1 Software system design

The main goals for the software system was the following:

1. To separate safety critical and non-safety critical code as much as possible.
2. Building the non safety critical part to be easily extended and changed without needing to even recompile other parts.
3. Build the safety critical part as robust as possible by regarding it as a hard real time system.
4. Have a flexible mission construct that can be sent to the robot to be carried out autonomously.

To begin with we had the software system from last years project (?ref?). After further examination it was clear that to be able to reach the goals the whole software system needed to be redesigned and rewritten. A design fulfilling the goals is presented in this section. First the mission construct is explained in section 1.1. Second what the big parts are and how the fit together is described in the overview (section 1.2), after this the design of each part is described in detail.

1.1 The mission construct

A mission is constructed as a series of simple tasks that the robot is to carry out to accomplish some work. For example if the robot have a snowblower accessory attached and we want it to carry out the work of clearing a area of snow a mission like this could be constructed:

1. SET ACCESSORY TILT: to 100%.
2. GOTO: starting point.
3. SET ACCESSORY TILT: to 0%.
4. ACCESSORY COMMAND: hold shute at 30^o.
5. ACCESSORY COMMAND: start blower.
6. FOLLOW PATH: plow area path.
7. ACCESSORY COMMAND: stop hold shute.
8. ACCESSORY COMMAND: stop blower.

In this example mission, first the accessory is tilted up to lift it upp as much as possible. Then it goes to the starting point. After this it tilts down the accessory, tells the accessory to hold the shute at a angle so the snow is moved to a optimal spot and start the blower. After the accessory is set up it follows a path that covers the area. When it's done clearing the snow it turns of the

parts of the accessory it turned on. This shows how a mission can be constructed using a list of tasks.

The type of tasks that can be used is given an overview in table 1 and detailed in the mission executor system in section 1.3.2.

Task type	Description
GOTO	Move the robot to a specific GPS point.
FOLLOW PATH	Make the robot follow a path represented as a list of GPS points.
WAIT	Wait for a set amount of time.
ACCESSORY COMMAND	Send a command to the sccessory.
SET ACCESSORY TILT	Tilt the accessory a percentage where 100% is as furthest from the ground and 0% is closest to the ground.

Table 1: Overview of task types.

1.2 Overview

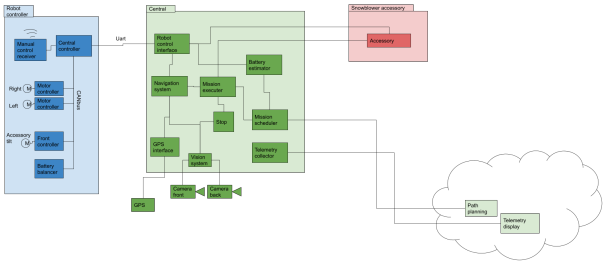

Figure 1: Overview of the software architecture.

The design of the software system is made up of different parts and are shown in figure 1. In figure 1 each system is symbolized by a square with a describing name and lines symbolize communication between systems. Some communication lines are omitted for clarity, what other system each system communicates with is described in the more detailed description of the system design.

The design was divided into 4 main parts, each part has its own color in figure 1:

The safety critical part is shown in blue and controls the robot directly. It's made up of mikrocontrollers running custom firmware written in rust using the RTIC framework (?ref?) except for the motor controllers that run the Vesc firmware (?ref?). This systems in this part communicate with each other using one single shared CANbus and communicate with all other systems over a UART link to the robotcontroller interface.

The non safety critical part is shown in the green big rectangle. This part handles everything needed to do to carry out a mission and sends the proper commands to either the robot controller or the accessory. This part is made up of many interchangeable systems orchestrated in a Arrowhead local cloud (?ref?).

The red part is the accessory, it controls the accessory directly and is a system in the local cloud. The accessory is designed to be able to be any kind of accessory. In our test case it is a snowblower.

The last part in light green in the little cloud is a separate Arrowhead local cloud running on a server somewhere. It is designed to be the interface between the robot and the world being able to get telemetry from the robot and send missions for it to do.

To show how the systems may work together a sequence diagram is shown in figure 2. In the sequence diagram the robot first do a mission where it navigates to a point and is preempted by a higher priority mission to charge the robot. The way preemption is handled is that the mission executor when it's get a new mission will construct a new mission that will carry out the work left on the current mission and send this new mission to the mission scheduler.

1.3 Design of all the parts

In this section the design of each system is described in detail.

1.3.1 Mission scheduler

The mission scheduler makes sure the missions are executed according to the following rules:

1. A mission is sent to the executer only when it is ready or running a different mission.
2. When the mission executor is ready the pending mission with the highest priority is sent to the mission executor to execute, if there are more then one pending mission with the highest priority the one received first is sent.
3. If a mission with a priority higher then the one running on the mission executor is received it is directly sent to the mission executor.

It provides the following services:

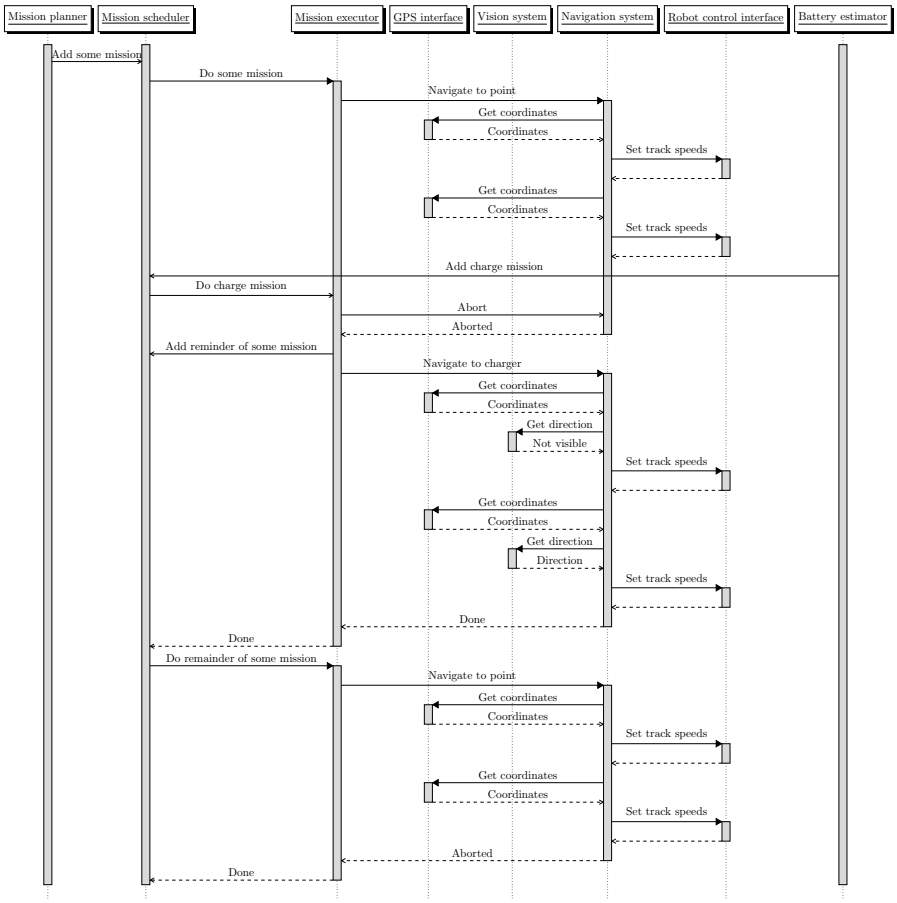

Figure 2: Sequence diagram for snowblower robot systems executing a mission and then interrupt it with a charge mission.

Add mission: Add a mission to be scheduled.

Request

Name	datatype	description
priority	int	The priority of the mission.
mission	Mission	The mission to be scheduled

Response

Name	datatype	description
status	Enum	Either ADDED or ERROR.

Executor Ready: Should only be used by the mission executer to indicate that it is ready.

Request

Name	datatype	description
status	Enum	Either READY or NOT_READY.

Response

The response is always the string "OK".

1.3.2 Mission executor

The mission executor executes the mission by for each task sending the corresponding request to the corresponding system and wait until it is done. This system needs to know what service to call for each task type and whit what data. Therefore it can be seen as a translator between task and service call. It also need to know when the called service is done and this is done by sending a id with every service request and waiting until the same id is sent back in a Task done service request. It provide the following services:

Task done: Is used by all systems consumed by the mission executor to tell it that the task is done.

Request

Name	datatype	description
id	long	The id of the task that is done.

Response

A object with a status string = "ok".

2 Theory

Theory behind the design Arrowhead, SoS etc

3 Methods

software requirements. Design goals. Software design.

4 Result

What requirements where fulfilled?

5 Discussion

How did the software design and implementation meet the design goals?

6 Conclusion

What was archived? Further improvements?

A Documentation