

AUTONOMT BEVATTNINGSSYSTEM

Jonatan Karlberg



39:2019

Datum för godkännande: 10.12.2019
Handledare: Joakim Isaksson

EXAMENSARBETE

Högskolan på Åland

Utbildningsprogram:	Informationsteknik
Författare:	Jonatan Karlberg
Arbetets namn:	Autonomt bevattningssystem
Handledare:	Joakim Isaksson
Uppdragsgivare:	

Abstrakt

Detta arbete beskriver konstruktionen och designen såväl hårdvaru- som mjukvarumässigt för ett autonomt bevattningssystem ämnat för växter.

Genom att använda sig av olika sensorer kan systemets huvudenhet få en generell överblick över växtens tillstånd och bestämma bästa tillfälle för när växten behöver vattnas.

Vid bevattning så styr huvudenheten en ventil som är kopplat till ett existerande vattensystem. Ventilen förblir öppen tills huvudenheten med hjälp av sensorerna anser att växten har fått tillräckligt med vatten.

Via ett externt program kan användaren fjärrstyra huvudenheten, observera växtens tillstånd samt göra egna inställningar om växten har speciella skötselbehov.

Nyckelord (sökord)

Raspberry Pi, Automation, Sensor, Analog-digital konvertering, Gränssnitt, Växt, Bevattning, Fjärranslutning

Högskolans serienummer:	ISSN:	Språk:	Sidantal:
39:2019	1458-1531	Svenska	37 sidor

Inlämningsdatum:	Presentationsdatum:	Datum för godkännande:
09.12.2019	04.12.2019	10.12.2019

DEGREE THESIS

Åland University of Applied Sciences

Study program:	Computer Science
Author:	Jonatan Karlberg
Title:	Autonomous Watering System
Academic Supervisor:	Joakim Isaksson
Technical Supervisor:	

Abstract

This thesis describes the construction and design in terms of both hardware and software an autonomous watering system for plants.

Through the use of multiple sensors that feed a central unit with data the system is able to perceive a general overview of the condition of the plant, thus enabling it to determine the optimal time for it to receive water.

At the time of watering the central unit is able to control a valve connected to an existing water network, where the valve remains open until the central unit with the help of the sensors deem that sufficient water has been administered.

With the utilization of an external program the end user is able to access the system remotely to observe the condition of the plant themselves, and make adjustments if the plant in question is in need of special treatment.

Keywords

Raspberry Pi, Automation, Sensor, Analog-Digital-Conversion, Graphical Interface, Plant, Watering, Remote Access

Serial number:	ISSN:	Language:	Number of pages:
39:2019	1458-1531	Swedish	37 pages

Handed in:	Date of presentation:	Approved on:
09.12.2019	04.12.2019	10.12.2019

INNEHÅLLSFÖRTECKNING

1. INLEDNING	5
1.1 Syfte	5
1.2 Metod	5
1.3 Avgränsningar	6
2. EXISTERANDE LIKNANDE PROJEKT	7
3. BYGGSTENAR FÖR ETT AUTONOMT BEVATTNINGSSYSTEM	8
3.1 Hårdvara	8
3.2 Mjukvara	11
4. UTVECKLING AV SYSTEMET	14
4.1 Generell arkitektur	14
4.2 Modell-Vy-Kontroller	16
4.3 Sekvensmönster	17
4.4 Gränssnittskomponenter	20
4.5 Tillståndsmaskin	23
4.6 Trådar och datasynkronisering	25
4.7 Filhantering	28
5. RESULTAT	31
5.1 Sammanfattning	31
5.2 Framtida arbete	31
KÄLLOR	32
Bilaga 1: Kretsschema	35
Bilaga 2: Sekvensdiagram - periodiskt kontroll	36
Bilaga 3: Slutprodukt - prototyp	37

1. INLEDNING

1.1 Syfte

Syftet med detta arbete är att utveckla och designa ett autonomt system som skall klara av att bevattna en växt med hjälp av diverse sensorer och komponenter. Systemet skall självständigt kunna observera och bestämma bästa möjliga tillfälle för att utföra bevattningen.

Slutanvändaren skall även ha möjlighet att via ett terminalbaserat program observera systemets tillstånd samt bestämma konfigurationer för hur växten skall vattnas.

Det skall också vara möjligt för användaren att via ett externt program på en separat dator, kunna ansluta sig via ett nätverk till terminalprogrammet för att kunna utföra konfiguration och observation.

1.2 Metod

Första steget blev att bestämma en grundläggande kravspecifikation över de krav som systemet skall uppfylla. När de minimala kraven var bestämda införskaffades komponenter som uppfyllde dessa.

Varje komponent som skulle användas blev först undersökta teoretiskt. Detta gjordes för att kunna förstå deras kommunikationssätt och hur informationen som de genererar skall behandlas.

Därefter byggdes varje komponent upp praktiskt i separata testmiljöer för att kunna utföra diverse tester utifrån informationen från det teoretiska steget.

Slutligen fick varje komponent ett kretsschema som visar hur den kopplas ihop med de övriga komponenterna.

Själva systemet som behandlar dessa delar är uppbyggt med hjälp av programspråket C++. Biblioteket WiringPi används för att förenkla kommunikationen med sensorerna.

Biblioteket curses används för att ge möjligheten att skapa ett gränssnitt i terminalen som kan förmedla information till användaren om systemets tillstånd på ett smidigt sätt.

1.3 Avgränsningar

Systemet hanterar endast en blomkruka/växt och finns endast tillgängligt för Raspberry Pi, då vissa av de bibliotek som utnyttjas är specifikt konstruerade för denna plattform.

Säkerhetsåtgärder som t.ex. systemrättigheter vid användning av systemet beaktas inte. Fjärråtkomstmöjligheten i programmet kommer att ske via redan utvecklade program.

2. EXISTERANDE LIKNANDE PROJEKT

Det existerar många variationer av liknande arbeten. Dessa arbeten är oftast uppbyggda på liknande sätt när de läser av diverse sensorer.

Några exempel på dessa projekt är (Adafruit, 2012, 2018):

- Avläsning av markfuktighet via en SMSM sensor.
- Kontroll av temperatur via en TMP36 sensor.

Dessa system har gemensamma brister rörande hur de kan vidareutvecklas.

Systemen drivs av Arduino. Detta är en mikroprocessor som har en begränsad mängd programmerbart minne, där den populäraste varianten ger tillgång till 32KB (Arduino, u.å). Den låga mängden minne som erbjuds gör det svårt att skapa ett system som kan erbjuda ett användbart gränssnitt. Samtidigt sätter det gränser på mängden funktionalitet som kan implementeras, mer komplicerade system som kräver mer omfattande algoritmer kan möjligtvis inte implementeras pga. deras komplexitet.

Vidare är systemen programmerade i Python. Medan själva programmeringsspråket Python inte är ett direkt hinder så begränsar det mängden andra verktyg som kan utnyttjas då dessa är skriva i bl.a. C/C++. En av dessa verktyg är t.ex. en som underlättar kommunikationen mellan sensorerna.

Därför är målet att ta ett steg vidare och erbjuda ett system som saknar dessa begränsningar och ge ett program med bättre funktionalitet, ett användbart gränssnitt och samtidigt få större kontroll över systemets generella beteende.

3. BYGGSTENAR FÖR ETT AUTONOMT BEVATTNINGSSYSTEM

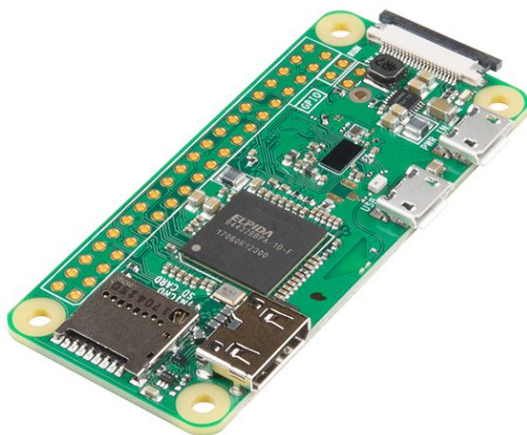
3.1 Hårdvara

3.1.1 Raspberry Pi

Minidatorn Raspberry Pi utvecklades av Eben Upton i samband med Rob Mullins, Jack Lang & Alan Mycroft. Målet var att kunna erbjuda en billig dator som kunde användas i utbildningssyfte för att hjälpa unga att tidigare kunna sätta sig i programmering och förstå hur en dator fungerar. 2006 bildades gruppen “The Raspberry Pi Foundation” som tog fram en billig lösning som blev tillgänglig på marknaden 2012. Idag finns många olika variationer av produkten tillgänglig (Andrew, 2019).

Versionen av Pi som detta projekt använder sig av är en Raspberry Pi Micro W.

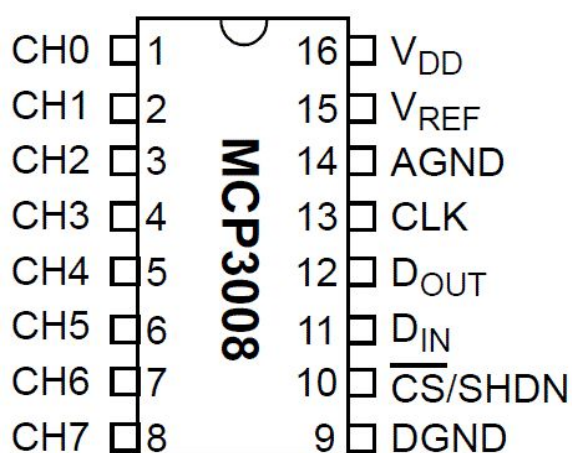
Denna version har både tillgång till wifi, bluetooth och har en flerkärnig processor och 40 stycken GPIO PINs. GPIO är de s.k. “General-Pin-Input-Output” som tillåter oss att skicka läsa och skicka elektriska signaler till andra enheter. En PIN motsvarar en enskild GPIO-kontakt. Det är via dessa pinnar som vi kan kommunicera med andra komponenter när de skall aktiveras. Hur denna variant av Raspberry Pi ser ut kan ses i figur 1.



Figur 1. En Raspberry Pi Micro W (Sparkfun, u.å).

3.1.2 MCP3008

Eftersom vissa sensorer avger sina mätvärden som analoga signaler så behöver vi något sätt att kunna avläsa signalerna digitalt för att kunna arbeta med dem. Detta kan vi uppnå genom att använda en MCP3008. MCP3008 är en s.k. ADC (Analog-Digital-Converter) som kan mäta upp till 8 analoga sensorer samtidigt via kanalerna CH0-CH7, se figur 2. Kretsen läser in sensorsns voltvärde och omvandlar detta till ett motsvarande 10-bit värde. Dessa värden mellan 0-1023 som vi får ut kommer senare användas för att bestämma systemets tillstånd.



Figur 2. MCP3008 datablad (Adafruit, 2016).

3.1.3 TMP36

TMP36 är en analog temperatursensor som anger ett voltvärde som motsvarar nuvarande temperatur i omgivningen. Den här sensorn är till för att vi ska kunna få en bättre inblick när vi observerar växtens tillstånd (Adafruit 2012).

3.1.4 SMSM

SMSM (Soil Moisture Sensor Module) är en analog sensor som läser av kapacitansen i jorden med hjälp av två metalledare. SMSM är en av de viktigare sensorerna eftersom vi i denna version av arbetet använder den för att kontrollera om växten behöver vattnas ifall jorden skulle vara för torr (Adafruit, 2018).

3.1.5 Magnetventil

Själva bevattningen sker via en NC (Normally Closed) magnetventil som endast öppnar sig om den får en inmatningsspänning på ~12V. Ventilen är kopplad till en vattentank eller annan vattenkälla. För att en tank skall fungera som inmatning så krävs det att den skall kunna uppnå ett tryck på 3 PSI vilket tillåter vattnet att fritt kunna flöda igenom. För att uppnå detta värde så måste tanken nå en höjd på cirka. 2 meter (Johnson, 2018).

För att undvika detta kan någon sorts vattenpump användas för att skapa ett konstgjort tryck, men i detta fall så kan det naturliga trycket som finns i vattenkranar användas för att undvika detta problem.

3.1.6 NPN transistor

Hårdvaran i vårt system kan endast ge ut en signal som är högst 3.3V. För viss elektronik är denna mängd för liten för att kunna pålitligt driva den.

Lösningen på detta problem är att använda en NPN transistor som en strömbrytare.

De mera krävande elektronikkomponenterna får sin egen strömkälla och är kopplade med transistorn så när systemet skickar ut signalen med 3.3V så kommer detta tillåta den andra strömkällan att flyta igenom och förse den återstående elektroniken med strömmen som krävs (Electronicshub, 2019).

3.1.7 Resistor

Resistorer används för att begränsa mängden el som kan färdas i en krets för att förhindra att komponenter blir överbelastade och därmed skadade (Rapidtables, u.å).

3.2 Mjukvara

3.2.1 Raspbian

Raspbian är ett operativsystem som har officiellt stöd av grundarna av Raspberry Pi och anses vara det operativsystem som är rekommenderat vid användningen av en Raspberry Pi. Operativsystemet inkluderar en stor samling förinstallerade verktyg, som i stor del kan användas i lärosyfte (Raspberry Pi, u.å).

Möjligheten att installera operativsystemet utan dessa verktyg finns, men för att enklare kunna bygga en prototyp av projektet utnyttjas hela versionen av Raspbian.

3.2.2 WiringPi

Wiring är ett programmeringsramverk för mikrokontrollers med ändamålet att förenkla 'kreativa projekt' (Wiring, u.å). Detta är även det ramverk som Arduino använder i sina produkter.

WiringPi är ett bibliotek utvecklad för Raspberry Pi och är designat att användas med programspråken C/C++. Målet med biblioteket är att göra det möjligt att återskapa *Wiring* ramverket på en Raspberry Pi genom de GPIO som finns tillgängliga på kretskortet (WiringPi, u.å).

Genom att skapa en enkel wrapper runt WiringPis generella funktioner så kan vi enkelt få ett upplägg där vi kan kommunicera med våra sensorer och mikrochip med samma syntax som en Arduino skulle använda.

3.2.3 Curses

Curses är ett bibliotek som implementerar en *wrapper* runt en samling koder som en terminal använder sig för av att kunna visa upp information på skärmen. *Wrappern* ger ett flexiblere sätt att kunna hantera och skapa terminal-baserade program. Biblioteket ger möjligheten att använda ett programmeringsgränssnitt som användare kan använda för att skapa egna terminalfönster och manipulera textpositioner och textfärger.

Metoden för att kunna skriva ut information i terminal med olika färgkombination och tecken kan variera mellan olika operativsystem. Utveckling av ett sådant system blir eventuellt orimligt tidskrävande att porta till andra miljöer eftersom gränssnittet måste byggas upp nästan från grunden. Genom att använda detta bibliotek så undviks detta problem (Pradeep, 2005).

3.2.4 VNC

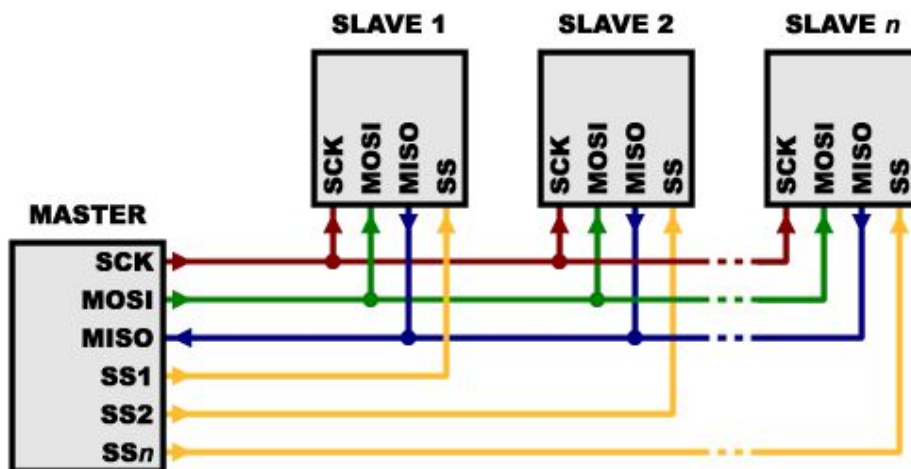
VNC (Virtual Network Computing) är ett fjärranslutningsprogram som tillåter användaren att kontrollera och göra förändringar på datorer via internet. Programmet är baserat på en server/klient arkitektur där servern skickar en kopia av sitt gränssnitt (skärmen) till klienten och tar emot klientens tangent- och musinput för att kunna återskapa vad som skall ske på serverdatorn (Rouse, 2006).

I detta projekt är det Raspberry Pi som har rollen som server och låter oss komma åt programmet med en annan dator när vi vill kunna se och göra förändringar i systemet.

3.2.5 SPI

SPI är en form av kommunikationsbuss som vanligtvis används vid dataöverföring i bl.a. mikroprocessorer och sd-kort. Kommunikationen sker vanligen via tre stycken linjer. En linje agerar som synkroniserings klocka mellan de anslutna komponenterna, en linje används för datatransport och en som indikerar vilken komponent som vi försöker läsa av beroende på dess spänningstillstånd. Se figur 3 för ett exempel på detta.

Anledning till de separata linjerna är att SPI använder sig av ett master/slave system, där mastern är den komponent som står för klockpulsen medan de andra komponenterna agerar som slaves och skickar data vid bekräftelse av denna puls. Skulle vi försöka skicka data samtidigt med flera slaves till mastern så är det sannolikt att vi får ut korrupt data (SparkFun, 2013).



Figur 3. Visualisering av SPI master/slave koppling med flera slaves.
(Sparkfun, 2013)

4. UTVECKLING AV SYSTEMET

4.1 Generell arkitektur

Systemet är uppbyggt med en kombination av programmeringsmönster och andra arkitekturlösningar för att uppnå en design som enkelt kan vidareutvecklas i framtiden för att ge ytterligare funktionalitet.

Systemet är primärt indelat i tre huvudkomponenter enligt arkitekturmönstret Modell-Vy-Kontroller. Hur denna arkitektur tillämpas beskrivs i kapitel 4.2.

Genom att skapa en egen samling av gränssnittskomponenter så kan de mer svårhanterade grafikbiblioteken som curses bli mer användarvänliga vid uppbyggnaden av programmets gränssnitt. Dessa gränssnittskomponenter beskrivs i kapitel 4.4.

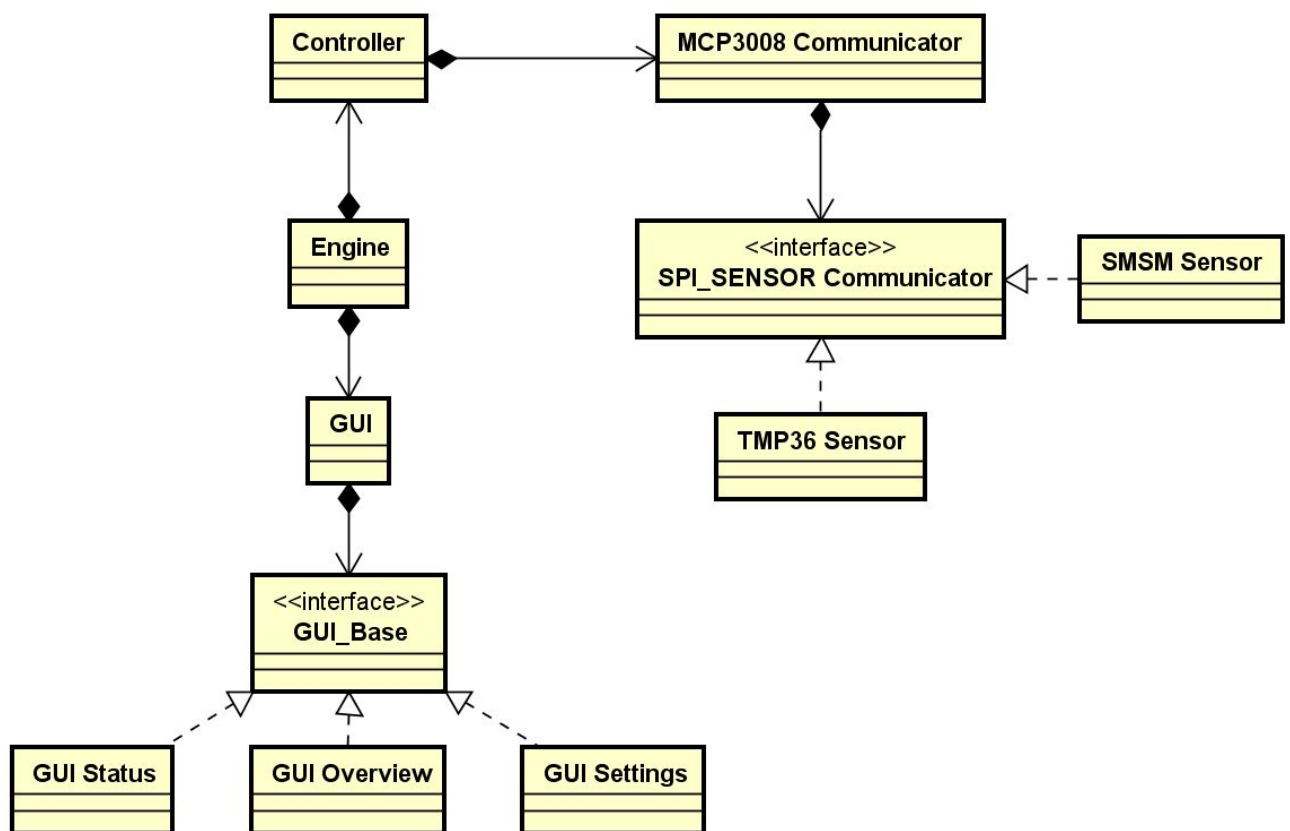
Varje klass som utgör en större del av systemet är uppdelat i en samling gemensamma funktioner. hur dessa funktioner implementeras är upp till vardera komponent beroende på dess uppgift.

Strukturen av dessa funktioner uppnås via användningen av ett sekvensmönster som delar upp programmets flöde i tre separata faser och en tillståndsmaskin som delar klasstrukturen i mindre subklasser beroende på vilket tillstånd systemet befinner sig i. Dessa två mönster beskrivs noggrannare i kapitel 4.3 respektive kapitel 4.5.

För att få maximalt utnyttjande av sensorerna så har de blivit tilldelade sina egna trådar för insamling av data. Vissa komponenter i huvudsystemet har fått egna trådar för att utföra vissa mer kritiska operationer som inte skall kunna störas av andra komponenter under programmets körning. Trådhanteringen beskrivs i kapitel 5.6.

Alla inställningar som kan kalibreras sparas av systemet i en fil som laddas in vid programstart. Detta gör att systemet automatiskt kan återhämta sig ifall något skulle gå fel. Kapitel 5.7 innehåller mera information om filhanteringen.

En förenklad överblick över hur alla komponenter är sammankopplade kan ses i figur 4.



Figur 4. Klassdiagram över systemets generella uppbyggnad.

4.2 Modell-Vy-Kontroller

Systemet ger användaren tillgång till ett gränssnitt som kan användas för att observera systemet och göra förändringar i systemets konfiguration. För att göra systemet både robust och flexibelt så är gränssnittet och sättet datan hanteras på separerade från varandra. Denna typ av arkitektur är mer bekant som Modell-Vy-Kontroller.

Arkitekturen delar upp systemet i tre huvudkomponenter: Modell, vy och kontroller.

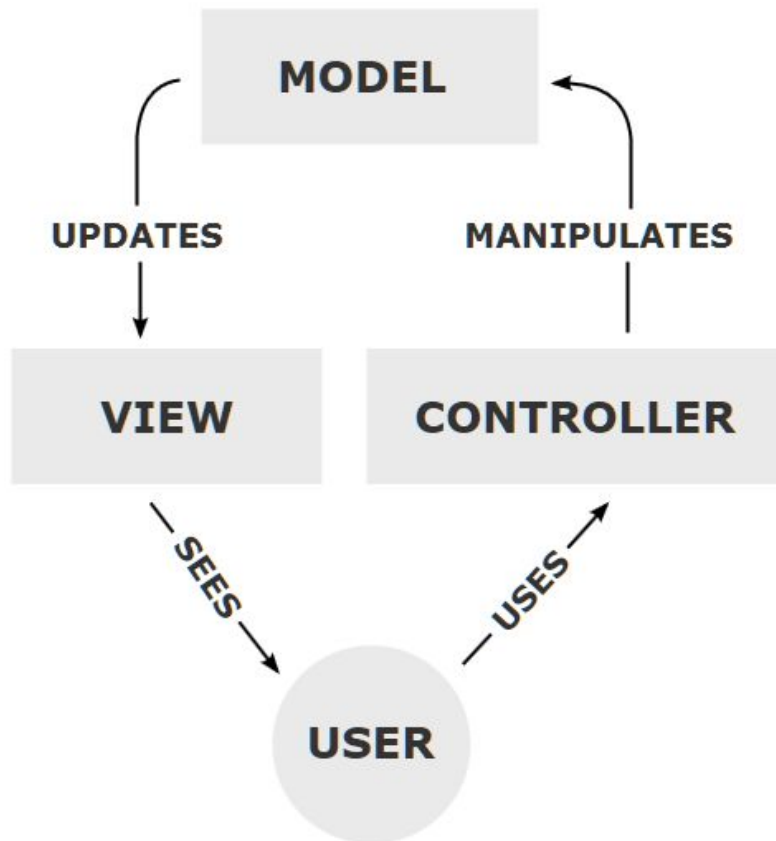
Modellen representerar den data som vi vill hantera. I detta fall så är det sensorernas data som de genererar som representerar våra modeller. Modellerna hanterar de variabler som representerar vår data.

Vyn är vårt gränssnitt som tillåter oss att se och via kontrollern förändra på inställningar. Detta kan vara allt från att enkelt se växtens tillstånd till att göra större förändringar i systemet konfiguration.

Kontrollerns uppgift är att manipulera datan som vår modell representerar. Den har också uppgiften att hantera önskemål som vi kan skicka via vår vy (Tutorialsteacher, u.å).

Genom att dela in systemet i dessa tre komponenter så kan större kontroll fås över hur deras interaktion ser ut. Detta ger också möjligheten att byta ut komponenter utan att resten av systemet behöver göras om.

Genom att begränsa sätten som komponenterna kan kommunicera på så minskas också mängden fel som kan uppstå, ett exempel på hur Modell-Vy-Kontroller vanligtvis tillämpas kan ses i figur 5.



Figur 5. MVC Diagram (Wikipedia, 2010).

4.3 Sekvensmönster

För att göra programmet enklare att vidareutveckla i framtida iterationer så har ett sekvensmönster tillämpats. Detta är ett mönster som oftast kan ses i spelprogrammering men det kan även användas för andra ändamål. Målet med detta mönster är att skapa en separation mellan användarens indata och resten av programmet. Detta gör att vi tydligare kan dela in vilken klass som utför vad i vår kod och uppnå samma upplevelse även om vi skulle byta hårdvara. För att uppnå detta delas programmet in tre separata steg genom funktionerna hantera, uppdatera och rendera (Nyström, 2014).

Om en klass behöver utföra någon av dessa tre funktioner implementerar den sin egen variant av vad som skall ske i detta steg.

Många av klasserna som befinner sig högre i hierarkin får klassmedlemmar som också får implementera någon av dessa funktioner. Då är det den övre klassen som får skyldigheten att anropa respektive klassmedlem och påbörja kedjan av anrop (se figur 6). Klasserna har även möjligheten att vidarebefordra ytterligare information om deras tillstånd till de lägre klasserna om nödvändigt.

```
while (engine.running())
{
    engine.handle();
    engine.update();
    engine.render();
}
```

Figur 6. Översta lagret i ett sekvensmönster.

4.3.1 Hantera

Varje klass som implementerar funktionen `handle()` har som uppgift att hantera och se över användarens indata. Klasserna har en samling variabler som fungerar som flaggor för att kontrollera vilken input som blivit aktiverad under detta steg. Dessa flaggor behövs för att senare i nästa steg kunna utföra åtgärder baserad på den input som vi fått, se figur 7.

```
void GUI_Overview::handle(int input)
{
    m_horizontal_menu.handle(input);
    if (input == INPUT_KEY_ENTER)
    {
        m_selected_option = true;
    }
    else
    {
        m_selected_option = false;
    }
}
```

Figur 7. GUI_Overview implementerar sin egen variant av `handle()` ;

4.3.2 Uppdatera

I uppdatera uppdateras tillståndet för alla relevanta variabler. Detta kan göras genom att begära data från andra klasser eller utföra beräkningar. Det är även i detta steg som vi

analyserar vår lista med indatavariabler från föregående steg och utför diverse uppdateringar beroende på vilka variabler som har blivit aktiverade under denna cykel. Detta ses i figur 8.

```
void GUI_Overview::update(std::unique_ptr<GUI_BASE>& menu, Controller& controller)
{
    //Populate with data from Controller and sensors
    m_current_time.set_message(Utilities::get_current_time());
    m_current_temperature_data.set_message(Utilities::convert_temp_data_to_string(
        controller.get_mcp3008_reading(0)));
    m_current_smsm_value_data.set_message((int)(
        controller.get_mcp3008_reading(1)));
    m_valve_open_data.set_message(Utilities::get_open_closed_state_msg(
        controller.get_valve_open_state()));

    //Update menu based on input settings
    m_horizontal_menu.update();
    if (m_selected_option)
    {
        int transfer = m_horizontal_menu.get_menu_index();
        switch (transfer)
        {
            case 0:
                clear();
                menu = std::make_unique<GUI_Status>();
                break;

            case 1:
                clear();
                menu = std::make_unique<GUI_Statistics>();
                break;

            case 2:
                clear();
                menu = std::make_unique<GUI_Settings>();
                break;

            default:
                break;
        }
    }
}
```

Figur 8. GUI_Overview implementerar sin egen variant av update() ;

4.3.3 Rendera

Det sista steget går ut på följande: finns det något som skall ritas ut eller representeras på något sätt för användaren på skärmen så utförs detta genom att anropa respektive funktion enligt figur 9.

```
void GUI_Overview::render()
{
    m_horizontal_menu.render();
    m_splitting_line.render();
    m_current_menu_title.render();
    m_current_time.render();

    m_current_temperature_text.render();
    m_current_temperature_data.render();

    m_current_smsm_value_text.render();
    m_current_smsm_value_data.render();

    m_valve_open_text.render();
    m_valve_open_data.render();
}
```

Figur 9. GUI_Overview implementerar sin egen variant av render();

4.4 Gränssnittskomponenter

För att underlätta uppbyggnaden av de olika gränssnitten som finns i programmet har en samling klasser konstruerats som fungerar som gränssnittskomponenter.

Detta har gjorts genom att skapa *wrappers* runt curses-bibliotekets olika funktioner för att få enklare och mer användarvänliga funktioner som gör att vi kan visa text på skärmen genom endast en funktion istället för de två till fem funktioner som vanligtvis krävs av curses (se figur 10).

```

//Move text prompt to screen coordinates
move(screen_y_position, screen_x_position);

//Turn color code ON
attron(COLOR_PAIR(color_code));

//Print message
printw("Display Message");

//Turn color code off
attroff(COLOR_PAIR(color_code));

//Update screen so new changes become visible
refresh();

```

Figur 10. Rendering av färgad text enbart via curses funktioner.

Samlingen med komponenter som existerar för tillfället kan delas in i två generella grupper.

- Komponenter som presenterar text för användaren.
- Komponenter som ger tillgång till interaktion, t.ex. en meny.

Varje gränssnittskomponent som hanterar text får klassmedlemmar i form av koordinater som beskriver på vilken position i terminalen texten skall börja och vilket format den har, t.ex.färg. Textfälten kan även få en förbestämmd maxlängd som kan användas för att definiera deras textområde och som hjälper till för att förhindra textfält från att bli överskrivna av andra komponenter.

Komponenterna måste initialiseras med förbestämda värden på sina variabler för att undvika ett odefinierat beteende. Se figur 11. Dessa variabler kan sedan förändras fritt under programmets gång om nödvändigt.

```

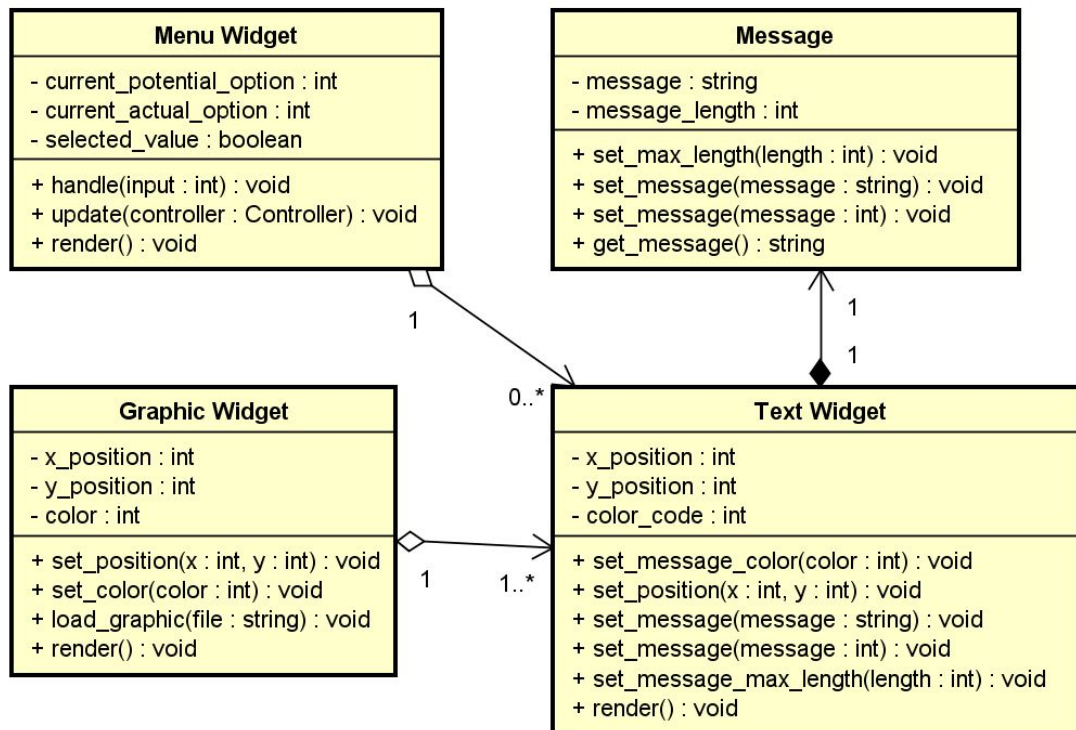
m_current_menu_title.set_message_max_length(20);
m_current_menu_title.set_message("SETTINGS");
m_current_menu_title.set_message_color(GUI_COLOR_WHITE_BLUE);
m_current_menu_title.set_position(4, 5);

```

Figur 11. Initialisering av en gränssnittskomponent i en konstruktor..

För att kunna skriva ut text med ncurses krävs att texten är en sträng eller en pekare till början av en sträng. Detta gör att andra datatyper först måste omvandlas för att kunna användas.

Textkomponenten har därför fått en ytterligare klass, `Message` som medlemsattribut. `Message` har i sin tur en sträng som attribut. Klassens funktioner gör det möjligt att omvandla olika datatyper till denna sträng. Vi kan senare anropa funktionen `get_message()` för att få tillgång till vårt meddelande vid visning av texten (se figur 12).



Figur 12. Klassdiagram för en gränssnittskomponent.

De menybaserade gränssnittskomponenterna implementerar de tre funktionerna som beskrivs i kapitel 4.3 och kan innehålla ytterligare textkomponenter.

Beroende på hur menyerna implementerar dessa tre funktioner så kan vi ge dessa komponenter möjligheten att hantera input som är skickad från de klasser som de är en del av.

På så sätt kan de också uppdatera sitt interna tillstånd på samma sätt som de andra klasser som använder sig av sekvensmönstret (se figur 13).

```

m_horizontal_menu.update();
m_vertical_menu.update(controller);

```

Figur 13. Menybaserade gränssnittskomponenter använder sig av `update()` på sina egna sätt för att uppdatera sitt tillstånd.

4.5 Tillståndsmaskin

För att undvika att gränssnittet blir en monolitisk klass som måste ta hänsyn till alla olika variationer av hur vårt objekt kan bete sig beroende på vad användaren gör har ett tillståndsmönster tillämpats.

Eftersom vårt gränssnitt kan befinna sig i flera olika tillstånd som t.ex. översikt, inställningar, status osv. är målet att introducera en abstrakt klass som representerar alla våra tillstånd (se figur 14). Subklasserna får därmed implementera sina egna variationer av vad som skall ske för det specifika tillståndet som objektet befinner sig i (Gamma, Helm, Johnson Vlissides, 1994).

```
class GUI_BASE
{
    public:
        GUI_BASE();
        virtual ~GUI_BASE();

        virtual void handle(int input) = 0;
        virtual void update(
            std::unique_ptr<GUI_BASE>& menu,
            Controller& controller) = 0;
        virtual void render() = 0;
};
```

Figur 14. Klassdeklaration för GUI_BASE.

För att kunna uppnå polymorfismen som vi får genom detta mönster måste objekten vara användas via en pekare eller en referens. Om basklass variabeln skulle bli tilldelad en subklass utan detta krav så kommer det slutgiltiga objektet inte kunna anropa de funktioner som subklassen har implementerat. I detta fall hanteras variabeln av en unik pekare. Unika pekare har egenskapen att endast ett objekt får vara tilldelad ägarskapet av denna variabel, samt att pekaren frigör sitt allokerade minne om den skulle någonsin bli ogiltig (Cppreference, 2019).

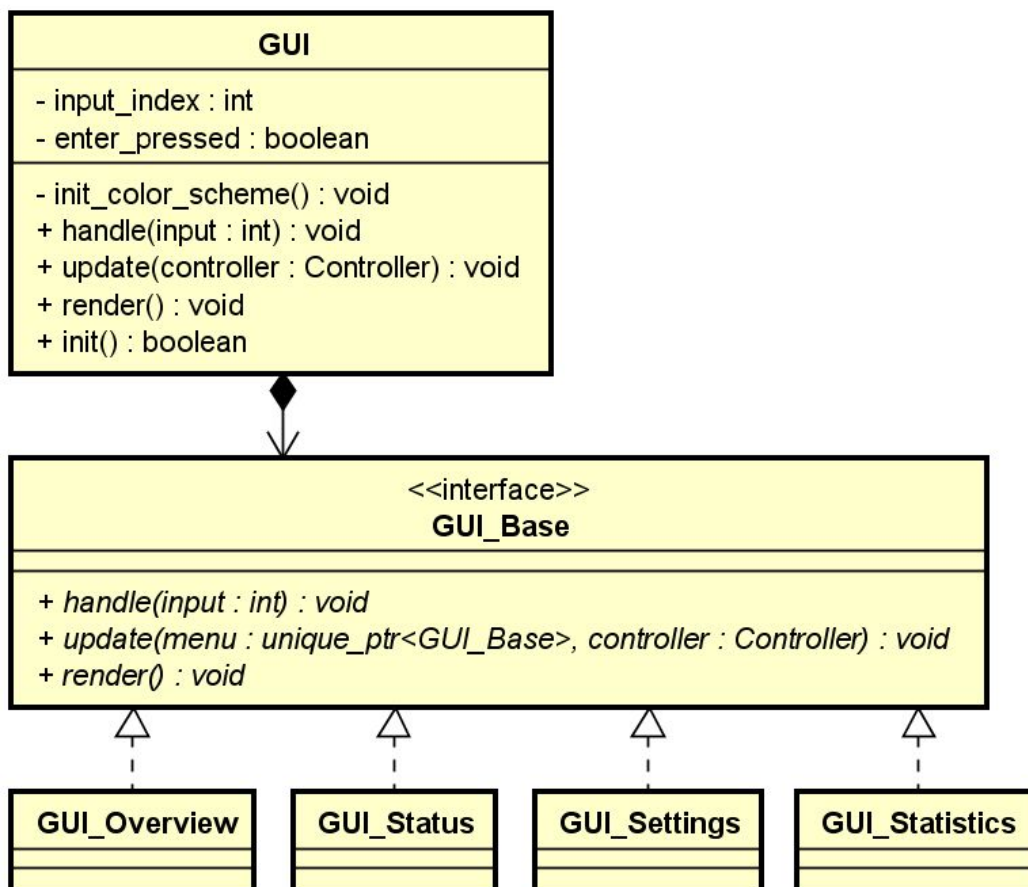
Det enda sättet för en unik pekare att få en ny ägare är att flytta den med hjälp av funktionen `move()`, som kan ses i figur 15.

```
std::unique_ptr<GUI_Overview> x = std::make_unique<GUI_Overview>();
menu = std::move(x);
```

Figur 15. Referensen `menu` blir tilldelad en ny unik pekare: `x`, som blir dess nya tillstånd.

I många fall är dessa tillstånd uppbyggda som singletons, objekt som det endast finns ett exemplar av under programmets gång. I detta fall är klasserna i gränssnittet relativt lätta att konstruera så de skapas när det nya tillståndet uppnås. Eftersom unika pekare används behöver vi inte oroa oss över några möjliga minnesläckor när vi byter tillstånd.

För att underlätta hanteringen av den unika pekaren är den en del av klassen `GUI`. Detta objekt hanterar kommunikationen mellan användaren och kontrollern samt bestämmer olika detaljer angående om hur gränssnittet är initialiserad; lokalisering, färgkombinationer etc och kan ses i nedanstående figur 16.



Figur 16. Klassdiagram för gränssnittet.

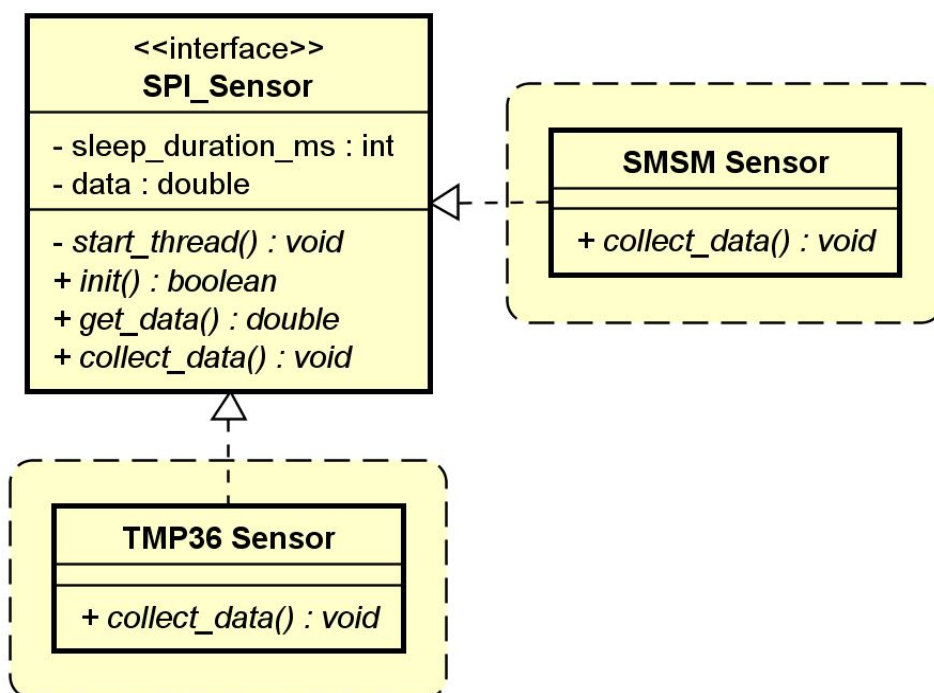
4.6 Trådar och datasynkronisering

Vissa delar av systemet utför sina diverse uppgifter inom olika tidsramar men många av dessa behöver eller kan inte utföras i varje programcykel. Dessa begränsningar beror oftast på hur hårdvara kan bete sig och andra faktorer, där sensorer och annat endast kan samla in information ett visst antal gånger inom en viss tidsperiod.

Samma sak gäller för vattnandet av växten, kontrollen som görs för att se om den behöver vattnas eller inte behöver inte vara konstant utan kan utföras periodiskt.

För att begränsa användningen av timers som håller trådar aktiva i onödan för att se om ett objekt kan utföra sin funktion så har kontrollern och sensorerna blivit tilldelade sina egna trådar för att utföra dessa uppgifter.

Sensorernas programmeringsgränssnitt är uppbyggda via en abstrakt klass som gör att en ny tråd skapas vid initialisering av en ny sensor. Sensorn har möjligheten i sin egna tråd att utföra sina mätningar i sin egen takt och spara resultatet i en variabel som vi sedan kan anropa från andra komponenter när datan behövs. Exempel på hur detta är uppbyggt kan ses i figur 17.



Figur 17. Varje ny sensor som är uppbyggd via SPI_Sensor får sin egen tråd vid initialisering.

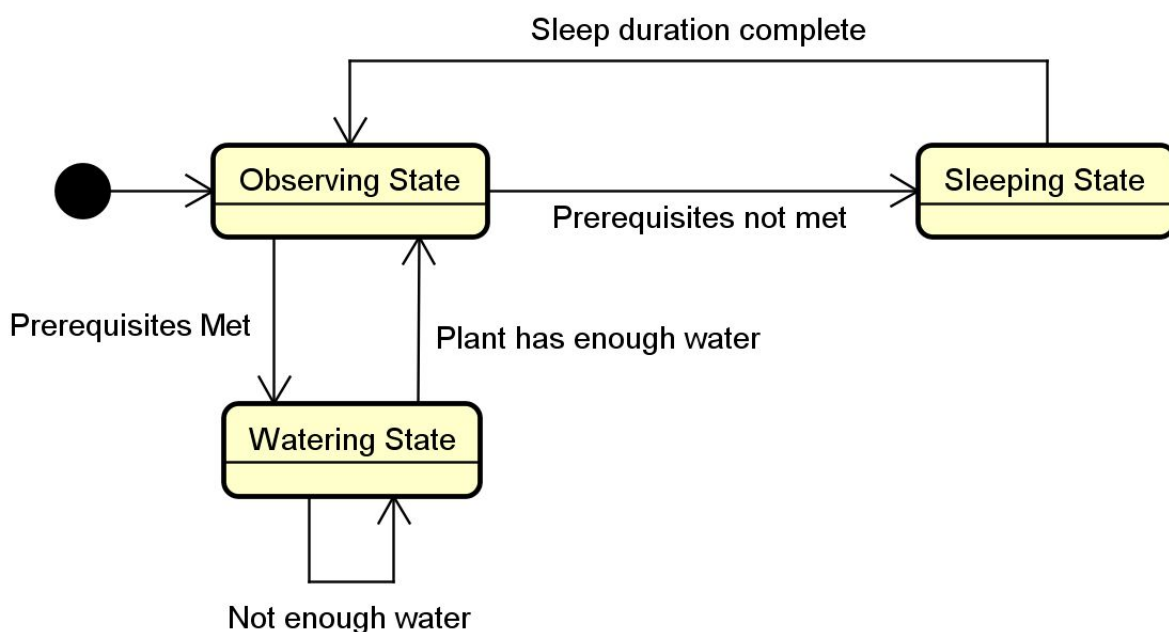
Sensorernas trådar kan också bli inaktiverade när de inte är i bruk. Tidsperioden för hur länge tråden skall förbli inaktiv kan bestämmas under programmets gång. Dock måste den föregående perioden först passera innan den nya börjar gälla.

Normalt är kontrollern också en del av huvudtråden där den skickar och modifierar data beroende på användarens gränssnitt och input. Kontrollern har även fått som uppgift att via en egen tråd periodiskt undersöka växtens tillstånd för att se om det är dags att vattna den.

Detta görs genom att undersöka all data som har samlats in via våra sensorer. Om alla krav uppfylls får växten vatten, annars inaktiveras tråden tills nästa periodiska kontroll skall ske.

När bevattningstillståndet nås så kan tråden inte längre inaktiveras eftersom detta skulle kunna orsaka problem om inaktiva perioden mellan kontrollerna skulle vara väldigt stor dvs. växten skulle kunna få för mycket vatten under denna tid innan nästa kontroll sker.

I bevattningstillståndet är tråden konstant aktiv för att kunna observera mängden vatten som växten har fått. När växten når en punkt då systemet anser att den har fått tillräckligt med vatten så återgår den till sitt observerande tillstånd, enligt beskrivningen i figur 18.



Figur 18. Tillståndsdigram för kontrollerns observeringstråd.

Vid användning av flera trådar som kan anropa och förändra samma data så krävs det att dessa trådar är synkroniserade.

Detta innebär att dessa trådar aldrig kan utföra samma kodavsnitt samtidigt. Skulle flera trådar försöka utföra en uppgift samtidigt så kan det uppstå något som kallas för ett “data-race”, ett tillstånd där gemensamma variabler kan förändras av vardera trådar under uppgiftens tid och ge oförutsedda resultat. En annan risk med flera trådar är att om de skulle försöka läsa och skriva till samma variabel samtidigt så kan datakorruption uppstå (GeeksforGeeks, u.å).

Lösningen på detta problem är att dela in de delar av koden som endast en tråd får vistas i något som kallas för kritiska sektioner. Sektionernas mål är att begränsa mängden trådar som har möjlighet att utföra kod när de når en kritisk sektion. Om en annan tråd redan är inne i samma sektion så får den vänta tills sektionen blir tillgänglig.

För detta används en klass *Mutex*, som står för ömsesidig uteslutning. Objekten som klassen skapar tillåter användningen av “lås” vilket stoppar trådar som anländer till låset om det skulle vara aktiverat.

I detta arbete används ett modernt lås som finns tillgänglig från C++17 som ger tillgång till en variant som kallas för `shared_mutex`. Vanligtvis med dessa lås så är det endast den tråd som anländer först och aktiverar låset som får utföra den kritiska sektionen men detta lås ger oss ytterligare funktionalitet.

Det delade låset ger oss möjligheten att använda en ytterligare metod som tillåter flera trådar att vistas i en kritisk sektion så länge en annan definierad kritisk sektion inte är aktiv, enligt exemplet i figur 19.

Denna teknik används för att kunna begära data av sensorerna från flera trådar samtidigt. Huvudtråden begär datan för att kunna visa upp det för slutanvändaren medan observeringstråden från kontrollern använder datan för att se om växten behöver vattnas. Eftersom båda trådarna endast läser av data så uppstår ingen datakorruption.

```

void SPI_SENSOR::write(double data)
{
    //Lock when writing to one user (This sensor)
    std::unique_lock<std::shared_mutex> writer_lock(m_mutex_data);
    m_data = data;
}

double SPI_SENSOR::read()
{
    //No limit on readers with shared_lock
    std::shared_lock<std::shared_mutex> reader_lock(m_mutex_data);
    return m_data;
}

```

Figur 19. Två funktioner där ett gemensamt lås `m_mutex_data` tillåter endast en funktion att vara aktiv åt gången och tillåter flera trådar att läsa data men endast en att skriva data.

En ytterligare funktionalitet som låset ger oss möjligheten att automatiskt avaktivera låset när funktionen når sitt slut. Vanligtvis eller beroende på hur ens kritiska sektioner är uppbyggda så måste ett lås avaktiveras i något skede efter den kritiska sektionen så att nya trådar kan ta sig in och utföra sina uppgifter. Skulle ett lås förbli låst efter en tråd är färdig med sin kritiska sektion så kan programmet eventuellt låsa sig eftersom inga nya trådar kommer kunna utföra den kritiska sektionen. Detta inkluderar också den tråd som senast aktiverade låset.

Genom att använda en samling av dessa lås för att skydda datan som kommer i kontakt med flera trådar i samband med sina kritiska sektioner så förblir datan skyddad från de generella problem som kan uppstå vid ett flertrådat program.

4.7 Filhantering

Vid varje uppstart av programmet utförs en validering av systemets datafil `waterplant_settings.data`. Skulle filen inte existera eller inte vara åtkomlig vid programstart så skapas en ny automatiskt med förbestämda värden.

Den här datafilen innehåller information om hur ofta vi skall utföra vissa kontroller av systemet, hur ofta vi vill be våra sensorer om nya mätvärden etc.

Datan i den här filen blir också överskriven om vi skulle utföra en förändring i våra

inställningar under menyn “Inställningar” under programmets körtid. Exempel på dessa inställningar kan ses i figur 20.

STATUS	STATISTICS	SETTINGS

SETTINGS		
Controller Update Rate	<sec>:	30
Temperature Sensor Update Rate	<ms>:	800
Moisture sensor update rate	<ms>:	400
Watering start at [<=] value:		25
Watering stop at [>=] value:		50

Figur 20. Inställningsmeny för systemets sensorer och gränser för vattnande.

För att underlätta lagringen av datan så har vi skapat två speciella antaganden som programmet följer vid körning.

Den första är att all data har en förutbestämd position över var den får förekomma i vår fil.

För detta har vi har skapat en enumeration som innehåller positionerna för all data som vi vill lagra och komma åt i framtiden. Detta gör att vi kan enkelt hänvisa till vår lista när vi vill skriva och läsa av diverse värden (se figur 21).

```
enum FILE_DATA_POSITIONS
{
    FILE_DATA_CONTROLLER_TICKRATE    = 0,
    FILE_DATA_TMP36_TICKRATE          = 2,
    FILE_DATA_SMSM_TICKRATE           = 4,
    FILE_DATA_VALVE_OPEN_VALUE        = 6,
    FILE_DATA_VALVE_CLOSE_VALUE       = 8
};
```

Figur 21. Enumeration av datapositioner i fil

Det andra antagandet vi har gjort är att alla enskilda dataposter som lagras i vår fil har en förbestämd maxstorlek på två bytes.

Datan som lagras i två bytes blir placerade enligt de förbestämda positionerna. Dessa två bytes ger oss möjlighet att lagra värden mellan 0-65535. Tar man och jämför med figur 22 och 20 så ses att värdena som varje byte representerar hexadecimalt går att matcha.

Jämför man med `TMP36_TICKRATE` som motsvarar offset 2 så ser man att de bytes har ett värde på 2003. I figur 22 motsvarar detta vårt decimala värde på 800 i figur 20.

Samma sak gäller för de återstående värden i filen.

Offset (h)	00	01	Decoded text
00000000	1E	00	..
00000002	20	03	.
00000004	90	01	..
00000006	19	00	..
00000008	32	00	2.

Figur 22. Hexadecimal visualisering av fil

Tack vare dessa två regler minimeras antalet problem som kan uppstå vid filhantering eftersom vi har begränsat sättet datan kan hanteras på.

5. RESULTAT

5.1 Sammanfattning

Som helhet fungerar systemet som önskat. De olika sensorerna mäter och lagrar data som sedan kan användas av kontrollern för att bestämma när växten skall vattnas.

Fjärrstyrningen är delvis svåränvänd eftersom den använder sig av en klient/server modell så kan det ibland uppstå små fördröjningar när man försöker utföra sina kommandon.

Största utmaningen med systemet var kommunikationen mellan trådarna. Trådarna för sensorerna var skapade via en gemensam basklass vilket gjorde att i vissa scenarion så försökte trådarna använda sig av otillåtna minnesområden vilket ledde till programkrascher. Detta löstes genom att noggrannare undersöka exakt vad tråden försökte göra vid detta ögonblick och åtgärda relevanta problem som detta ledde till. Detta problem uppstår nu inte längre i den slutgiltiga versionen av projektet.

5.2 Framtida arbete

Det finns en stor samling riktning i vilka man skulle kunna bygga vidare på detta arbete.

Några av de punkter som direkt sticker ut är:

- Lagring av statistik från systemet i en databas.
För tillfället sker ingen lagring av hur mycket vatten växten får, hur ofta och när vattnandet sker. Statistik är hjälpsam för att få överblick över hur systemet beter sig.
- Inställningar för hantering av diverse olika växter.
Systemet hanterar som sagt endast en växt. Möjligheten att kunna spara inställningar så att det går att byta ut växten utan att behöva kalibrera om systemet om samma växtart skulle användas igen i framtiden skulle vara bra att ha.
- Mobilapplikation för systemet
Istället för att använda sig av VNC för att komma åt systemet så skulle det vara bekvämare att kunna göra det via en app på mobiltelefonen. Då skulle man samtidigt kunna göra ett mer intressantare gränssnitt.

KÄLLOR

Adafruit. (2012). Using a temp sensor. Hämtad 2019-09-19 från

<https://learn.adafruit.com/tmp36-temperature-sensor/using-a-temp-sensor>

Adafruit. (2016). MCP3008. [Datablad]. Hämtad från

https://cdn-learn.adafruit.com/assets/assets/000/030/456/original/sensors_raspberry_pi_mcp3008pin.gif?1455010861. (CC BY 3.0) <https://creativecommons.org/licenses/by/3.0/>

Adafruit. (2018). Soil sensor, arduino and python I2C overview. Hämtad 2019-09-25 från

<https://learn.adafruit.com/adafruit-stemma-soil-sensor-i2c-capacitive-moisture-sensor?view=all>

Andrew Dennis. K. (2016). *Raspberry Pi Computer Architecture Essentials* (1st ed.)

Packt Publishing.

Arduino. (u.å). Arduino UNO Rev3. Hämtad 2019-09-21 från

<https://store.arduino.cc/arduino-uno-rev3>

Cppreference. (2019). std::unique_ptr. Hämtad 2019-11-03 från

https://en.cppreference.com/w/cpp/memory/unique_ptr

Electronicshub. (2019). Transistor as a switch. Hämtad 2019-10-09 från

<https://www.electronicshub.org/transistor-as-a-switch/>

Gamma Erich, Helm Richard, Johnson Ralph, & Vlissides John, M. (1994). *Design patterns:*

Elements of reusable object-oriented software (1st ed.) Addison-Wesley.

GeeksforGeeks. (u.å). Mutex lock for thread synchronization. Hämtad 2019-09-12 från

<https://www.geeksforgeeks.org/mutex-lock-for-linux-thread-synchronization/>

Johnson Lee. (2018). Calculate pounds per square inch in elevated water storage tanks.

Hämtad 2019-10-03 från

<https://sciencing.com/calculate-elevated-water-storage-tanks-5858171.html>

Nyström Robert. (2004). *Game programming patterns* (1st ed.) Genever Benning.

Padala Pradeep. (2005). NCURSES programming HOWTO. Hämtad 2019-11-04 från

<http://tldp.org/HOWTO/NCURSES-Programming-HOWTO/index.html>

Rapidtables. (u.å). What is a resistor?. Hämtad 2019-10-04 från

<https://www.rapidtables.com/electric/resistor.html>

Raspberry Pi. (u.å). Raspbian. Hämtad 2019-10-26 från

<https://www.raspberrypi.org/downloads/raspbian>

Rouse Margaret. (2006). What is virtual network computing (VNC)?. Hämtad 2019-10-31

från <https://searchnetworking.techtarget.com/definition/virtual-network-computing>

Sparkfun. (u.å). Raspberry Pi Micro W. [Fotografi]. Hämtad från

<https://cdn.sparkfun.com/assets/parts/1/2/2/3/2/14277-01.jpg>.

(CC BY 2.0) <https://creativecommons.org/licenses/by/2.0/>

Sparkfun. (2013). Master/slave SPI med flera slaves. [Ritning]. Hämtad från

<https://cdn.sparkfun.com/assets/8/f/f/6/5/50e5d529ce395f2f7a000000.png>.

(CC BY-SA 4.0) <https://creativecommons.org/licenses/by-sa/4.0/>

SparkFun. (2013). Serial peripheral interface (SPI). Hämtad 2019-11-03 från

<https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi/all>

Tutorialsteacher. (u.å). MVC Architecture. Hämtad 2019-11-18 från

<https://www.tutorialsteacher.com/mvc/mvc-architecture>

Wikipedia. (2010). MVC Diagram. [Ritning]. Hämtad från

<https://upload.wikimedia.org/wikipedia/commons/a/a0/MVC-Process.svg>.

(CC BY-SA 4.0) <https://creativecommons.org/licenses/by-sa/4.0/>

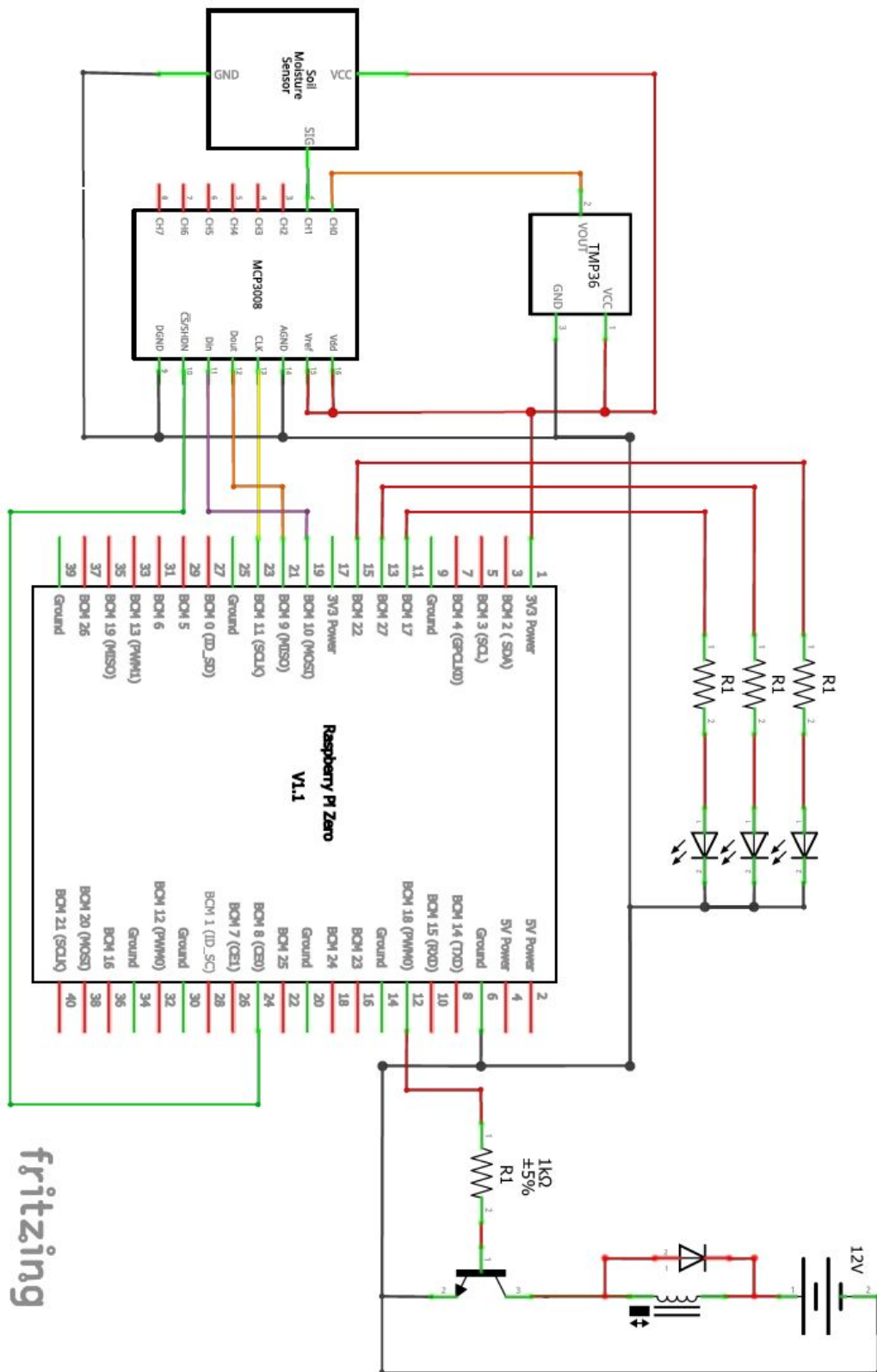
Wiring. (u.å). About Wiring. Hämtad 2019-11-10 från

<http://wiring.org.co/about.html>

WiringPi. (u.å). WiringPi - GPIO Interface library for the Raspberry Pi. Hämtad 2019-10-03

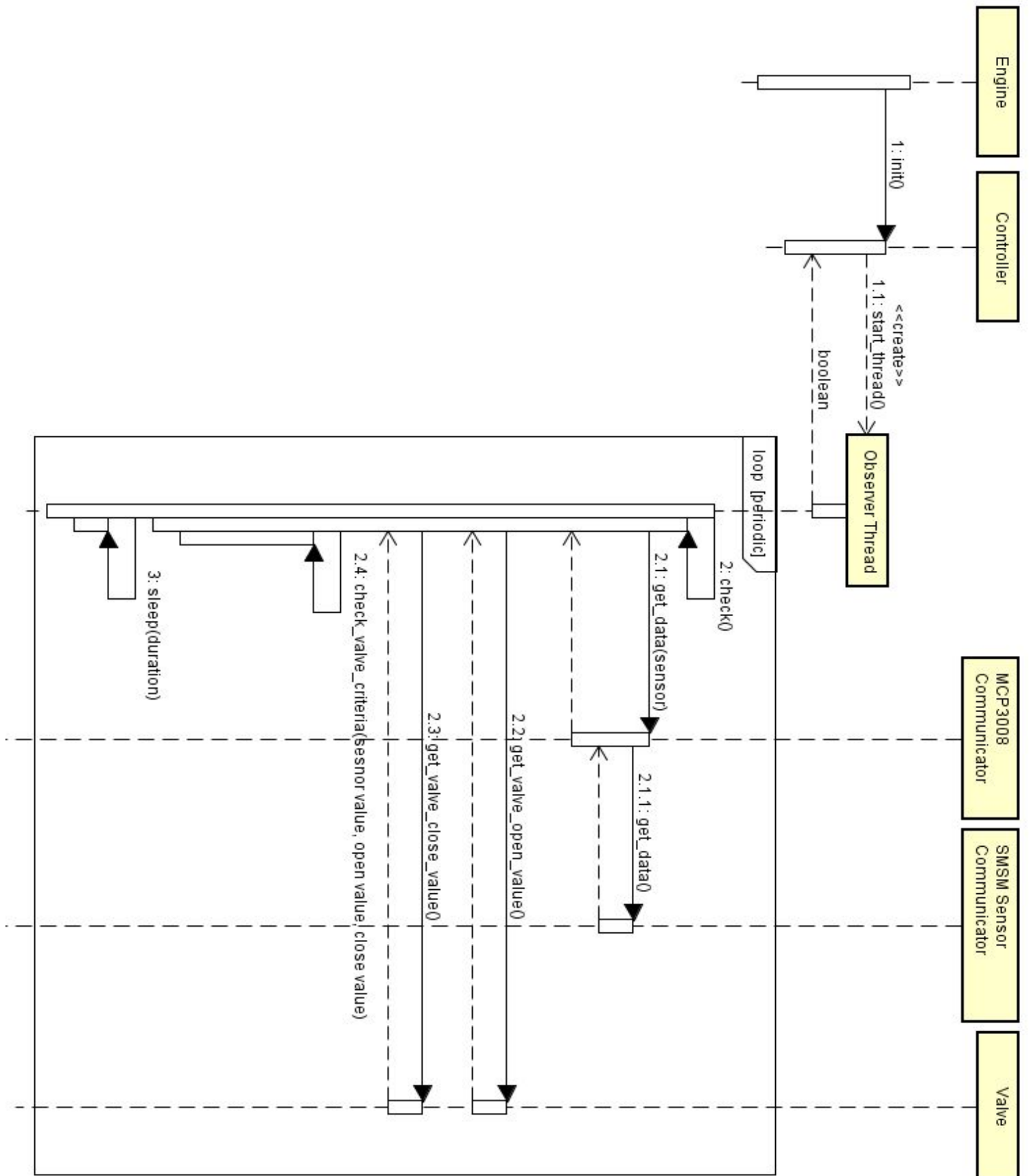
från <http://wiringpi.com/>

Bilaga 1: Kretsschema



fritzing

Bilaga 2: Sekvensdiagram - periodiskt kontroll



Bilaga 3: Slutprodukt - prototyp

