add the different voltage levels as labels on the connectors to avoid all possibility of confusion.

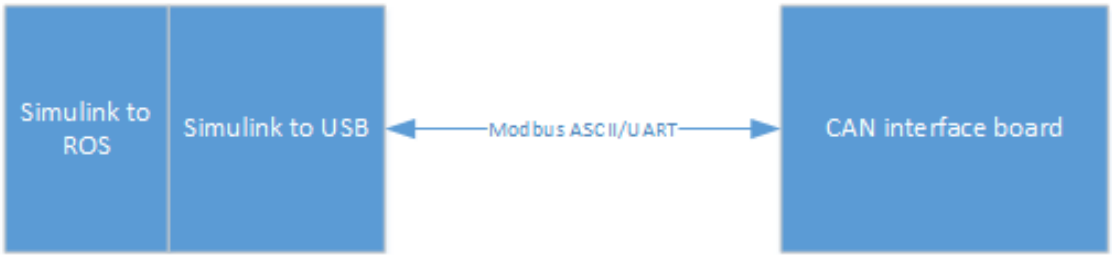
17.9 Power supply software-Daniel-Jacob-Mattias

17.10 Power supply software-Can card

This section describes the communication between between the PSU controlling card and the ROARy on board PC. It also describes how the PSU controlling card works and how a developer should do to work with the CAN cards.

The currently used cards are called CAN cards and are based upon the processor AT90CAN128. They are CAN compatible with UART, PWM, ADC and other GPIO ports. In this project, they are used for controlling the power supply unit of the ROARy. The messages sent from from the onboard ROARy PC are formatted according to the MODBUS format and sent to the CAN card over UART. The CAN card in turn is directly connected to the power supply unit which can read its voltage channels and also have the option to enable/disable them. The picture below describes the communication:

Figure 19: Power supply unit communication



17.11 Requirements and limitations

To start using the current code of the PSU controlling card, a few things need to be installed.

- GMAT GPL 2012 (AVR microcontroller format hosted on windows

- Atmel studio
- Atmel debugger (Atmel ICE used during this project)

The limitations which comes with this approach is that the developer has no option of setting breakpoints, and has to connect the card on to a power supply or a battery. UART can be used to get information of what is happening inside the code.

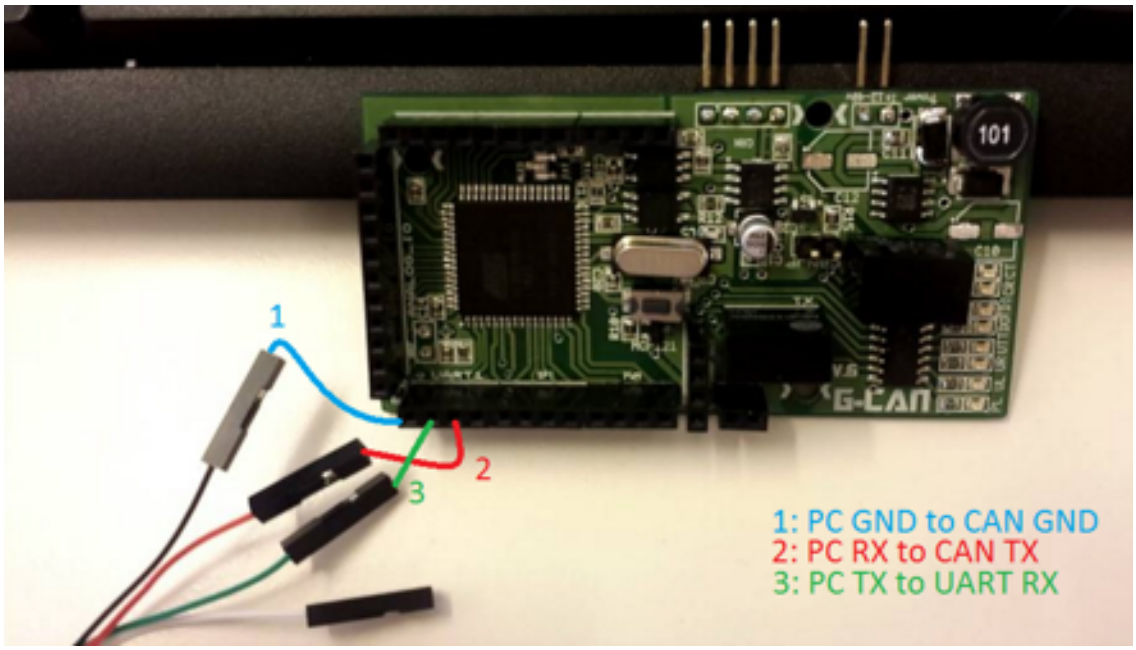
17.12 Design and interface

As mentioned before, the MODBUS communication scheme is used to communicate between the onboard PC and the CAN cards. Currently, the following UART configurations are set:

UART channel	PORT 0
Baudrate	115200

These configurations are set within the CAN cards and make it possible to communicate with simulink. If the developer does not wish open simulink for each test, a serial client can be used to communicate with the can cards. During this project, Hercules was mainly used, but other serial client such as putty or termite should work fine. To connect the CAN card with a computer, use the setup described in the picture below:

Figure 20: CAN card to PC setup



To communicate with the CAN card PSU controller, the following format is being used:

Format	Startbyte	Address	Function code	Data	CRC	Stop byte
Size(bytes)	1	2	2	4	2	1

The communication between the computer and the CAN cards is divided into read and write operations. The write operations are usedfor controlling the digital pins on the CAN cards which in turn controls a relay on the PSU. The read operations read the current of an analog pin of the CAN card. The different operations are described in the table below:

Write operations:

Address	Function code	Data	Affected pin	Action
06	06	0001	Digital pin 0	Restarts LIPO battery
07	06	0001	Digital pin 1	Restarts 5V relay
08	06	0001	Digital pin 2	Turns off 12V relay
08	06	0000	Digital pin 2	Turns on 12V relay
09	06	0001	Digital pin 3	Turns on 24V relay
09	06	0000	Digital pin 3	Turns off 24V relay
0A	06	0001	Digital pin 4	Turns on motor relay
0A	06	0000	Digital pin 4	Turns on motor relay

Read operations:

Address	Function code	Data	Affected pin	Action
01	03	0000	Analog pin 0	Reads LIPO battery current
02	03	0000	Analog pin 1	Reads 5V current
03	03	0000	Analog pin 2	Reads 12V current
04	03	0000	Analog pin 3	Reads 24V current
05	03	0000	Analog pin 4	Reads motor current

17.13 Method

–Hello

17.14 Test and simulation results

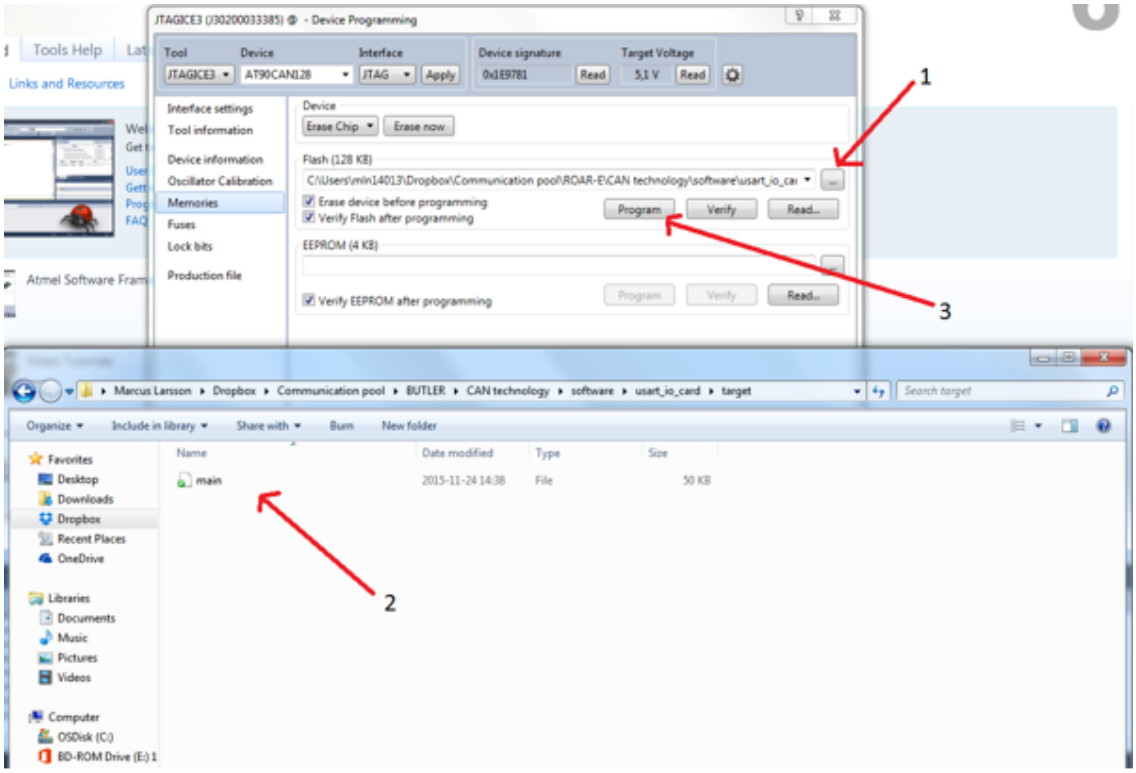
The test study was firstly done with hercules as command program. All commands were sent to the CAN cards and measure with oscilloscope to assure us that the messages came through. The second test was done with a windows simulink PC connected to both the CAN card and an Ubuntu ROS PC, whereas the commands were sent from ROS, the test results were once again verified by oscilloscope measurements. The third and last test was done on the actual PSU where the CAN card was connected to the PSU, reading voltage channels and setting relays, the can card was controlled from a ROS PC, and the results were verified by looking at the status LED's and also measuring the current of the voltage channels.

17.15 Using the system

Using the system can be a little complicated. To program and start working with the CAN cards, a few software programs are needed. The language which is used is ADA, which must be compiled with GNAT GPL 2012 using the AVR microcontroller format (hosted on windows).

When building files using this integrated development environment, an executable ELF file can be found in the target directory of the project. The generated ELF file can then be used by Atmel studio to program the CAN card. By following the process in the picture below, a CAN card can easily be programmed.

Figure 21: How to program the CAN cards



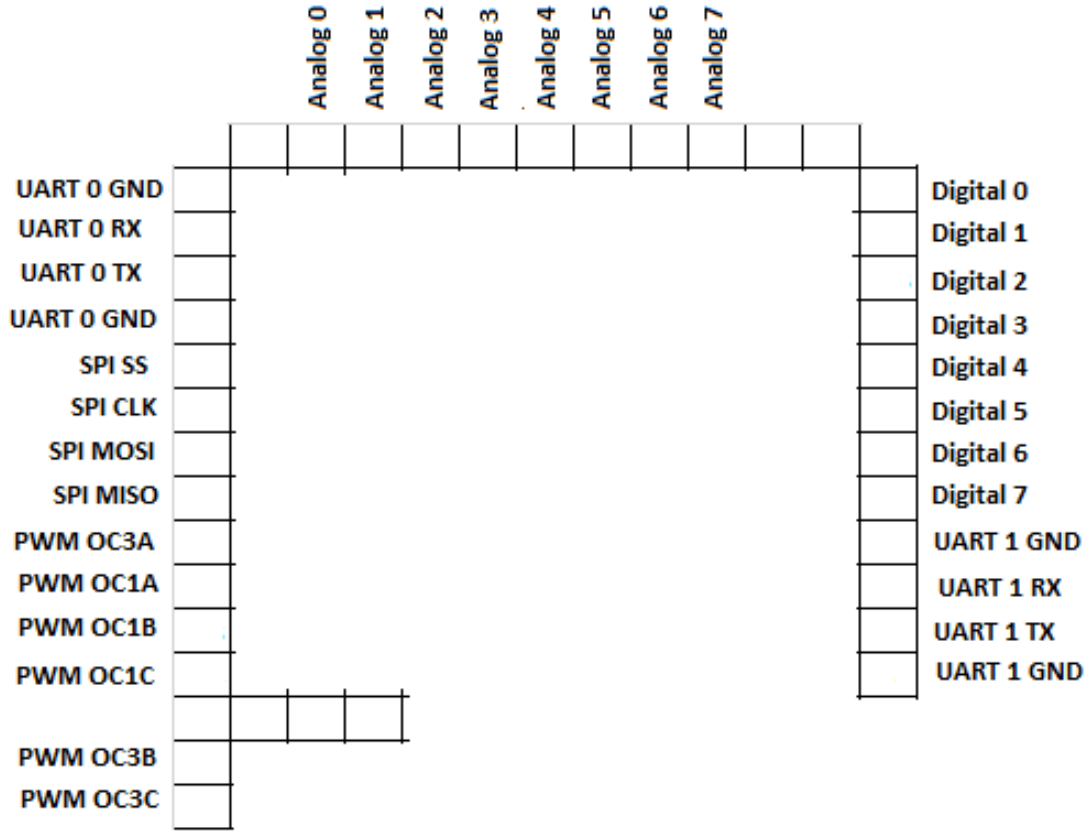
It is not recommended to use another compiler than GNAT GPL 2012 (AVR microcontroller format hosted on windows), even when writing single functions to be transferred to the real code. This is due to restrictions of the GNAT GPL 2012 (AVR microcontroller format hosted on windows). An example is when the MODBUS protocol was written. The MODBUS protocol was written in GNAT GPL 2015, and when trying to transfer that code into GNAT GPL 2012, several errors occured which took a long time to solve.

When searching for the pinouts on the CAN cards from the processor legs, it may be a little difficult to find the correct mappings. Using the figure below can make it a little easier to find the desired pinouts on the CAN version 6 card.

The debugger must also be connected in a specific way to the CAN cards in order to have it working properly. For simplicity, follow the example of the current soldered debugger for correct pin connections or follow the pinout in the ATMEL ICE user manual.

There exists two different jumper connections, one CAN jumper header and one ADC jumper header. In order to get CAN communication properly, one CAN jumper must be placed onto the

Figure 22: Processor pin mapping



CAN card. The same goes for the ADC, if ADC is going to be used, the ADC jumper must be in place.

To analyze which messages are transferred through CAN, the kvaser leaflight v2 can be used. This is a CAN to USB converter which visualizes the CAN messages on a PC. The software which should be used is KVASER CAN KING.

Lastly, the fuses which are on the processor set are listed in the table below:

Fuses:

BODLEVEL	DISABLED
TA0SEL	[ ]
JTAGEN	[ ]
SPIEN	[X]
WDTON	[X]
EESAVE	[ ]
BOOTSZ	4096W_F000
BOOTRST	[ ]
CKDIV8	[ ]
CKOUT	[ ]
SUT_CKSEL	EXTXOSC_8MHZ_XX_16KCK_0MS
EXTENDED	0xFF (valid)
HIGH	0x99 (valid)
LOW	0xDF (valid)

Using the code The current CAN cards has lots of implemented ready to use code. These libraries can be found in the firmware directory and include library functions to configure the pins to do specific tasks such as PWM and SPI and also base addresses to all processor pins. It is important to mention that the current mapping of the UART pins is wrong. UART port 0 in the code is actually UART1 on the CAN card, and reverse.

17.16 Future work

Future work regarding the PSU controller include making the controller intelligent. The controller should be able of making it's own decisions, e.g. if an abnormal current level is detected, the controller should be able to turn that channel off. This would decrease the response time of powering off a channel. Another improvement could be to re-write the code in to C-code so that it would be possible to debug with atmel studio, however this is a very extensive process, which may require alot of time since all firmware libraries that are needed has to be re-written as well.

17.17 PowerSupplySoftware-PC

17.17.1 Description

The power supply interface software is a Simulink model that handles both enabling and disabling power supply channels as well continuously reads power supply channels state of charge through ADC register on the power supply's mounted CAN card. Enabling and disabling of channels are executed through input ROS commands and state of charge data is made available to the ROS network through specific topics. Figure x below describes the interface of the power supply model.

17.17.2 Requirements and limitations

* A running ROS master. * Host PC with Matlab, Simulink, Simulink Stateflow and robotics toolbox. * Correctly configured COM port connected to the power supply CAN-card Tx/Rx.

17.17.3 Design and interface

Figure X show the Simulink model of the power supply interface. The model can be broken down into two parts. 1. Enabling and disabling channels by sending write commands to the CAN card. 2. Request channel state of charge by sending read commands to the CAN card.

17.17.4 Future work

A current problem with the power supply interface software is that if a single requested CAN-message is lost the whole sequence of reading every ADC channel individually will stop running. This is due to that the state machine will not transition until a message is received corresponding to the previously sent request. Therefore, if the requested message is lost the state machine will never transition and request message from the next ADC channel. This has to be addressed in future versions. One solution would to implement a timer for each ADC channel. In case of time out (a requested message was never received) the state machine should transition and request message from next ADC channel, not updating the timed out channels ROS topic. The power supply interface is currently only available in Simulink and not as a standalone ROS node. This is because no solution how to build Simulink models with COM ports to ROS compatible code was found. Since the power supply needs to run on the PC mounted inside the Husky, this issue has to be solved for future versions of the power supply interface.

18 Network and communication-Daniel Nabar

18.0.1 Description

The network and communication system connects the computers inside ROARy as well as providing the network with Wi-fi.

18.0.2 Requirements and limitations

18.1 Design and interface

18.2 Method

The 802.11n standard was used for the access point to provide the highest speed at 2.45 GHz. The intention is to place the access point on top of ROARy to have good transmission performance. Additionally this placement would make ROARy look more “human”.

The following 2 methods are being used for the networking topology.

- WDS (main-main connection):
  The both access points are acting as the main routers and will communicate with each other through WDS technology.
- Router-Repeater:
  Router and repeater architecture is the simplest structure to extend the Wi-Fi connection and hinder the interference between the APs.

18.3 Test and simulation results

The interference of signals between the access points can be occurred when the truck and ROARy stand/stay in less than 3 meters from each other. A number of methods can be considered to avoid interference. For example, to use a varied number of the channels for each access point in WDS or to use a structure such as pair router-extender(Repeater).

18.4 Using the system (Dependencies, installation, configuration and execution)

The router can be configured using the webbrowser and entering the IP adress of the router in the URI field.

18.5 Versions

18.6 Future work

Regarding the integration test we realized that managing the data traffic through the network is the key to improve the performance of the entire ROAR system. For instance, it is recommended to have separate networks of ROARy and quadcopter.

19 Results-Daniel

ROARy have successfully been able to perform autonomous waypoint following and picking up bins. However the testing is far from finished and the robot system is not stable enough for usage in public demo scenario. Navigation have proven to work very well in environment with medium to high level of detail. We've seen that SLAM using GMapping ROS package fails if the environment is big and containing low level of detail. Truck registration for the Unload system is not finished. Data segmentation, integration and testing has not been started on.

20 Future work-Daniel

20.1 Collaborating robots

In the future, ROARy can collaborate with another robot or human to collect the refuse bins. The ROS navigation stack efficiently implements methods which can be used for such collaboration. Today ROARy uses a cost map describing the cost of entering different areas in the map. The cost is added to the movement of the robot and causes the robot to avoid obstacles. This cost map is currently modified by the LIDAR which projects the presence of obstacles into the map and causes the robot to avoid these obstacles. This cost map can be modified more generally, e.g. information relating to presence of another robot or human can be added.

21 Appendix

21.1 Contact list-PL

21.2 Cables and schematics-PL

List of cables and schematics is provided in the files "CableList.xlsx".

Name	Company	Email	Comments	user name	telephon
Daniel Adolfsson	MDH	dla.adolfsson@g	**Project manager**	dan11003	
Ragnar Moberg	MDH	ragnar.moberg@	**Mechanical Manager**	rmg11001	
Oscar Svensson	MDH	hydralhammare	Mechanical Engineer	osn10001	
Albin Barklund	MDH	albin.barklund@	**Electrical Manager**	abd11003	
Roxanne Anderberg	MDH	roxanne.anderb	Electrical Engineer	rag11001	
Erick Vieyra	MDH	sevieyra@gmail.	Electrical Engineer	svz13001	
Emil Johansson	MDH	ejn11014@stu	Electrical Engineer	ejn11014	
Jonatan Tidare	MDH	jte11001@stude	Software Engineer	jte11001	
Tobias Kriström	MDH	tkristrom@gmai	Software Engineer		
Rickard Holm	MDH	holm.rickard@g	Software Engineer		
Mattias Bäckström	MDH	mbm11007@stu	Software Engineer	mbm11007	
Ludvig Langborg	MDH	llg11005@stude	Software Engineer	llg11005	
Anders Olsson	MDH	aon11013@stud	Software Engineer		
Mobin Hozhabri	MDH	mhi14003@stud	Software Engineer	mhi14003	
Dennis Eklund	MDH	eds04001@stud	Software Engineer		
Jakob Danielsson	MDH	jdn11003@stud	**Dat communication manager**	jdn11003	
Nandinbaatar Tsog	MDH	ntg14001@stud	Software Engineer	ntg14001	
Tobias Andersson	MDH	tan10006@stud	Software Engineer	tan10006	
Marcus Larsson	MDH	mln14013@stud	Software Engineer	mln14013	
Sudhangathan Bankarusamy	MDH	sby14001@stud	Software Engineer	sby14001	
Robin Andersson	MDH	roband3333@hc	Other		
Peter Cederblad	MDH	peter_cederblad	Other		