Group 1 / Gruppe 1

Date:17/12/2021

Height measure system / Høydemåler system

Members

Johannes

Øystein

Alexandr

Index

Introduction

We have been tasked to build a height measuring device, and this report accounts for the result.

System requirements / task guidelines

See attachment: Studentoppgave høydemåler.docx

System overview (design)

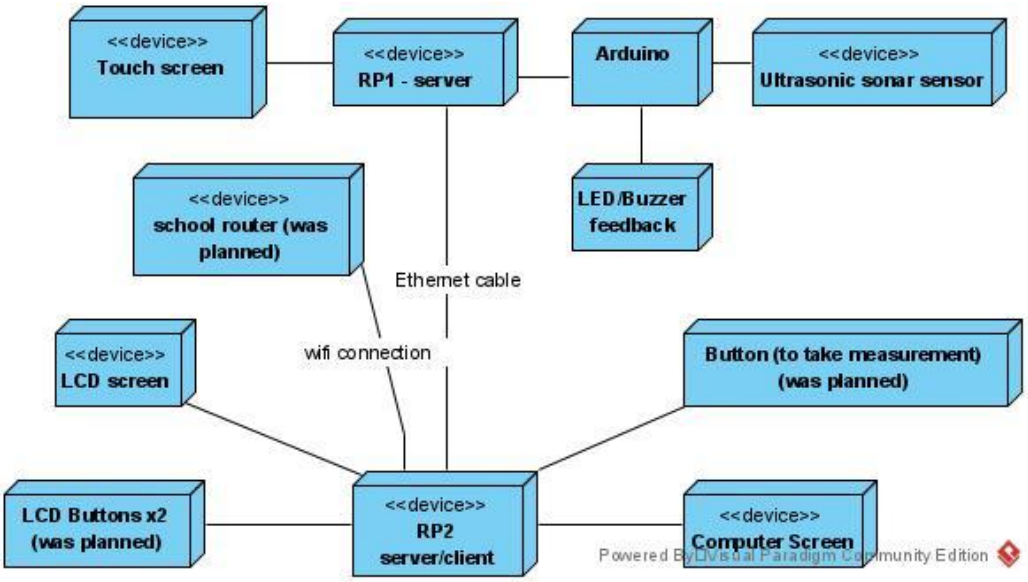


Diagram 1: Overall System design

Solution

Workload spread / Time usage

Week	Task(s)
44	Arduino introduction and practice. Group creation and presentation of possible projects.
45	Intro to Agile methods. Continued practice with Arduino tasks. Chose assignment: “Height measure system” (Høydemåler).
46	Learn use of Linux, Github, Git commands. Work on individual elements.
47	Work on individual elements.
48	Integration of base functionality with console interface
49	Integration of html interface, lcd screen output.
50	Final testing and documentation writing.
51	Presentation with question asking.

All tasks were divided and chosen freely among group members. Core functionality was identified and chosen first. We regularly convened and decided new tasks based on what was completed and what functionality we wanted to include next. Each member will now account for their individual contributions.

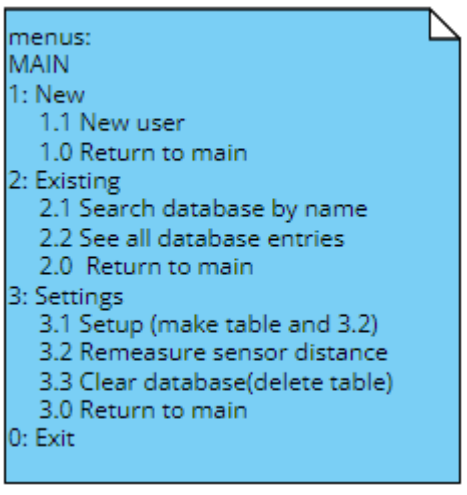
Johannes

Tasks

- RP2 Main.cpp for use through the console.
- Coms files for output to user through console
- Database interaction
- Integration of functionality into main.cpp
- Testing related to RP2

Design

RP2 Main.cpp / console interface
In one of the addable parts to the system, a web interface was listed. I saw that as too large a leap to start with. I chose to go for the simpler option of starting with a user interface based on the console and navigating with user input in the form of numbers.
In this document, console menus will be described with numbers; options within the main menu being a single digit between 1 and 3, additionally 0 being exit option/input in every menu. Sub menus are written with two digits split by a “.”, for example 2.2.



Note 1: menus

Coms.h and .cpp
The concept is to have a separate class with all output messages within ,so that Main.cpp creates an instance and then can call functions that contain output messages. In the paragraph entitled “Changes”, you will see that the files have changed names.
Database.h and .cpp
I would like to say that this was not the easiest thing to create. It required thorough research on my part, to learn about databases, since I did not participate in the database part of the course. I eventually chose to go with sqlite 3 as the basis for the database.
The initial design of the databases’ functionality was left to me, and I chose the following functions: Insert new, search by name and view all entries.

Changes

RP2 Main.cpp / console interface
The original design of the console interface did not contain a menu option regarding settings. Settings and its options have been added, because the functions they call upon have already been made as a core feature, or for testing purposes. 3.2 falls into the category of using a core function, as that function is also used to set the variable to calculate height of the user. 3.3 is for deleting the database table, and comes from the necessity to periodically empty the database during testing.
The main.cpp file has been continually updated during development.
Coms.h and .cpp
Coms ended up being split into Coms_NO and Coms_EN to support both languages, either Norwegian or English. Depending on what language is desired, the code must be changed in Main.cpp, as the chosen language option is hard-coded, depending on what #include and accompanying compilation is done.

Database.h and .cpp
The addition of functions arose by necessity, due to attempted integration of other files and hardware. The lcd screen needed additional functionality to get the last input from the database. This, in turn, meant the table required a unique int index. However, this ended up failing, because no variables could be changed within the save_callback(...). In order to fix this, I had to create a pointer to the class. That is set within the class constructor and initiated in the cpp file. This pointer can now be used to allow changes to variables from within a static function.
The addition of the main-dynamic led to the need for increased functionality. This includes checking if a name is used, as well as a function to update a user with a given name. These new functions were made in cooperation with Alexandr.

Implementation

Here is an example sequence diagram, showing the sequences from startup to sub menu 2.2 , see diagram 2. The diagram also shows an extra event, when an input within the main menu is of an invalid type or out of range.

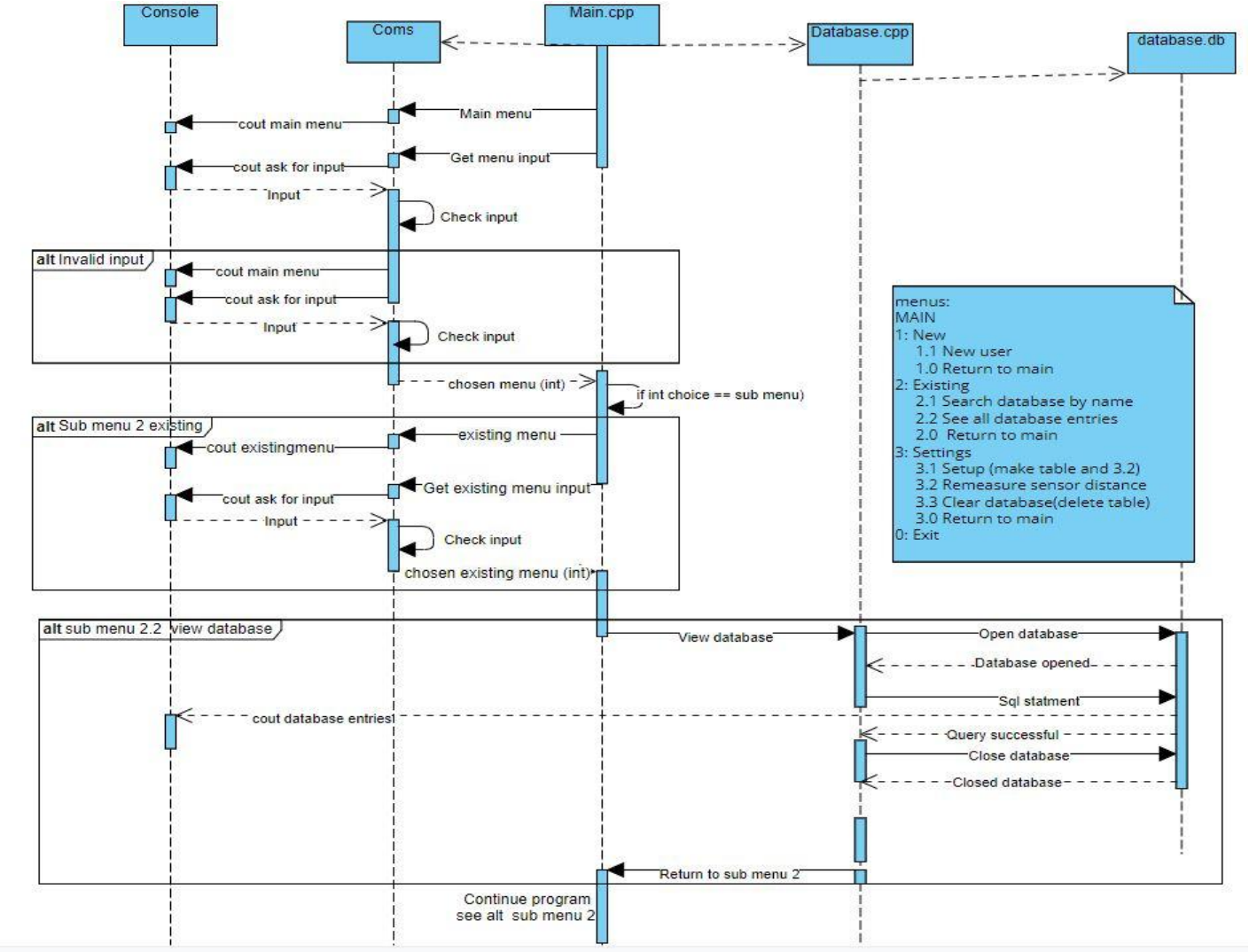


Diagram 2: Sequence (startup to submenu 2.2)

Alexandr

Tasks

Communication between RP1 and RP2

Communication type is sockets using TCP/IP protocol (see diagram 1). RP1 and RP2 are connected in a local area network (LAN) using static IP on both.
RP1 runs the server (RP1/main.cpp). RP2 sends requests using function `int sendRequest(char * msg)` in file `send-distance-request.cpp`. RP1 will initiate `measureHeight()` once a request with message “height” is received, else it will reply with a bad request code “10000”.

Settings

Settings package can be seen in the context of other packages on diagram 7. Value for sensor baseline is stored in data.xml under tag <sensor>. Update of sensor value is done by sending request to RP1 and store returned value `int readDistanceAndUpdateXml()` in file `read-distance-and-update_xml.cpp`. Activity diagram for this function see diagram 3.

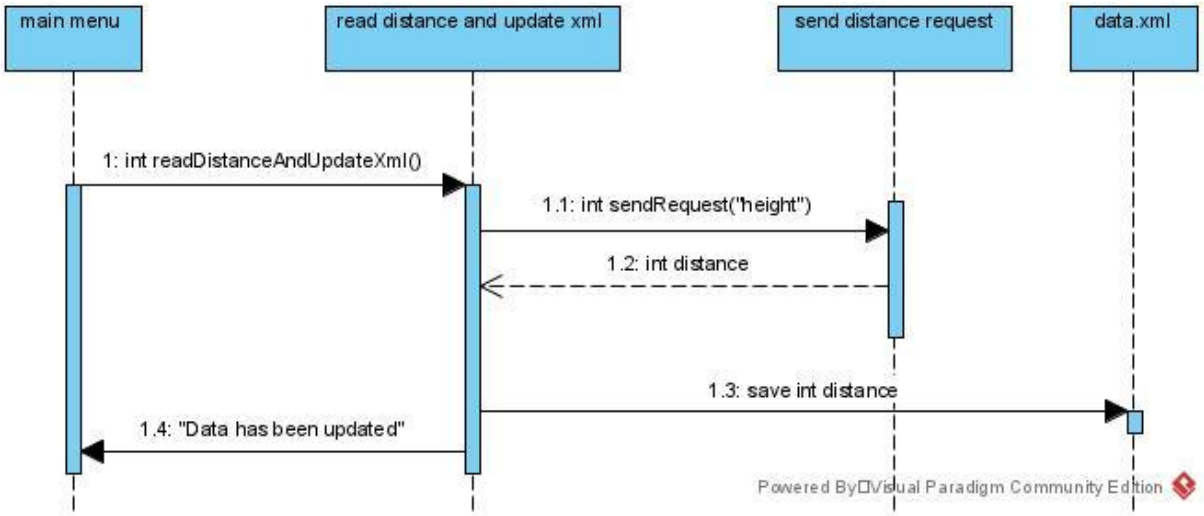


Diagram 3: Activity diagram for readDistanceAndUpdateXml().

Take height measurement

To take height measurements, use the function `int takeHeightMeasurement()` in file `take-height-measurement.cpp`. Activity diagram for this activity see diagram 4.

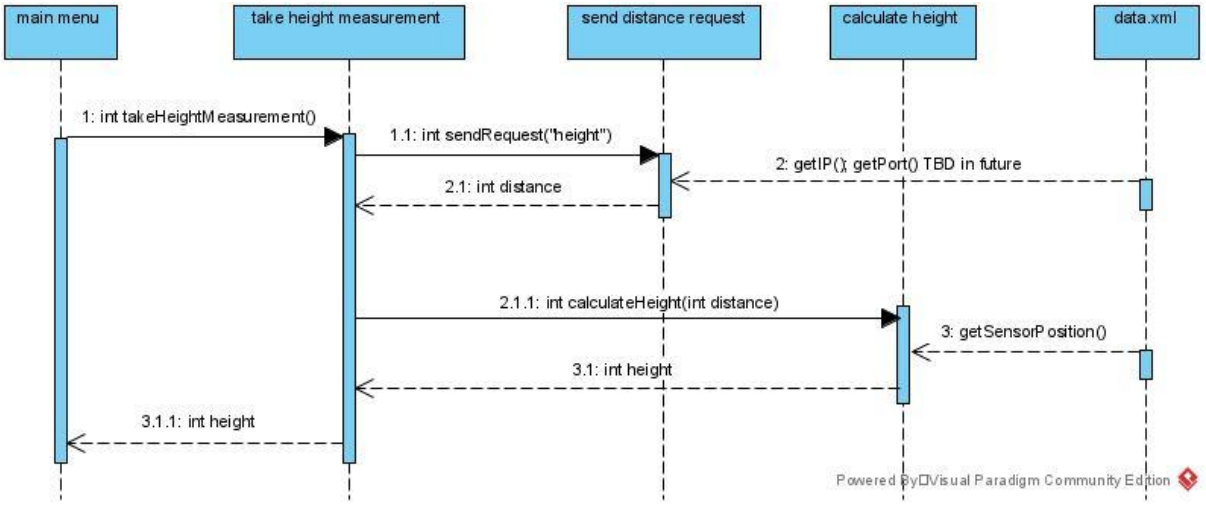


Diagram 4: Activity diagram for takeHeightMeasurement().

Web interface

Web server `main-dynamic.cpp` is a stateless server that will reply to URL requests with template rendered html code. See use case on diagram 5 for the user menu. Diagram 6 shows menu selections and related URLs.

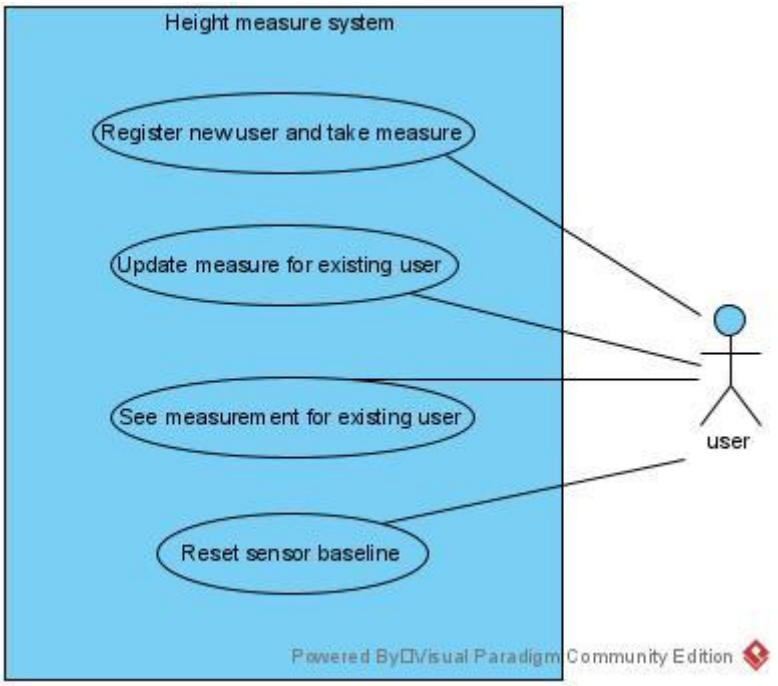


Diagram 5: Use case for web UI

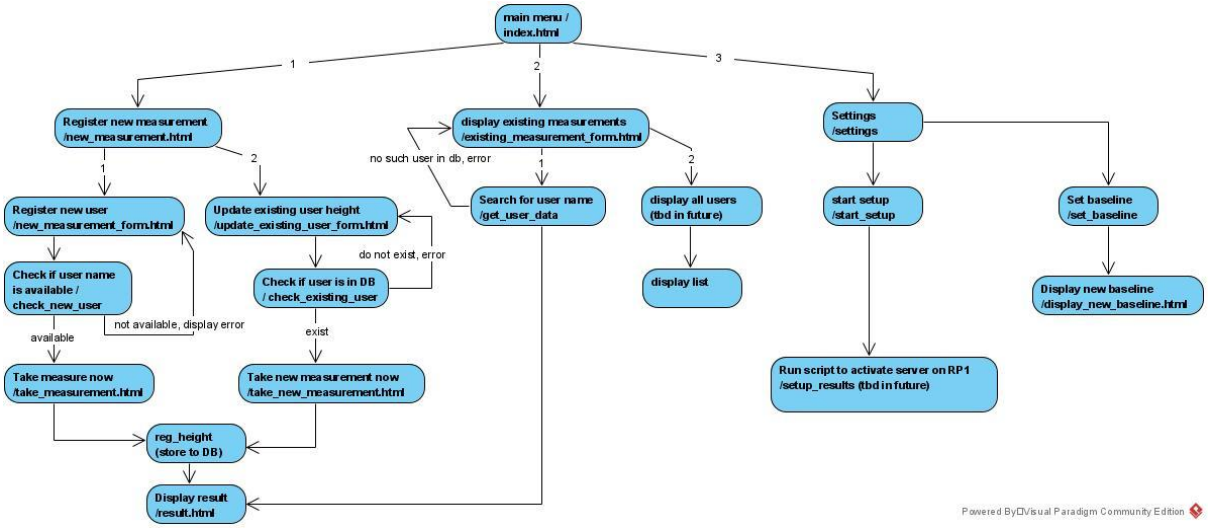


Diagram 6: HTML menus and related URLs

Øystein

Tasks

- Interfacing external gadgets: sensor, button(s), display
- Raspberry GPIO pin handling
- Arduino-Raspberry communication (Serial)
- Function measureHeight(), RP1

Design

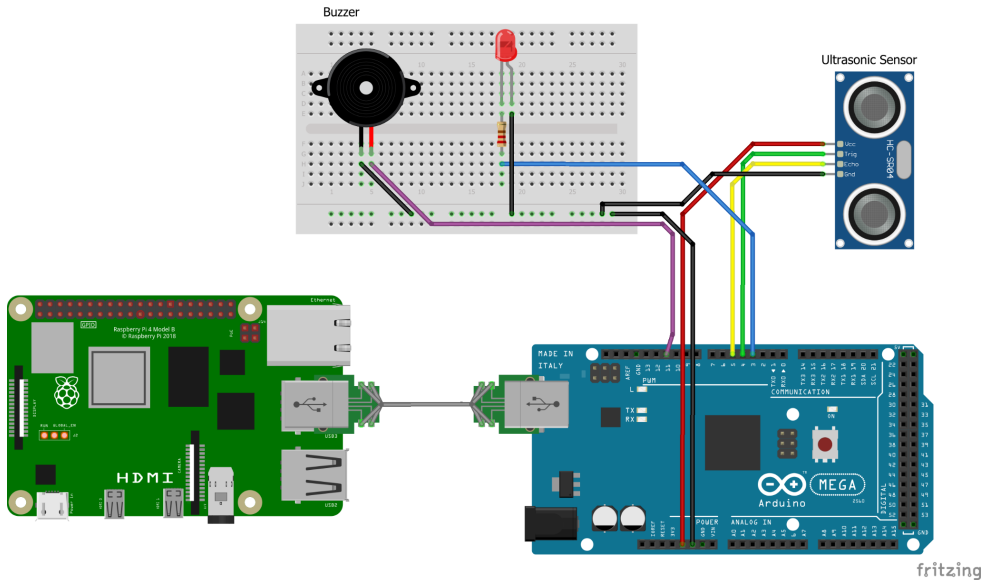


Figure 1: RP1 hardware configuration

Figure shows the current hardware setup of RP1 sub-system and all its auxiliary parts - an Arduino Mega 2560, ultrasonic sensor, buzzer and a LED. The assignment specifically asked us to connect the sensor directly to the Raspberry Pi, but in the end we opted to use an Arduino as a go-between.

Changes

The reasons for this were several: The sensor requires 5 volts to operate and it returns a pulse signal with the same voltage (Too much for Raspberry to handle). Although we had a solution for this, setting up a voltage divider, GPIO handling on the Raspberry proved much more challenging than on the Arduino, especially for C++.

Currently we can do simple tasks with the Raspberry GPIO pins, such as control LEDs and attach buttons (if we wish), but I spent too much time trying to figure out how to actually read and handle the sensor signal. I didn't find a compatible third party library to help, and at the time, didn't have enough experience with C++ timing functions and interrupts to figure out how to do this myself in a timely fashion.

Since I already had a working sensor code for the Arduino from initial tests, we decided to let the Arduino do the measurement instead. It was hard to give up on the Raspberry, but in the end we have a working solution that probably performs better than what I could have coded myself from the ground up.

Implementation

RP1 is connected to Arduino with a USB cable, and they communicate using rs232 standard serial communication.

The reason for this connection type is mostly practical. The Arduino requires its own power supply, and the cable provides this. The cable also protects the Raspberry from the Arduino's higher native voltage outputs. The parts would also always be in close proximity, so there was no need to implement wireless communication between them.

Once this connection was in place, it made little sense to connect other parts, such as buzzer and LED to the Raspberry, as that would only complicate the wiring and code, and make physical wiring mistakes more likely during assembly. The Arduino handles all that now.

The purpose for the buzzer is to signal to the user that a measurement is about to begin.

The purpose of the LED light is to show that measurements are being taken.

The Arduino takes 20 samples and calculates the average, rounded to the nearest whole number (in centimeters), and returns this value to RP1. RP1 waits for Arduino to finish, however long it takes.

RP2 contributions

Using what I had learned working on the RP1 and its GPIOs, I made some code modules for parts that we wanted connected to RP2. These modules were to be integrated with the existing user interface of RP2, and they include functional buttons and an LCD display.

Display: A small 16x2 LCD display connected via i2c. It can show 2 lines at a time, 16 characters long each. Its functionality is very limited at the moment. It can only show the last measurement taken, and not scroll through the database as originally planned. It can also not print any lines longer than 16 characters long. It uses an external library `lcd1602.h` that provides all the functions to write to the screen (just not enough of them).

Buttons: There are currently no buttons attached to RP2, but we have the working means to add some. We had originally planned to attach three buttons - one to initiate measurements, and two to navigate the display up and down the entries of the database.

Since the display lacks the latter capability, we have not implemented those buttons. The final button is made redundant by the user web interface, and is therefore also removed from the system.

The button code is interrupt based, and needs `wiringPi.h` to work. A button to initiate measurements has been tested along with the display, and it works - measurements were taken, and shown on the display. The LED shown in Figure 2 (page 13) was just “stand-in” for yet unfinished tasks. It would simply change state if a button push was registered, and is no longer part of the delivered system. Only the display remains.

Total system

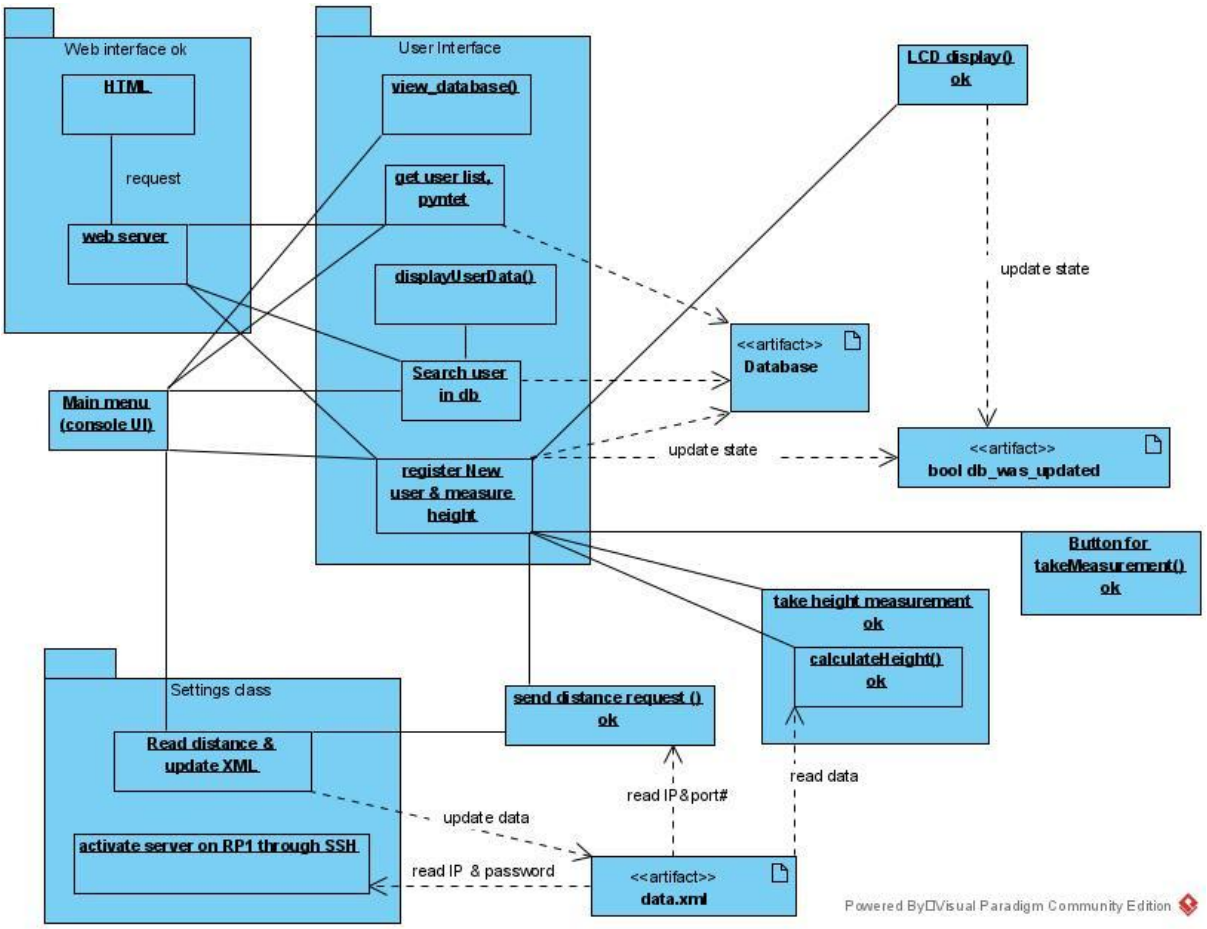


Diagram 7: RP2 package overview.

RP1: Physical

See Figure 1: RP1 hardware configuration

RP1: Code Diagram

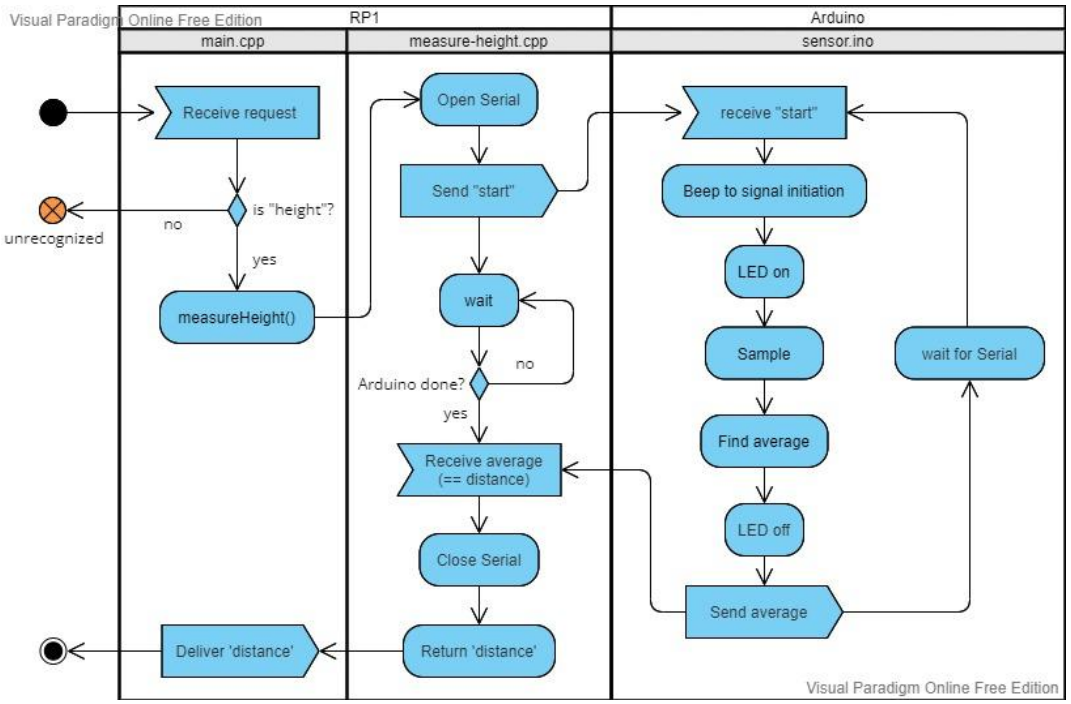


Diagram 8: Flow of operations (activity) on the RP1 sub-system

RP2: Physical

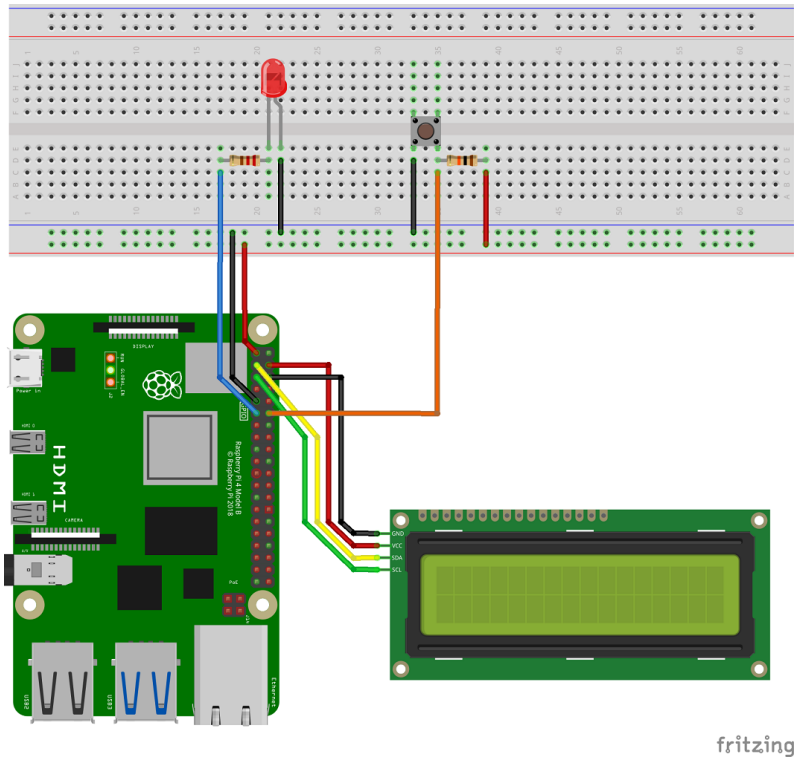


Figure 2: Showing RP2 and its parts as originally planned (Button and LED are now removed)

RP2: Code Diagram

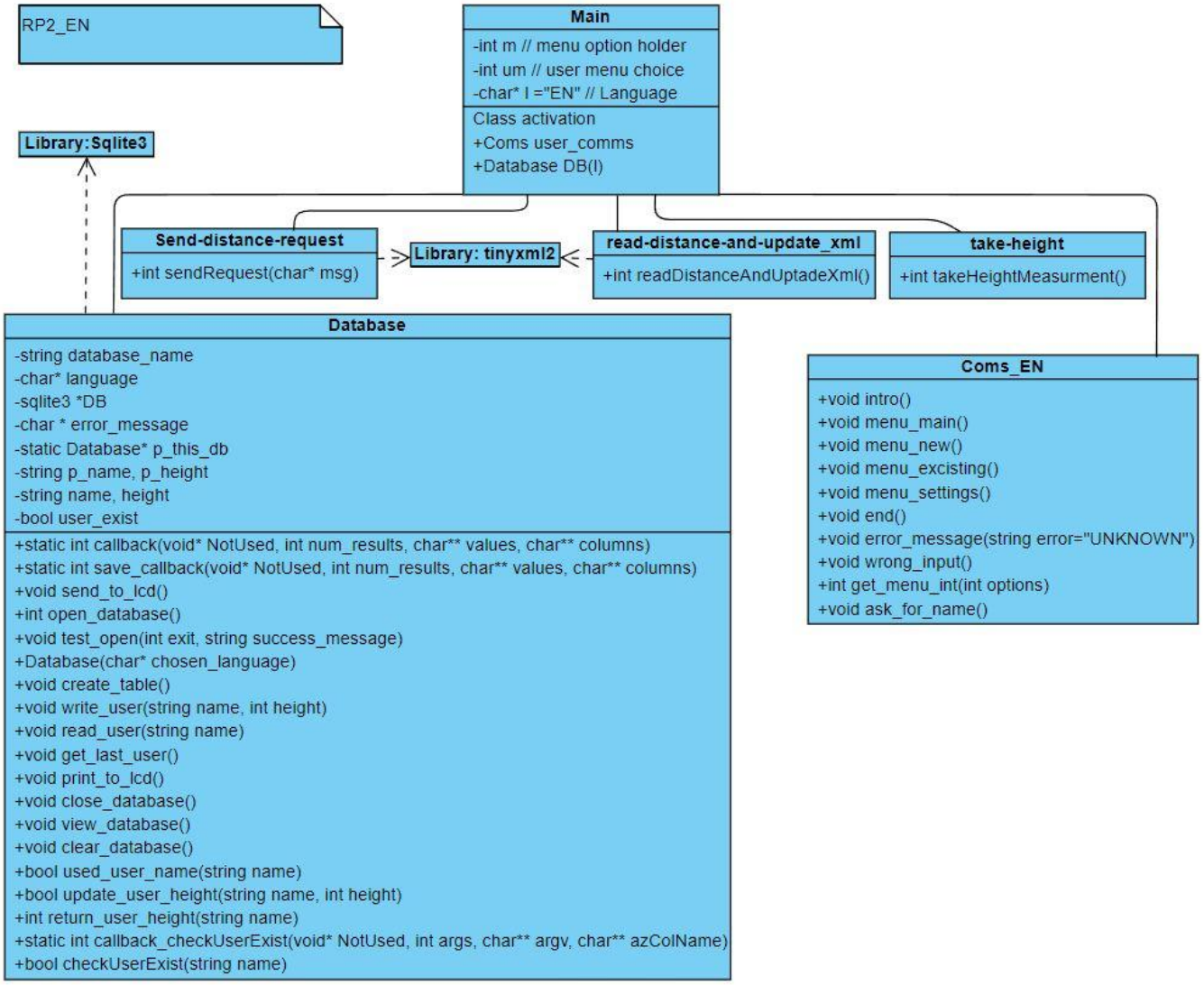


Diagram 9: Class diagram for RP2 with English language and console interface

Possible further development

- Fully integrate the mini-display along with navigation buttons, letting the user navigate through the menus and database, and look up previous measurements.
- Or, expand and further integrate the user web interface, providing it all the same above mentioned capabilities.
- Abandon the Arduino, connect everything to the Raspberry Pis.
- Wireless network communication between RP1 and RP2.
- Implement the use of the unique user ID or number along with name, to remove problems related to searching for one user by name.
- Control RP1 remotely (from RP2)
- Tidy up and condense functions related to database(database.h & .cpp)
- Database input sanitization
- External static IP or domain name for RP2 to control measurements from any device.
- Add port and IP reading from xml file. Currently port and IP are hard coded.
- Possibility to setup RP1 through SHH on RP2 from a web interface menu. Currently RP1 server `main.cpp` has to be activated directly on RP1.
- Test for mistakes and subsequently fix code to not allow unwanted operations.

Reflection on learned knowledge/course

We have all gained insight into these new concepts: Github, Linux and Arduino. Github is perhaps the odd one out, as it was not a requirement in all the tasks we could choose from. However, we are glad we had to learn it, as it has made sharing, saving and updating code a lot easier.This probably saved us time overall, during the project.

Johannes

The development process gave multiple challenges to overcome, multiple operating systems and, at times, limited access to physical hardware. There are different libraries depending on what operating system you use., showing the importance of not developing in the vacuum of one operating system. At times, the limited access to hardware made testing of functionality more difficult, having to make and build on functionality, without being able to properly test it. However, the use of github and branching made storing the untested code easier.
Sql and sqlite3 for database and their following functionality, is also knowledge I have gained. As previously mentioned, not having used either sql or databases before, made this more time-consuming than I may have wished for, but not wasted. I do see this as valuable knowledge and skills.
Accessing and changing variables from within static functions was a challenging and interesting problem to solve. I do feel I had to stretch myself beyond my knowledge to be able to solve it.

I would also like to add that working with a team to create a system larger than I would have made on my own, given the limited time frame, is something I have found motivating and enjoyable.

Alexandr

During this project new knowledge acquired:

- Linux, compilation in linux with g++, LAN setup, socket for linux.
- Version control with git and github
- Read/Write xml files with tinyxml2
- Web server mongoose
- Template engine inja
- Few other helpful libraries like: mjson, filesystem
- Never to do again this: $g++ -o main.cpp

Øystein

It's been a tough, but rewarding exercise to work on. I'm proud and impressed of what the group has achieved together (in practically just 4 weeks). I personally only wish I had come further with the hardware challenges. The learning curve was very steep at first, at times felt like butting heads against the wall, but then even more satisfying when something finally worked. My time has mostly been spent tinkering, reading, and not so much coding.

I soon discovered that the C++ community for Raspberry Pi is quite small, and that support libraries and tutorials on how to interface various gadgets are a bit lackluster and hard to find. I should probably have looked more into incorporating python code, to interface all our gadgets more easily, as the resources for that appears to be plentiful and more versatile.

Figures & diagrams

Used figures & diagrams text description immediately followed by the file name. In chronological order of appearance.
Unused files will be found in the folder described in Attachments.

Diagram 1: Overall System design
Deployment Diagram1.jpg
Diagram 2: Sequence (startup to submenu 2.2)
Sequence RP2 Main.cpp to sub menu 2.2 View database.JPG
Diagram 3: Activity diagram for readDistanceAndUpdateXml().
Read distance & update XML Sequence Diagram.JPG
Diagram 4: Activity diagram for takeHeightMeasurement().
take height measurement ok Sequence Diagram.jpg
Diagram 5: Use case for web UI.
Use Case Diagram1.jpg
Diagram 6: HTML menus and related URLs.
HTTP Activity Diagram.jpg
Figure 1: RP1 hardware configuration
RP1 System.png
Figure 2: Showing RP2 and its parts as originally planned
RP2_all_wiring.png
Diagram 7: RP2 package overview.
RP2 Deployment Diagram.jpg
Diagram 8: Flow of operations (activity) on the RP1 sub-system
RP1 activity diagram.jpg
Diagram 9: Class diagram for RP2 with English language and console interface
RP2_EN.JPG

Attachments

- Code in folders: RP1 and RP2
- Unused Code in folder: AddOns-FutureDev
- Documentation folder:
  - Wiring_diagrams folder: diagrams with hardware wiring
  - cyphy_diagrams folder: Diagrams and pictures of interface
  - requierment.txt: library versions