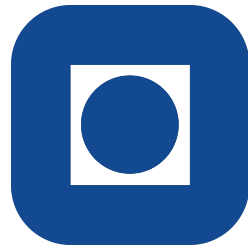


TFE4205

NeoKomp
Technical Report

Magne Hov, Sigurd Heide Rosland, Sigurd Hellesvik

May 19, 2019

Contents

1 Intro **1**
- 1.1 Problem owner . . . 1
- 1.2 General problem description . . . 1
- 1.3 Solution . . . 1
- 1.4 Overall Design Architecture . . . 2

2 Problem description: Magne Hov **4**
- 2.1 Graphical User Interface . . . 4
- 2.2 Input devices . . . 5
- 2.3 Bluetooth interface . . . 5

3 Solution: Magne Hov **5**
- 3.1 Graphical User Interface Tools . . . 5
- 3.2 Hub implementation . . . 6
  - 3.2.1 Guide Frame . . . 7
  - 3.2.2 Search Frame . . . 8
  - 3.2.3 Cart Frame . . . 9
  - 3.2.4 Balance . . . 10
  - 3.2.5 Receipt . . . 10
  - 3.2.6 Generic Frame Layout . . . 10
  - 3.2.7 Timers . . . 11
- 3.3 Input devices . . . 11
  - 3.3.1 Keyboard Capture . . . 11
  - 3.3.2 RFID Scanner . . . 11
- 3.4 Bluetooth Manager . . . 12

4 Problem description: Sigurd Hellesvik **13**
- 4.1 Database API . . . 13
- 4.2 Nordic BLE . . . 14

5 Solution: Sigurd Hellesvik **15**
- 5.1 Database and SQL . . . 15
- 5.2 How to set up test user for EDB development . . . 16
- 5.3 Application Programming Interface / API . . . 16
- 5.4 API programming in PHP . . . 17
- 5.5 Vending machine and fixes . . . 18
- 5.6 Python API interface . . . 18
- 5.7 Nordic Semiconductors: BLE Architecture . . . 21
- 5.8 Nordic Semiconductors: Programming . . . 22
- 5.9 Future features . . . 23

6 Problem description: Sigurd Heide Rosland: Shelf Module Hardware & Non-bluetooth firmware **23**
6.1 Microcontroller 23
6.2 Small display 23
6.3 Input devices 23
6.4 LED 24
6.5 PCB Antenna 24
6.6 Shelf Module Hardware Drivers and Firmware 24

7 Solution: Sigurd Heide Rosland: Shelf Module Hardware & Non-bluetooth firmware **24**
7.1 nRF52832 microcontroller 24
7.2 Switch Mode Converter 24
7.3 RGB Screen 25
7.4 Input Devices 26
7.4.1 Incremental Rotary Encoders 26
7.4.2 Push Button 26
7.4.3 LED 26
7.4.4 PCB Antenna 26
7.4.5 Passive Components 27
7.5 Second Revision & Future Improvements (post delivery) 27
7.6 User Interface Firmware 27
7.6.1 GPIO 27
7.6.2 Button 27
7.6.3 Encoders 27
7.6.4 LCD Display control 27

Appendices **28**

A Module Schematic **28**

1 Intro

This document is a technical report on the system, but some extra measures are taken to ensure it can also function effectively as a reference on how to use, maintain and further develop the project. It has seven sections, first an intro and then two sections per participant. One project description and one explanation of the solution. The git for the project is found at `https://git.omegav.no/ov/neokomp/`

1.1 Problem owner

Omega Verksted is an organisation at Gløshaugen. Any student can become a member, and thereby get access to a comprehensive workshop for electronics and other activities. Members can buy parts, solder, design PCBs and generally build prototypes from scratch. Omega Verksted also has equipment for mechanical work.

1.2 General problem description

Omega Verksted sells electrical components. There are a lot of different components, and they are bought from different suppliers. There are mainly two kinds of suppliers: the Western corporate and the cheap Asian. Western corporate are distributors like Farnell and Digikey. Asian suppliers are usually smaller traders found via Ebay or Aliexpress. The shipping time varies a lot, especially when ordering from Asia. When Omega Verksted receives the components, they are registered into a database. But because of the fact that suppliers sometimes change components and prices, and that Omega Verksted often change what components they sell. Marking all components with a scannable barcode is a lot of work, if not impossible. So the problem is: Omega Verksted has no way to register what components are sold, and thus no way of keeping a count of components. This creates a delay in the stock of components, which are not refilled until after they are detected empty, then bought and shipped to Gløshaugen. This might at worst take months. Additionally a set of plausible utility functionality has been identified, such as finding components in the shelves. These will be addressed by the design in addition to the main problem.

1.3 Solution

The components are already sorted at positions, which are indexed in sections, shelves, rows, columns and depth. This can be seen in fig. 2. Depth is into the plane, because each box can contain multiple different components. To solve the problem, we will make a module per section that allows registering the position of the component that the customer wants to buy. This module will send the component data to a shopping cart at a Point Of Sale, where the customer can buy the component from. This Point Of Sale will also get support for other methods of adding items to the shopping cart, which will

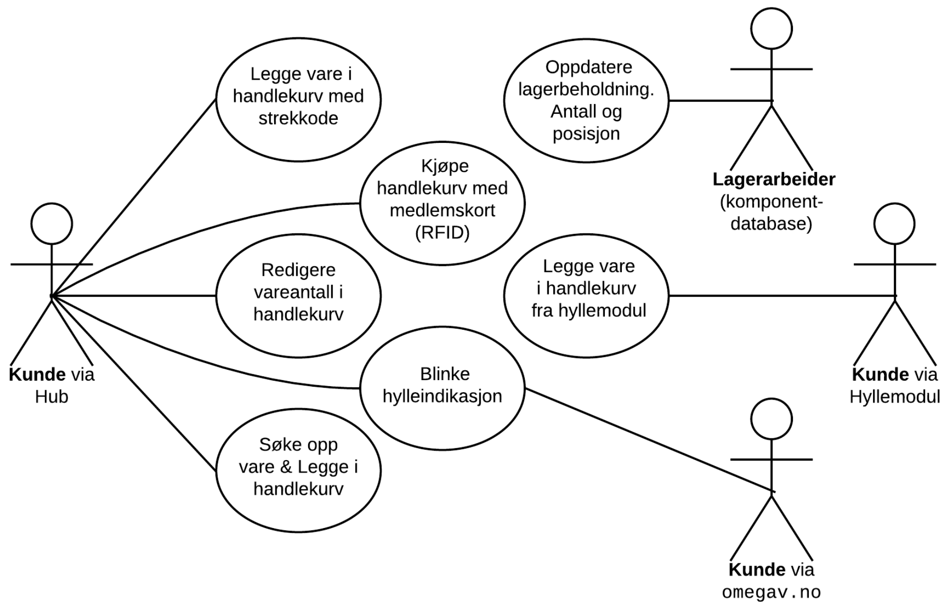


Figure 1: Use case diagram for the component store. Different users are illustrated as interacting with the system differently.

be detailed in a later section. The Point of Sale will interface with Omega Verksted's stock database. The intended use cases for different parts of the system are illustrated in Figure 8.

1.4 Overall Design Architecture

The solution to the problem is designed as a distributed system, as illustrated in Figure 3.

A central point of sale (the *hub*), will allow the customer to enter orders into a virtual shopping cart. The customer can then buy the components present in the shopping cart by authorizing the sale with his or her unique RFID chip. The orders can be entered in a number of ways. A bar code scanner allows the customer to add orders by scanning the physical bar code on the component or its packaging. A keyboard allows the customer to search the component database and add components to the shopping cart from search results. However, the primary input method of this solution is a number of PCB modules that will also allow the user to add items to the shopping cart by selecting component locations.

A number of modules connect to the hub via a wireless interface. The modules are distributed around the workshop. One module typically serves one section of components. A section is organised as a group of shelves, and each shelf is structured as a 3D grid of rows, columns and depth. The modules contain input devices that let the customer select a component at a specific row, column and depth. By pressing an *order button*, the selected component is issued to the central point of sale and inserted into the current shopping cart. The module will be able to display the current selected position on a display. The module also has a LED that can be requested to turn on for fast localisation of a section or shelf.

A component database already exists, and is used to query information about components. A

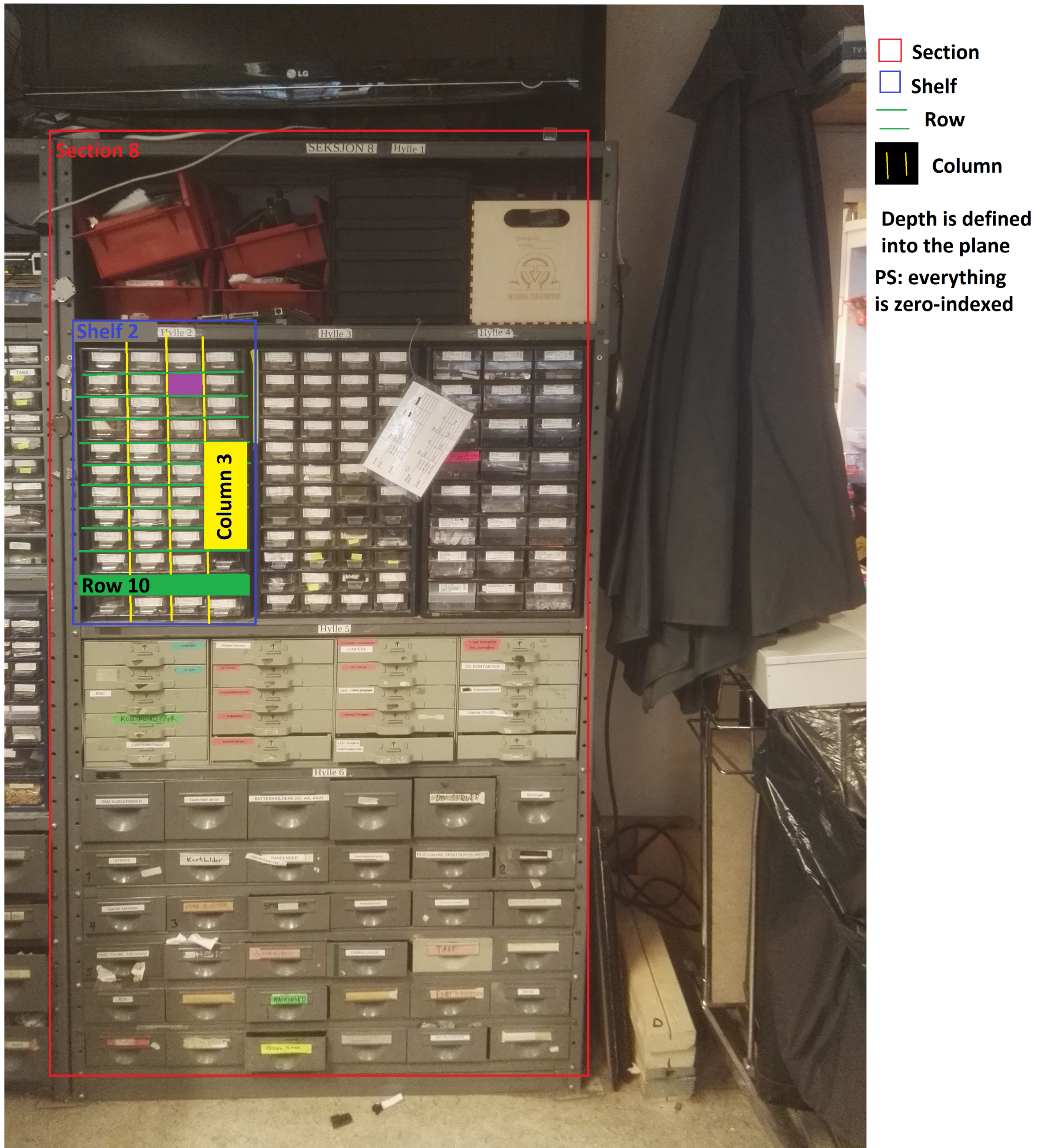


Figure 2: Picture and explanation of how the Omega Verksted Komp component sorting is set up.

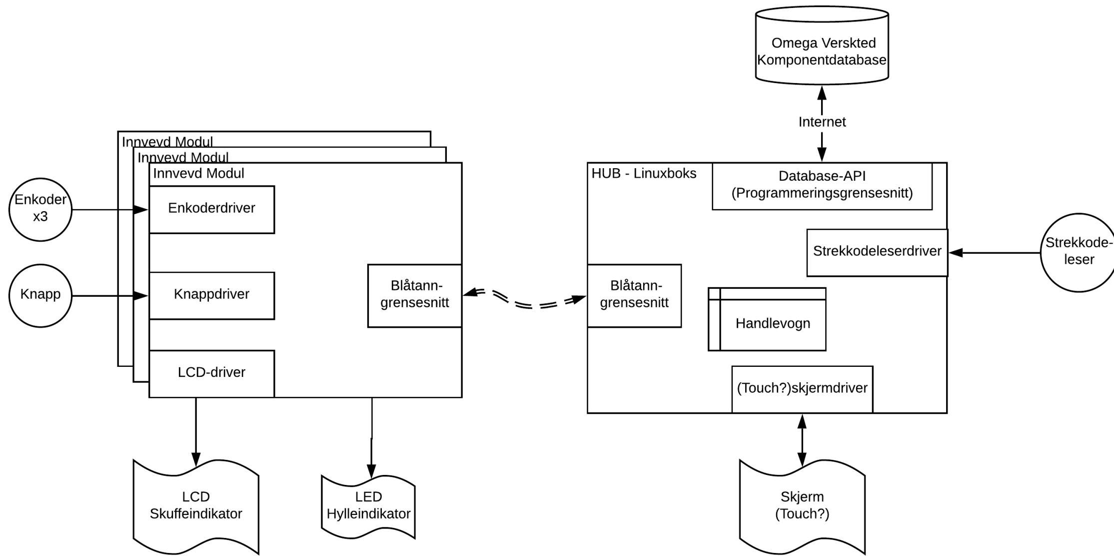


Figure 3: Architecture of complete system, with modules, hub and database. The modules are connected to the hub via a wireless interface.

search query will return components whose name match a string. A position query will return the details of the component that resides at that specific location. Components can also be bought by querying the database. All queries are handled via a dedicated API.

2 Problem description: Magne Hov

The Hub is a central actor in the overall system architecture and has multiple responsibilities. It receives input from customers to look up components, commands to alter the contents of a shopping cart, and to purchase the components in the shopping cart. It receives input from modules to insert components into the shopping cart, and it can issue commands to modules to turn on their LEDs. It will also makes use of the database API to interface the component database.

The hub will be implemented on a Raspberry Pi computer running the Linux operating system.

2.1 Graphical User Interface

The hub will display information to the customer. The information will be displayed on a regular computer monitor, which will get a dedicated position in the workshop. The monitor must display currently selected components as they appear in a shopping cart, in such a way that the customer is aware of what components he or she is about to purchase. If there is enough time to develop, there should also be developed some method of displaying search results for a query string that the customer can enter. The search results should be able to be selected, such that they can be inserted into the shopping cart.

As the Hub acts as a point of sale, it should also display information about the total sum that a

shopping cart will cost. A customer might want to check the balance on their account, which must be possible to do via the Hub.

The graphical user interface is planned to be implemented as a state machine, with each state representing a different graphical window. For example, the shopping cart and search results can be displayed as two different panes that are active when dealing with the appropriate functionality. The various forms of input the the Hub will then do different things depending on which state the machine is in. The GUI will be implemented in Python.

2.2 Input devices

The search function, various select and deselect functions of the shopping cart require an input device in order for the customer to operate them. A standard PC keyboard should cover this functionality. Additionally, a bar code scanner should allow the user to insert items into the cart by scanning the bar code on a component. A RFID scanner must let the user check his or her balance, as well a authorize the purchase of a cart.

2.3 Bluetooth interface

The distributed modules will communicate with the Hub over *Bluetooth Low Energy*. A specific network topology model will have to be chosen. A broadcasting topology might prove to be an easier solution when dealing with many individual nodes. A connection-oriented topology might provide more reliable and safer communication, but a the cost of a higher complexity related to establishing and maintaining connections.

Regardless of which topology is chosen, commands must be exchanged between the Hub and the modules. The hub must be able to receive commands to insert specific components into the shopping cart from the modules. If there is time, the hub should be able to distribute commands to turn on LEDs.

The raspberry pi hardware has an integrated Bluetooth controller which will be used for all Bluetooth communication.

3 Solution: Magne Hov

The hub was implemented on a Raspberry Pi 3 B+. This computer has a reasonable amount of processing power, while also providing a Bluetooth interface out of the box. Additionally, it has Ethernet connectivity, and it has multiple USB ports for connecting keyboard, mouse, RFID reader and bar code scanner.

3.1 Graphical User Interface Tools

This section describes the implementation of the graphical application, as well as tools used to make it and decisions made on the way.

It was early on decided to implement the graphical user interface in Python. The decision is based on the fact that the developer has previous experience with Python. The choice of using Python also proved to be a good one because several other software libraries that were needed for the project were readily available in Python, and their ease of use greatly reduced the required development efforts.

Initially, the TkInter library was used to construct the graphical interface. This graphical library is a high level graphics library which lets the user construct layouts by combining widgets from a collection of graphical components. This allows for very fast prototyping, because the structure and low level implementation of different widgets do not need to be created. A code skeleton was created for the state machine describe in the subsection 3.2.

While implementing the search functionality (which is described in subsection 3.2), TkInter was found to be lacking in support for multi-threaded applications. The search functionality, which depended on the API described in subsection 5.3, proved to express blocking behaviour. This caused the whole application, with its graphical elements and all, to freeze until the API had return with results. When doing searches that returned a large amount of results, the degradation of the user experience was significant; the user had to wait for tens of seconds before the application returned to a responsive state.

Based on the problems regarding multi-threading and GUI responsiveness, a decision was made to switch to the Qt Framework for all GUI needs. The Qt Framework is originally written in and for C++, but luckily there exists a good wrapper library for Python. PyQt5 proved to supply a much richer set of features than TkInter. In the remainder of the report, PyQt5 will be referred to as PyQt. In addition to providing a richer library of interactive and graphical widgets, Qt (and thus also PyQt) also has better support for multi-threading.

The source of the frozen GUI problem was found to be the main event loop of TkInter being blocked by long API calls. The solution to the problem was PyQt's inherent support for Threading. PyQt implements a thread class `QtCore.QThread`, which together with the concepts of *Signals* and *Slots* are able to efficiently and safely communicate with the main event loop of PyQt.

In Qt and PyQt, *signals* and *slots* are key concepts, which connect various objects together. Put shortly, a signal is an event that can contain arbitrary data. The signal can be emitted manually by function calls, or by one of the many predefined signals in the library of Qt Widgets. Examples of predefined signal emitters are button widgets getting pressed, elements in a table being selected, and keyboard press events. A signal can be connected to a slot, which will receive the signal and can then call a function with the data. The slot must be associated with an object derived from the base class `QObject`, such as a frame, interactive widget or QThread. The communication of signals to connected slots is handled by the main event loop, such that they can be interleaved with the internal procedures that are required to update the visual elements of the GUI.

All API calls that were found to take a long time to respond were offloaded to Querier Threads. These threads were constructed to emit signals containing the retrieved query information. The appropriate custom widget then consumes the signal via a connected slot method. Since the signal is only emitted when the querier thread has completely retrieved the information, the widget is free to update its visuals and run other procedures at the same time. When the results are ready, the signal is delivered asynchronously via the main event loop.

3.2 Hub implementation

A state machine was designed to capture the necessary graphical frames and the actions that would encode the transitions between them. The state machine is illustrated in Figure 4. The following paragraphs will explain how the Hub is intended to work, with descriptions of each state.

The Hub code is located in `https://git.omegav.no/ov/neokomp/tree/master/Software-hub`. Each visual frame is implemented as a super class of a `QtWidgets.QWidget`. Then each frame is

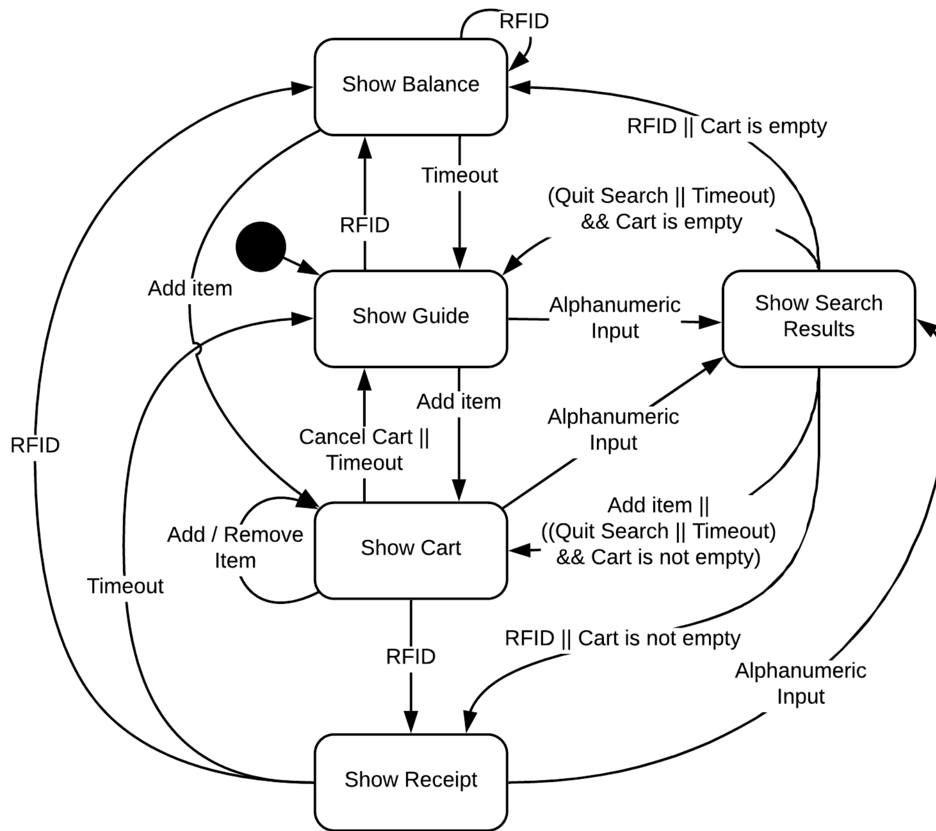


Figure 4: State machine illustrating the graphical states of the Hub and the transitions between them.

inserted into a top level `QtWidgets.QStackedWidget`, which allows one of multiple stacked widgets to be displayed. A frame transition can then be performed by setting the currently active widget of the stack.

3.2.1 Guide Frame

The default state of the hub is to show a screen with a helpful *Guide* text that will tell the customer how to use the Hub. A snapshot of the visual frame is shown in Figure 5. The text includes a description of how to start searching for components, how to operate the distributed modules that are placed around the store, how to purchase the items in the cart, and how to check the customers balance. If the Hub is left inactive for a long time, it will always return to the *Guide* state.

As from any frame, the user is free to give the Hub any kind of input. If the user starts writing, the machine will promptly switch to the Search frame, where the entered text will be inserted into a search field. If the user activates the RFID-reader with his or her RFID card, the machine will switch to the Balance Frame. If the user scans a bar code, the item will be added to the cart, and the Cart Frame will be displayed. If the user operates one of the distributed modules, an item will also be added to the cart before the Cart Frame is displayed.

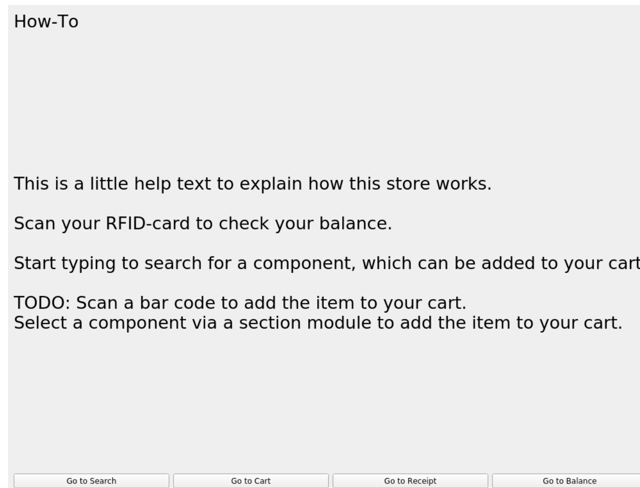


Figure 5: Guide frame, as one of the operational states of the hub.

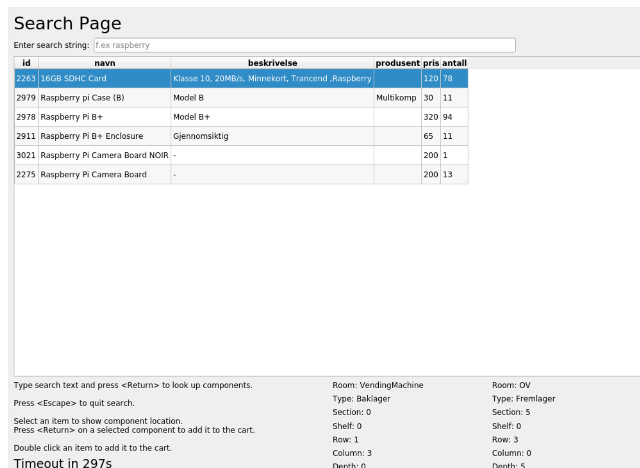


Figure 6: Search frame, as one of the operational states of the hub.

3.2.2 Search Frame

The Search Frame allows the user to search for items in the shop. A snapshot of the search frame is shown in Figure 6. The search frame is made active and displayed when the user enters alphanumeric characters by typing on the hub keyboard. A text entry widget (`QtWidgets.QLineEdit`) is then focused, and the typed text is inserted into it. The alphanumeric input can be provided from any frame, and will result in a transition to the search frame. If enter is pressed while the search entry widget is focused, the search text will be queried via the API (detailed in in subsection 5.6).

When a search query is issued, a new SearchQuerier object is created. This class derives from `QtCore.QThread`, and will therefore run concurrently with the main event loop. The search object will initially query the search text to acquire a list of item IDs that match the search text. Then the search object will request item information for each of the acquired item IDs. When a search text matches a large amount of items, it can take a considerable amount of time to retrieve the full search result. As soon as information about a single item has been retrieved, it is propagated to the search

Figure 7: Cart frame, as one of the operational states of the hub.

frame widget as a custom signal. Each item is stored in a native Python dictionary, with fields for each item parameter.

The search result signals are picked up by the main event loop, and are eventually delivered to the connected slot in the search frame widget. When the search frame widget receives item data, it adds the item to the search results. By offloading the time expensive part of the search task to its own thread, the GUI stays responsive.

The search results are displayed as a table. The table is implemented as a `QtWidgets.QTableView`, and each row corresponds to an item. The columns contain item parameters such as item id, item name, item description, item availability and item price. The item id column also stores a reference to a dictionary containing the full item data. The full item data contains additional fields that are useful when displaying item positions and adding an item to the cart.

Items in the search results table can be selected, either by using a mouse or the arrow keys on the keyboard. When an item is selected, information about the position of the item is displayed in the lower right corner of the search frame, as illustrated in Figure 6. This information is retrieved from the items dictionary which is stored in the first column in the table. If the item is stocked in multiple positions, all of them will be shown.

If a selected item is double clicked, or if the *Return* key is pressed while an item is selected, the selected item is transferred to the cart, and the cart frame is focused.

3.2.3 Cart Frame

The Cart Frame shows which items are currently in the shopping cart. A snapshot of the cart frame is shown in Figure 7. A cart is internally stored as a `QtWidgets.QTableView` object, where each row in the table corresponds to an item. The columns contain information like item id, item name, item description, item count and item price. Additionally, a hidden dictionary object with the full item data is stored in the item id column. This hidden object contains all item parameters as they were provided by the API, and is used f.ex when the item is added to the cart.

The table widget is displayed in the cart frame widget. The items in the table can be selected, and by entering a Return key event, the selected item can removed from the cart. By pressing Escape,

one can clear the cart and go back to the guide frame. Additionally, one can also start typing to start a new search for items.

The usual use case from the cart screen is to present a RFID card to finalise a purchase. At this point, the cart aggregates the items into a list, and sends them to receipt frame, which will attempt to carry out the purchase. The cart is cleared, and the receipt frame is made active.

3.2.4 Balance

The Balance Frame is presented each time the user presents his or her RFID card while the cart is empty. The unique RFID number is used to query the central database through the API, from which an effective balance will be returned. This balance is displayed in the frame together with the username associated with the unique RFID number.

After a certain duration has elapsed, the guide frame will again be made active. If an item is added before the timeout, then a transition to the cart frame is performed instead.

3.2.5 Receipt

The Receipt frame is similar to the balance frame in appearance, but performs more internal procedures than the balance frame.

When purchasing the contents of a cart, the receipt frame will receive a list of items to be purchased, as well as an RFID number. The first thing the receipt frame does is to query the current balance associated with the RFID number. If the current balance is less than the total sum of the items being purchased, an error is raised on the screen and the purchase is dropped. If there is sufficient balance associated with the RFID number, then each item will be purchased, one at a time. The result of each item purchase is registered, and if any of the items fail, an additional warning is raised on the screen. If all items were successfully purchased, then no warning is raised. As long as at least one item was purchased successfully, the total sum of purchased items is displayed along with the new current balance associated with the RFID number.

After a certain time has passed, the guide frame is again made active. If new items are added to the cart, or a search string is being inserted, a transition to the appropriate frame will happen instead of returning to the guide frame on a timeout.

3.2.6 Generic Frame Layout

All the frames use the same general frame layout. Therefor, a generic frame class `GenericFrame` was implemented to provide a common top level layout. Each graphical frame is made to inherit from `QtWidgets.QWidget`. The `GenericFrame` is implemented to also inherit from `GenericFrame` in order to help resolution of super classes.

The generic frame installs a vertical layout (`QtWidgets.QVBoxLayout`) as the top level layout. Additionally, it creates a virtual focus method which is called by the top level Qt Application object when a new frame is to be made active. Only a few of the frames have specific methods that the need to run when made active (or focused). The `focus` method is therefore implemented as an empty method, such that those frames that need specific focus actions can re-implement it. The generic frame also installs an empty title label, which can have its text specified by the derived frames.

3.2.7 Timers

Several of the states require a timer object to keep track of a timeout event. The cart, search, receipt and balance frames are all dependant on a timeout to go back to the guide frame when there is a longer period of inactivity. Because four of the five frame classes require this functionality, a `TimedFrame` class was implemented with this functionality. The frames that require the timers inherit this class. The `TimedFrame` inherits from `QtWidgets.QWidget` to help in super class method resolution.

3.3 Input devices

3.3.1 Keyboard Capture

The Hub has a standard keyboard connected to it. By default, the top level Qt application will capture all keyboard events that are passed along to it from the operating system.

Each keyboard press results in a Qt event. These key events are normally passed along to which ever widget is currently in focus. When a widget receives a key event it will check its configuration and programming to check whether it should do something with it, f.ex the `QtWidgets.QLineEdit` widget that is used in the search frame will accept all kinds of keyboard presses to insert characters into the search field.

For most widgets, the standard actions for key events are fine, but there are certain key events that need to be mapped to specific transitions. By referring back to the state machine that describes the behaviour of the hub (Figure 4), we see that certain keyboard events should produce transitions between frames. In order to catch these keyboard events, we make use of an event filtering concept that is provided by Qt. Each widget can install a custom event filter. The custom event filter is implemented as a custom function that is run before each event is passed along to the widget for standard processing.

In order to install the event filter, one must inherit from one of the widget classes. Each of the grapihcal frames are derived from `QtWidgets.QQidget`. All other widgets are inserted into these, meaning that there is always one of these frames in focus or sub-focus. By placing the event filter on the frame widget, the events can be intercepted before they are propagated down to the widgets that make up the frame.

Every frame is installed with a filter that switches to the search frame if alphanumeric input is given. Every frame other than the guide frame will intercept Escape keys and produce a transition to the guide frame, while also emptying all carts and search results. In the search frame, a Return key event is intercepted to mean different things depending on whether a search result is selected or the search text box is selected. If the search text box is selected, then a new search query is started (or the results table cleared if search text is empty). If a search result is selected, then that item is inserted into the cart, and the cart frame is put into focus. In the cart frame, the Return key event is also intercepted to remove currently selected items.

3.3.2 RFID Scanner

The hub has an RFID scanner connected to it. An RFID scanner is able to receive an identity by monitoring the electromagnetic fields around the scanner. An RFID card or tag that is held in proximity of the scanner will be activated by the scanner and will transfer a unique identifier number to the scanner.

The RFID scanner presents itself as an USB device to the operating system, and the operating system receives the identification number as a sequence of keyboard input events. The identification number will normally be inserted into whichever window is currently in focus, from the perspective of the operating system. This is undesirable for the identification numbers, which should be considered as special input, and not the same as f.ex a search string.

The `evdev` python package was utilised to capture all input from only the RFID scanner. A `rfid_driver.py` module was implemented to initialise the capturing of the events, and to convert the serial sequence of key events into an RFID string. The string containing the RFID number is attached as custom data to a `Qt` signal. The signal is connected by the top level NeoKomp module, which accepts the RFID number and propagates it to the appropriate frame.

The RFID number is initially delivered to the cart frame, which will check whether its cart is empty. If the cart is empty, it will relay the RFID number to the balance frame, which will query the user's balance. If the cart is not empty, the RFID number and cart contents are delivered to the receipt frame, which will attempt to carry out the purchase.

3.4 Bluetooth Manager

The raspberry pi computer is equipped with a BCM43438, a combined IEEE 802.11 (Wi-Fi) and IEEE 802.15.1 (Bluetooth) chip. The Bluetooth interface is used to interface with the Bluetooth interface on the shelf modules.

The shelf module, which is implemented with a Nordic nRF52 chip operates with the *Bluetooth Low Energy* protocol family. There are several topology available for *BLE*, but the one we landed on was a Central - Peripheral pattern. This pattern is similar to a server client pattern. For more general information about Bluetooth and *BLE*, see subsection 5.7.

The clients and servers offer services as defined by the *Generic Attribute Profile* (GATT) specification. Services are collections of conceptually related characteristics. Each service can have one or more characteristics. Each characteristic represents some type of information that can be queried by a client, such as a temperature, or shelf position.

Take the use case of adding an item to the cart from a shelf module as an example. The module acts as a peripheral, or server, by offering a service that contains a characteristic for the module's current position. When a user turns the dials on the module PCB, the internally stored representation of the position is changed. The data associated with the GATT characteristic is then changed appropriately, along with the LCD display showing the new current position. When the customer is happy with the choice of the position, the button on the module PCB is pressed. When this happens, the device turns on its BLE advertising feature. This makes the device visible to other Bluetooth devices. Now that the device is visible, and has a valid position stored in its characteristic, a central client is able to query the state of the characteristic, and in that way retrieve the order from the module.

The Bluetooth driver on the hub is implemented as a central client. The Bluetooth manager is made as an extension to `QtCore.QThread`, so that it can run in its own thread while still being able to produce *Qt* signals with data.

The manager uses the python `gatt` library (acquired via `python pip`). This library implements a class for representing a device manager, and a class for representing connected devices. The `gatt.DeviceManager` class connects to the computer's Bluetooth adapter via `D-bus`, and is able to f.ex scan for devices. The hub extends this class to do filtering on discovered devices. The manager continuously scans for advertisements sent from devices. A device list is kept in a local file

`device_mapping.txt`. The list contains the Medium-Access-Code (MAC) address of each module, along with a section number which that module serves. When the manager detects Bluetooth advertisements from an authorized device, it attempts to connect to it.

If a connection to an authorized device is successful, a `gatt.Device` instance is created. The device class automatically queries the device for information about which services and characteristics are available. The hub extends this class by re-implementing the method that is run when the all services and characteristics have been discovered. The specific characteristic which stores the position will now be queried by the hub. If the characteristic contains a valid item position, a custom signal will be emitted. The signal is connected via the top level `NeoKomp` class, which will pass the item position down to the cart.

When the hub has connected to, and read the value of the position's characteristic, it will promptly disconnect from the device. The disconnect event is interpreted as a successful delivery of the item by the shelf module. Since the item was delivered successfully, the device will now disable Bluetooth advertising, which prevents the hub from connecting and reading the characteristic more than once. After disconnecting from the device, the BLE manager goes back to scanning for other modules.

At any point, the position of the shelf module can be changed by operating the dials on its PCB. However, it is only when the button is pressed that the module will turn on advertising, allowing the hub to connect to it.

The ability to request a module's LED to turn on is a future feature. It would require the shelf modules to continuously scan for the hub's advertising packets. The advertisement packets could then encode the command for the module to turn on the LED, or the command could encoded into a characteristic on the Hub.

4 Problem description: Sigurd Hellesvik

4.1 Database API

As the Hub will need a way to search for components, register sales and do more things that interact with the databases of Omega Verksted. The functions requested from Omega Verksted is:

- To be able to search for a component by name, and get all info about this component from the database. This is info like component id, position, name, etc.
- To be able to search for a component by position.
- Authenticate a client using an key (secret string).
- Search for a username by rfid.
- Check saldo on users account.
- Buy a component for a user, by checking saldo, register the sale and get money from the buyers account.

The functions also be seen in: fig. 8. Currently Omega Verksted mainly interacts directly with the database. This is bad practice, since it requires the developer who want to make a function which interacts with the database to know the SQL prorgamming language. The better way to do this is to use an API(Application Programming Interface). In a nutshell, this is finished functions a developer

Figure 8: Use case for function the API should have. The blue one is a optional function, and will be implemented if there is time.

can interact with using a simple interface. Most programming languages have support for interacting with such interfaces, so the developer can use it for whatever he/she want. One project Omega Verksted already have use such an API, the vending machine project. But this API is spesifically designed for only the vending machine, even though it uses the database for components and the database for members. My job is to migrate this API to an API which covers the whole database of components. In adition, I will make functions in python which use this API, for the Hub.

4.2 Nordic BLE

The Modules will use a nRF52832 from Nordic Semiconductors. Sigurd Heide Rosland will program the functionality of this. But it needs to communicate with the Hub. This is done using Bluetooth, which is also why we chose this chip. My job will be to make the functions which use bluetooth to communicate with the Hub. To do this, I will first have to cooperate with Magne Hov to figure out what kind of bluetooth communication we want to use.

5 Solution: Sigurd Hellesvik

My part will be written as a tutorial, where the reader should be able to follow me trough all the steps I took when doing my part of the project. Thus it should be easier to maintain the project. But it should also be possible to find a theme and use it for your own inventions, as this project like most other projects, is only made up of multiple general concepts.

5.1 Database and SQL

Omega Verksted has most of its components registered in a database. This makes it easy to get information about components, add new ones and register sales of components. This database uses the Structured Query Language (SQL), one of the most commonly used database management languages. Database entries are sorted into categories and tables. OV has two different ways to interact with the database. But before you try any of these, be aware that you have to very careful when interacting with the database, as accidental deletion of data is bad and will result in a lot of extra work. It is recommended to set up a test user if one are going to make siginificant changes. Please read the section 5.2 to see how that may be done. First, we have a graphical interface on a webpage: `https://www.omegav.no/phppgadmin/`. To get the login here, you need to ask a board member of Omega Verksted. OV has many different subcategories of data in its database, like files for altium, info for our webpage(extern) and data we use for normal operation(intern). These can be seen listed on the phpPgAdmin webpage. To access the different tables of data found here, we need to navigate trough sub menus of the sub categories, to Schemas→public→Tables. But in this graphical interface we can mostly edit the information manually. But because we want to do it automatically, we will need to access the database in another way. Therefore we don't need to know very much about this webpage for the project, but it is useful to know that it exist. The second way to interact with the database is by directly using the SQL language, if I assume that the reader have basic knowledge of programming. First we need to be logged into somewhere we can access the database, for example the user described in section 5.2. Omega Verksted use PostgreSQL, thus we will have to use the command "psql" to start interfacing the database. But we need to specify some options before we get in. We need to set the host to localhost, set username and say what database we will use. See "psql –help" for all commands. For example:

```
psql -h localhost -d intern_dev_slegge -U slegge
```

Now we can use SQL commands. The ones relevant for us are SELECT, UPDATE, INSERT INTO, BEGIN, ROLLBACK and COMMIT. SELECT returns the data we want to extract from the database. We need to specify what table we using the FROM keyword. If we want to returned all data, use "*" instead of an identifier. If we want to parse the data returned, we can use WHERE to specify what elements we want our data to contain. AND and OR can also be used with WHERE to combine requirements. Remember to use semicolon.

```sql
--return all IDs in the component database
SELECT id FROM komp_komponenter;
--return all information about the components, as a table
SELECT * FROM komp_komponenter;
```

```sql
--return all IDs in the component database that cost 20
SELECT id FROM komp_komponenter WHERE pris = 20;
--return all names of components that cost 20 and is updated by 1337
SELECT navn FROM komp_komponenter WHERE pris = 20 AND oppdatert_av = 1337;
```

UPDATE changes the database, so before we go trough that, lets have a look at BEGIN. BEGIN sets a checkpoint in the database. Now no changes you do will go live before you use the COMMIT command. If you do anything wrong, for example delete every item in the database, you can simply use ROLLBACK, and everything will go back to as it where when you did BEGIN. Always use BEGIN+COMMIT/ROLLBACK when changing data in the database.

UPDATE first selects what table name we want to edit. Then we use the SET keyword to choose how we want to edit the columns. The WHERE command also works here.

Lastly we have the INSERT INTO command, which adds an element to a table. First we select the tables and the columns we want to insert information into, and then we use the VALUE keyword to choose what values we want to insert. Columns which are not specified will get default values. Now we know enough to be able to interact with the database.

```sql
BEGIN; --set checkpoint
--Set the price of all components to 1000.
UPDATE komp_komponenter SET pris = 1000;
--Set the price of all components which cost 20 to 1000.
UPDATE komp_komponenter SET pris = 1000 WHERE pris = 20;
--Insert a component with id 1234 and price 20.
INSERT INTO komp_komponenter (id, pris) VALUES (1234, 20);
ROLLBACK; --undo changes
```

5.2 How to set up test user for EDB development

The database of Omega Verksted have a whitelist, such that only clients with specified IP addresses can use it. So we have to use username@omegav.no if we want to interface the database directly. When implementing new features to a database, it is good practice to first use a test database. Omega Verksted already has multiple such test databases. For example intern_dev_slegge and extern_dev_slegge. Ask someone who is part of EDB, the IT department of Omega Verksted, to set up a test user for you to be able to access the database. And then ask them again when you have your finished code, to get it set up live.

5.3 Application Programming Interface / API

As seen over, it is not straight forward to use the database. But when a developer need to use the database, it is most often just for searching trough it or to update it in some way. Like we do in this project, we want to search for components and to buy them. And to do this it should be sufficient to only use a simple function. To accomplish this, we use an Application Programming Interface(API). Wikipedia defines an API as: "In computer programming, an application programming interface (API) is a set of subroutine definitions, communication protocols, and tools for building software. In general terms, it is a set of clearly defined methods of communication among various components.".

I understood it a lot better when someone described it to me as a waiter. When you order food at a restaurant, you do not order it directly from the chef. Instead you give your order to the waiter, and then she brings it to the chef for you. And then your food is sent back from the chef to you trough the waiter. To access the API, OV have a web link that can be polled. So users of the API need only the link and a key to use it. Omega Verksted already had an API for the component database, but it was not complete when we started on our project. I will explain how I fixed it shortly. But as it is written in PHP, I will first go through the basics you need to know about PHP to make an API.

5.4 API programming in PHP

The code for the finished API can be found at `https://git.omegav.no/ov/neokomp/tree/master/api`, and it might be useful to look at when reading the following sections. This is just a copy of the live API which is to find on the servers of OV, at omegav@omegav.no. As before, I assume that the reader have basic skills in programming. SQL use $ in front of all variables. To edit the database, PHP use pg_query(). Although, it is not safe to use pg_query directly, as a very simple hacking technique is SQL-injections. To avoid this, we combine an array with the variables we want to get or set in the database. To do this, we use pg_query_params(). For example:

```php
<?php
    $query = "SELECT * FROM komp_komponenter WHERE id = $1";
    $params = array($id);
    $row = pg_query_params($query, $params);
?>
```

Now with the SQL we know and the code above, we should be able to use the database from our SQL functions. Next, we want users to be able to access the API. As mentioned, this will be done trough a web link. The PHP file with our code will be run when people try to reach this link. The link to the new API for components will be `https://omegav.no/api/komp.php`. To achieve this, the file will be saved at omegav@omegav.no at the path "∼ www/omegav.no/api/komp.php". For the PHP script to understand which variables we pass into it, we need to send a specialized request, using HTTP, to this URL. There is functionality in most programming languages to do this. I will explain further how I did it in section 5.6. When our PHP script get this information, it will either be in the of a HTTP method, in our case either $_GET $_POST, where the get method ask for data from the API, while the post method send data to the API. For simplicity's sake, our API will use $_REQUEST, which is a combination of those two. Now we check if the request contains data by using isset(), and return an empty string if not. Then we get the line to retrieve code from the API user like this:

```php
<?php
    $handling = isset($_REQUEST['handling']) ? $_REQUEST['handling'] : "";
?>
```

Now we need to return data to the user. This can be done by simply using "print". But we want to return all data in the same manner, to make it easier to handle by the user. Therefore we format the return to JavaScript Object Notation (JSON). This is a popular standard used for

formatting messages. Most programming languages have functionality for parsing JSON data, which again makes it easier for developers to use our API. For our part, PHP has the function json_encode() which is all we need to turn encode our data to the JSON standard. To be nice to the user, we can set the HTTP header to JSON by doing: header('Content-type: application/json'); Now we can return data to the user like this:

```php
<?php
    print json_encode($res);
    header('Content-type: application/json');
?>
```

5.5 Vending machine and fixes

Omega Verksted has a vending machine for sale of electrical components in "Coopen". This already has an API for interacting with the SQL database. Yet this API have too narrow functionality, as it is designed only to work with the vending machine. Members have profiles at Omega Verksted, where they put money on a kind of accounts we have for them. They can then use those money to buy components. The only thing they need to buy those is an RFID card registered under their account. This trust based system works fine when buying at the workshop itself, because there is always a board member at the location when open. But for buying components from the vending machine, OV wanted a more secure solution. This was implemented by requiring a PIN code from the buyer, to ensure their identity. Since the new component sale system will be stationed at the workshop, we do not need the pin for all sales through the API. The only sales we need to verify is the ones going through the vending machine. So I added an check for sale from vending machine around the pin verification in komp.php kjop() function.d. Furthermore the vending machine only have columns and rows. Thus, the API only had input for those two, while the component database have: Room, Section, Shelf, Row, Column and Depth. So I implemented inputs for the rest into the API. To make the API easier to use, I added documentation on Omega Versted's wiki. But since the API are an internal matter, I cannot link it. But I have added screenshots of it which can be seen in fig. 9 and fig. 10 to the rapport of this instead.

5.6 Python API interface

I made functions in Python to use the API. This was done both to make life easier for Magne, and since I needed the functions to test the API anyway. These can be found at `https://git.omegav.no/ov/neokomp/blob/master/Software-hub/api_communication.py`. The Python package "urllib" is used to handle the communication. The usage of this is pretty straight forward: First we use urllib.parse.urlencode and encode() to format a dictionary to a form the API can read. Here we need to know the names of the variables in the API, and input them like {'handling' : 'funksjon_1'}. Then we use urllib.request.Request to send data to the API and get data back at the same time. Remember from section 5.4, that request both does post and get. And lastly we use urllib.request.urlopen().read() to format the data back into a python dictionary. This is a format we can more easily work with.

Dashboard /... / API/AJAX

api/komp.php

Created by Sigurd Hellesvik, last modified on Feb 18, 2019

All interaksjon med APIet treng ein nøkkel 'api_key'. Dette er ein streng med bokstaver du må spørre EDB om tilgang til.

Det ligg på: `https://omegav.no/api/komp.php` `(Kun på omegav.no:8091 av dato 18.02.2019)`

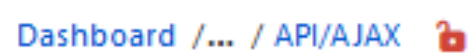
action	parameter	return	beskrivelse
bruker	rfid	id, brukernavn	Den sjekker også om brukeren har ein pinkode, trur eg -urd
saldo	rfid, pin	id, brukernavn, saldo, kriterett, fakturakunde	Denne fungerer kun viss bruker har rett pin.
kjop	rfid, ,rom,sek,hylle,row, col, dybde	True, False or error	Viss komponenten finnast, og bruker nok saldo, gjennomfør kjøp og registrer i databasen.

PS: "action" må brukast når ein kaller funksjonen.

Eksempel kjøp

```
    def API_buy(rfid,rom,sek,hylle,row,col,dybde,api_key):
1       try:
2           url = 'https://omegav.no:8091/api/komp.php'
3           data = urllib.parse.urlencode({'handling' : 'kjop',
4                                   'rfid' : rfid,
5                                   'rom' : rom,
6                                   'sek' : sek,
7                                   'hylle' : hylle,
8                                   'row' : row,
9                                   'col' : col,
10                                  'dybde' : dybde,
11                                  'key' : api_key})
12          data = data.encode('utf-8')
13
14          request = urllib.request.Request(url,data)
15          response = urllib.request.urlopen(request).read()
16          search_content = json.loads(response)
17          if 'Komponent not found' in search_content:
18              print(search_content)
19              return False
20          return 'success' in search_content
21      except:
22          print("exception")
23          return False
```

Figure 9: Documentation from the wiki of the API



ajax/kompsok.php

Created by Sigurd Hellesvik, last modified on Mar 21, 2019

https://omegav.no/ajax/kompsok.php

Her ligg komp sin database i json format. Dette kan parses for å finne info om komponenter.

Kvar komponent har følgende identiteter: id, navn, beskrivelse, produsent, nokkelord, pakketype, pris, oppdatert, antall, lager

Med følgende underinfo om lager: id, komponent_id, antall, type, rom, hylle, rad, kolonne, dybde, oppdatert, oppdatert_av, seksjon, count

Eksempel for parsing av kompsok.php

```
def AJAX_search_component_pos(rom,seksjon,hylle,rad,kolonne,dybde): #rom can be 'OV', 'GM' or 'VendingMachine'
    try:
        url = 'https://omegav.no:8091/ajax/kompsok.php'
        data = urllib.parse.urlencode({'rom' : rom,
            'seksjon' : seksjon,
            'hylle' : hylle,
            'rad' : rad,
            'kolonne' : kolonne,
            'dybde' : dybde})
        data = data.encode('utf-8')
        request = urllib.request.Request(url,data)
        response = urllib.request.urlopen(request).read().decode('utf8')
        search_content = json.loads(response)
        if('0' in search_content):
            if int(search_content['0']['antall']) < 1:
                print("Tomt for varen")
                return {'error': "Location is empty"}
            else:
                return search_content['0']
        else:
            return search_content
    except:
        print("Except")
        return {'error': "API Error"}
```

Figure 10: Documentation from the wiki of how to search for components using the API

5.7 Nordic Semiconductors: BLE Architecture

The nRF52 we use in this project is programmed in the c programming language, as is normal for many microcontrollers. Yet to program them, we need to know what an API is. And luckily for us, we just learned that. Now, this is because Nordic Semiconductors do not only produce the microcontrollers, they also make software for the microcontrollers, called Soft Devices. These are a set of functions we have to use to program the nRF52. When downloading the Soft Device, we get a folder with files and headers for these functions. When using the SDK, which we get to in a bit, the Soft Device function can more easily be found by backtracking function calls. Many IDEs (Integrated Development Environment) have support for "Go to function definition" which is helpful. As things go, it is nice to have functions made for us. But of course there is a downside as well. It is a lot of different functions, so it can be hard to find what function you want to use. To be more exact, the Software Device contains over a thousand files, resulting in a total of almost 190 000 lines of code and 140 000 lines of comments. Of course they are sorted in respective folders and with helpful file names, so if you want to use the Bluetooth, you navigate to the BLE(Bluetooth Low Energy) folder etc. Even so, it can be overwhelming. Especially for an application as small as ours, it would be unnecessary to have to learn how all this works. But Nordic Semiconductors have supplied us with yet another tool. They call it a Software Development Kit, the nRF5 SDK. Here we are supplied with a lot of examples covering different uses of the nRF52. Of course we won't do the exact same thing as in the example. But it does roughly the same thing as we want, and more importantly, uses the correct functions from the Soft Device. All we have to do is do the same things as the example does, but in different ways. As we have, and will, see(n), Bluetooth Low Energy is a complex protocol, which translates into a complex implementation. This is another reason that the SDK is a great support, as we can find examples which already do the setup for bluetooth. Since we not want to do anything fancy with our bluetooth, we are content with editing the examples. I will as such not go trough the setup in detail. Although, I will give a rough explanation of the different parts we use when doing BLE, which is what will have to be initialized when using Bluetooth. A generalized architecture of the nRF52 can be seen in fig. 11. ATT is for ATTribute Protocol. To say it in a very simplified way, all the ATT does is to keep track of three variables. A 16-bit handle , which identifies the attribute. A value of a certain length, which is the information we can work with. And lastly the UUID which is defined by the GATT. Now, GATT is for Generic ATTribute Protocol. This defines how multiple ATT attributes are grouped together to do useful stuff. We call these groups Services. For example GATT can make one service that gives the status of a button. It has to define an general UUID, say 0x2800, and one for the handle, say 0x0100. Now it can also do the same for another button. In that case, all we need to do is to give a new handle to the new button, while the general UUID remains the same. GAP (Generic Access Profile) on the other hand, defines the general working of our BLE. For example, it is what we set to choose if our chip is a broadcaster or observer. Or GAP sets if we wanna be a peripheral or central. SMP and L2CAP are bluetooth protocols which lays deeper down in the software stack, meaning we won't have to use them directly. You can look them up yourself if you want to know more. The Host then use the controller to make BLE signals for us work. If you want to learn more about the bluetooth protocol, there is a lot of tutorials and explanations on the internet.

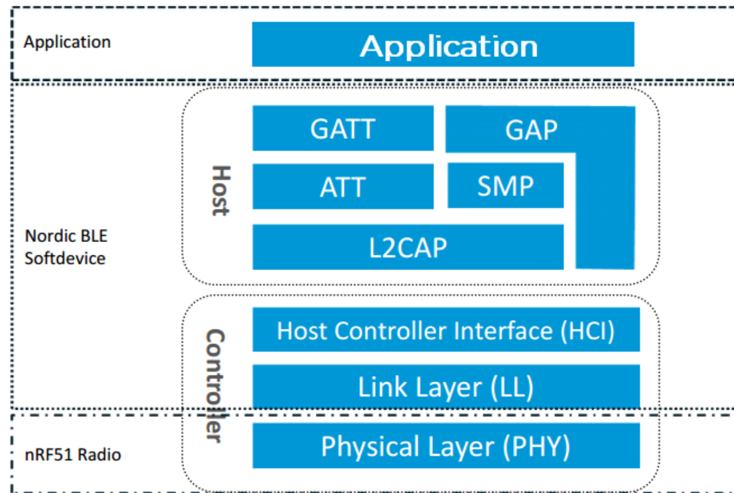


Figure 11: A generalized architecture of the nRF52

5.8 Nordic Semiconductors: Programming

The behaviour of the bluetooth of the module is well described in section 3.4. As our custom PCB did take time to design and produce, I used the nRF52 Development Kit for testing my code when I wrote it. For a beginners guide in how to program nordic, see the guide I wrote at: `https://confluence.omegav.no/display/OV/nRF52dk`. Now I will detail the changes I made to make the Module work as we wanted. The Module should start to advertise when a button was pressed, then host a service that hold a position, which the Hub should be able to read. Then when the Hub disconnect, the Module should go back to sleep.

See `https://git.omegav.no/ov/neokomp/tree/master/Software-modul`, in the folder "testurd",for the code I wrote for the developement kit. I recommend using Segger Embedded studio, as I will use this when I explain. All functions are in main.c unless otherwise specified. I started out with the ble_peripheral example ble_app_blinky. First I figured that to test with the Development Kit, it would be practical to have more buttons, so in the main.c file I added more in the button_init() function. I also had to define the buttons in the top of the file. And to handle different buttons differently, I made a switch case in button_event_handler(). Now we can choose a different 4 length array based on what button is pressed. Anyway, the code inside one switch could be changed individually by the encoders when we implement on the custom board. The example I am working with can send only send one integer at the time. And we need to send four: "Hylle", "Rad", "Kolonne" and "Depth". Thus I went to the ble_lbs_init() (from ble_lbs.c) function I found in services_init(). Here I edited init_len and max_len to 4. Thus we can send an array with length 4.

The "blinky" example advertise all the time. But we only want to advertise on button press. While main.c already have an advertising_start() function. So I added that one to the button_event_handler(). And then I figured out how advertising_start() works. And I made advertising_stop(), which is the same as start, but using sd_ble_gap_adc_adv_stop(), instead of sd_ble_gap_adc_adv_start(). To not start advertising multiple times on multiple button presses, I had to include the global variable advertising_now. Then all I had to do to stop advertising when connection stops was to add advertising_stop() to the BLE_GAP_EVT_DISCONNECTED case in the ble_evt_handler() function. And that is both how simple and how hard it is to make the bluetooth part of our Modules code. Then we migrated

the bluetooth code from the DK to the custom Module with no substenital problems.

5.9 Future features

As of when this report was handed in, the following features was yet to be implemented into the the API and bluetooth of the finished product. These will be fixed by me later, or by someone else in the future.

The API does not have support for search by barcode. This will have to be added to the komp-sok.php, so that every component in the json file gets the identity barcode(strekkode). Furthermore it is not possible to toggle a led on the Module fro the Hub. This can be done by making the Module scan in intervals, and then the Hub will advertise when it wants to turn on a led. The Module will only scan for one advertisement UUID, and so the Hub can target a specific Module. Then the Module can turn on a led with a timer attached when it connects to something. While writing the code, I could not figure out how to send more than a 4 long array per service. We fixed this by only sending "Hylle", "Rad", "Kolonne" and "Dybde". Then "Seksjon" will have to be a part of the name(UUID). This could be fixed either by figuring how to send more characters, or by doing two different services, with four characters each.

6 Problem description: Sigurd Heide Rosland: Shelf Module Hardware & Non-bluetooth firmware

The Shelf Modules will be a distributed, hands-on interface for the system. It should be small, ideally easy to mount onto a standard storage shelf without being in the way. This suggests a slim, rectangular layout. The module must provide an intuitive user interface that requires little or no explanation, and must support wireless communication running in parallel. The module should be power efficient, and should be reasonably flexible with regards to input voltages.

6.1 Microcontroller

A microcontroller on the module will be the sole control unit and only programmed device. It will handle wireless connections, control a display with simple geometric shapes as well as reading and interpreting various user inputs. It must be powerful enough to perform all these tasks in parallel, as well as potentially new features in later development.

6.2 Small display

In order to create an intuitive user interface, it was concluded that a graphical display would be very useful. If possible, it should be small, so that the module could still fit on the front of a shelf plate. It should be connected via a serial interface in order to conserve pins on the microcontroller.

6.3 Input devices

The physical input devices should be simple and intuitive. Their main function is to allow the user to select a shelf position {row, column, depth}. Preferably they should give some sort of tactile feedback when used.

6.4 LED

An LED should be mounted on the module. This will be used to visibly indicate a particular shelf, a function initiated via the web interface, in order to easily navigate the shelves.

6.5 PCB Antenna

The wireless functionality on the module will utilize a PCB antenna. It was found that since the device does not require a long range, a more sophisticated antenna was not needed, and was undesirable from a cost perspective.

6.6 Shelf Module Hardware Drivers and Firmware

The device will be in sleep mode until an input event occurs from encoders or button.

7 Solution: Sigurd Heide Rosland: Shelf Module Hardware & Non-bluetooth firmware

The Shelf Module is implemented of a small PCB (120x25mm) with a nRF52832 microcontroller. The user interface consists of a 128x96 pixel RGB screen, three incremental encoders and a button. Additionally, the board features a step-down switch mode regulator as well as additional supporting circuitry.

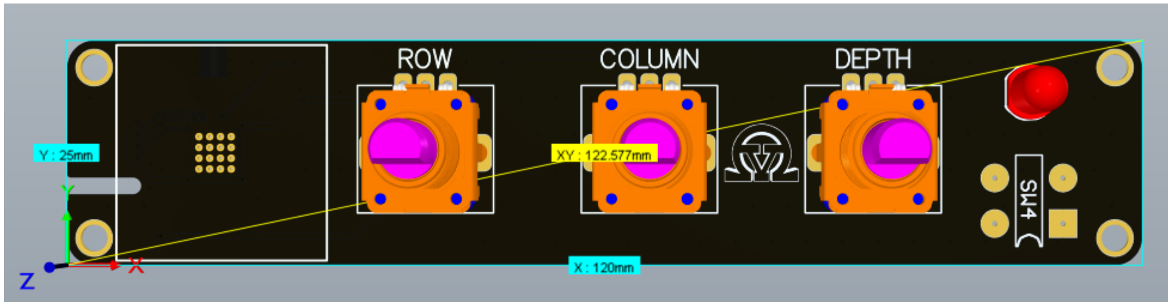


Figure 12: Prototype PCB design. The white square represents screen, which has a fold-around flat cable.

7.1 nRF52832 microcontroller

The nRF52 was selected due to being a mature part commonly used in similar bluetooth products. It is based on a 32-bit Cortex-M4 ARM processor, with peripherals designed with a focus on 2.4GHz radio applications. The device with supporting circuitry can be seen in Fig 13

7.2 Switch Mode Converter

A RT8059 PWM Step-Down DC/DC Converter was used. It is rated for 1A and uses a PWM frequency of 1.5MHz. An electric feedback circuit to the regulator configures the operating voltage of

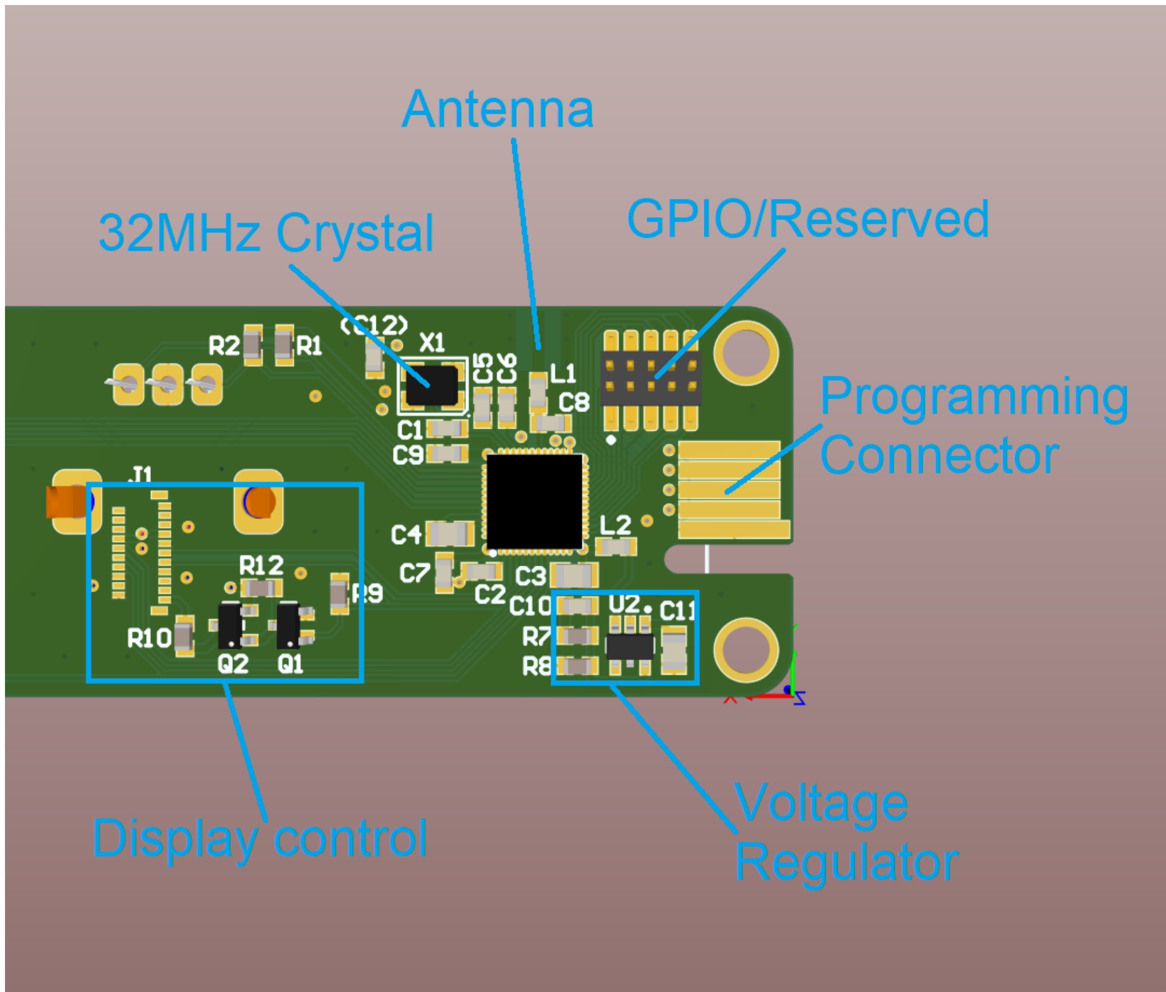


Figure 13: Prototype PCB design, controller area detail.

the microcontroller to around 2.83V. As a result the required range of the regulator input voltage is 2,83V-5,5V. The nRF52, the LCD control and the encoders are driven by this voltage domain, while the LCD backlight is driven directly by the input voltage. Thus, the LCD screen will black out if the regulator input voltage is too low, limiting the input voltage range to around 3.3-5.5V for practical purposes.

7.3 RGB Screen

A MIDAS MCT010A0W12896LMLIPS RGB LCD screen with a SPI interface is the primary graphical user interface. The screen is connected via a 21-pin FPC cable. The screen will display a grid representing the shelf, with one box highlighted. An RGB display gives much greater freedom to modify and improve the graphical user interface. The backlight is controlled via a transistor driver circuit, in order to be able to drive the backlight at a higher voltage.

7.4 Input Devices

7.4.1 Incremental Rotary Encoders

The rotary encoders are used by the user to increment or decrement the ROW, COLUMN or DEPTH indices of the currently highlighted shelf position. A clockwise rotation represents an increment, while a counterclockwise rotation represents a decrement of the index. The encoders have mechanical detents, and a full rotation is equivalent to 12 iterations of the index. The specific encoder was selected based on user feedback on comfortable usage.

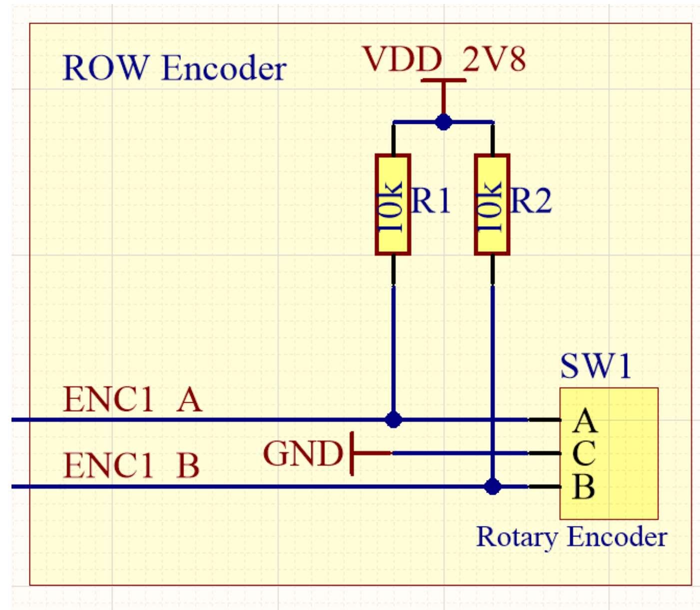


Figure 14: Encoder circuit with pull-up resistors

7.4.2 Push Button

The push button is located at the right-hand side of the module in Fig 12. A press of the button signals that the currently highlighted shelf position should be selected and sent to the hub, resulting in the corresponding product to be added to the shopping cart.

7.4.3 LED

A red 5mm LED was selected. A red LED minimizes the voltage drop over the LED.

7.4.4 PCB Antenna

For the prototype PCB, it was important to leave room for tuning the PCB antenna of the module. The antenna is a reference design by Nordic Semiconductor, modified to the specific application. The antenna is designed longer than in the reference design in order to be able to tune it by shortening it with a scalpel. However, the antenna was found to work sufficiently well to radiate to the entire room that the system is intended to operate in. If the system should be made into a more general purpose system, the antenna should be tuned empirically to maximize range.

7.4.5 Passive Components

It is desirable to maximize the ease of assembly of the module, so that hopefully even inexperienced solderers at Omega Verksted can assemble and repair modules in the future. However, through-hole components were found to be infeasible, so all passive components are surface mounted in either 0603 or 0805 (imperial) packages.

7.5 Second Revision & Future Improvements (post delivery)

Because of the fold-around cable, it was difficult to estimate the correct position of the FPC connector for the display. The position of the connector will be adjusted for better mechanical assembly and visual result. The antenna will be tuned to improve its range (for a more generalized use case), and a power connector will be selected to provide better strain relief. The PCB may also be improved in various ways based on user feedback. For instance, another encoder to index the "HYLLE" dimension may be added. We also see potential in using color coding between the screen and the shelves to improve the intuitiveness of the interface.

7.6 User Interface Firmware

7.6.1 GPIO

The red LED and display backlight are simple GPIO, writing to P0.06 and P0.03 respectively. (*nrf_gpio_pin_set*)

7.6.2 Button

The button is read via an interrupt handler: *button_event_handler*. Pushing the button is one of the ways to trigger the module to exit sleep mode. When the button is pushed in the wake state, the currently selected position by the encoders will be sent to the hub, and the position indices will be reset for the next selection.

7.6.3 Encoders

The encoders are quadrature encoded, and are read via an interrupt handler (*button_event_handler* on a different pin). When a falling flank is detected on channel A, the state of channel B is read using *nrf_gpio_pin_read*, and the direction of rotation is thus determined.

7.6.4 LCD Display control

The display is controlled by a ST7735 TFT-LCD Controller via a 4-wire SPI interface (*CS, MOSI, SCK* and *"DATA/COMMAND"*). A preexisting code library for the chip was used, found in the Software Development Kit provided by Nordic Semiconductor (*examples/peripheral/gfx*). The library was configured to use the correct screen size and pin mapping for the hardware (*sdk_config.h*).

Appendices

A Module Schematic