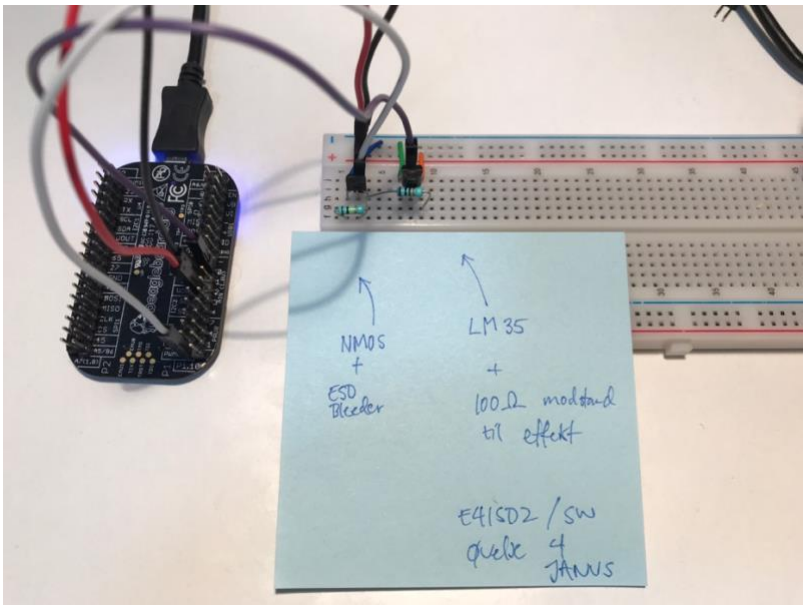
Temperaturservice på PocketBeagle

E4ISD2/SW – Øvelse 4



Rapportskriver: Janus Bo Andersen
Studienummer: JA67494
Afleveringsdato: 29. marts 2020

Indholdsfortegnelse

Forkortelser og definitioner

Forkortelse, begreb	**Definition**
ADC	Analog to Digital Converter
GND	Stel (jord)
NMOS	N-type MOSFET
PID	Proportional-Integral-Derivative teknik til design af regulatorer
PWM	Pulse-Width Modulation
RPC	Remote Procedure Call
SIGHUP	Hangup Signal
TCP	Transmission Control Protocol
$+V_s$, V_{CC}	Positiv forsyning

1 Indledning og formål

Denne rapport besvarer den anden af de obligatoriske afleveringsopgaver i software-delen af kurset E4ISD2. Rapporten er udarbejdet individuelt.

Formålet med opgaven er at oparbejde erfaring med ingeniørudfrodringer i distribuerede systemer, bl.a.:

- Klient-server arkitektur.
- Kommunikation mellem processer via kommunikationsprotokoller (intro til RPC).
- Interaktion med den fysiske omverden via hardwareinterfacing.
- Generel systemprogrammering i Linux.

Løsningen til PocketBeagle er udviklet i C++ ved brug af cross-compiling og remote debugging med toolchain fra Linaro (arm-linux-gnueabihf ver. 7.4.1 2019-02). C++ standarden er C++14 (c++1y).

Løsningen til Linux på desktop er udviklet med CLion. Her er C++ standarden også sat til C++14.

2 Problemstilling / kravsspecifikation

Der skal udvikles en **temperaturservice**. Det er system, hvor:

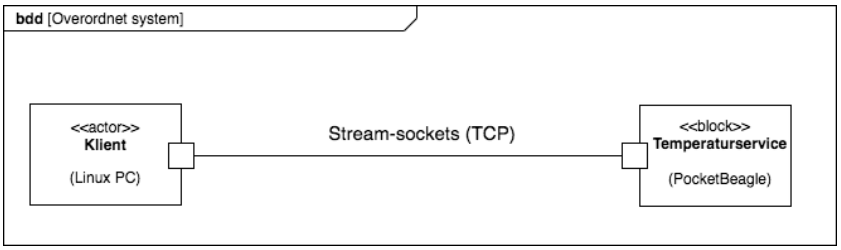
- En service, dvs. et server-program, skal
    - Acceptere forbindelse fra klient.
    - Modtage kommandoer fra klient.
        - Kommando ”GET TEMP”: Levere en aktuel temperaturmåling.
        - Kommando ”SET TEMP <*værdi*>”: Forsøge at sætte temperatur (set-point).
- Temperaturmåling foretages vha. LM35 (minimum) hvert 15. sekund (spec. øvelse 2).
- Temperaturen reguleres, så set-point modtaget fra klient forsøges opretholdt.

Teknologi:

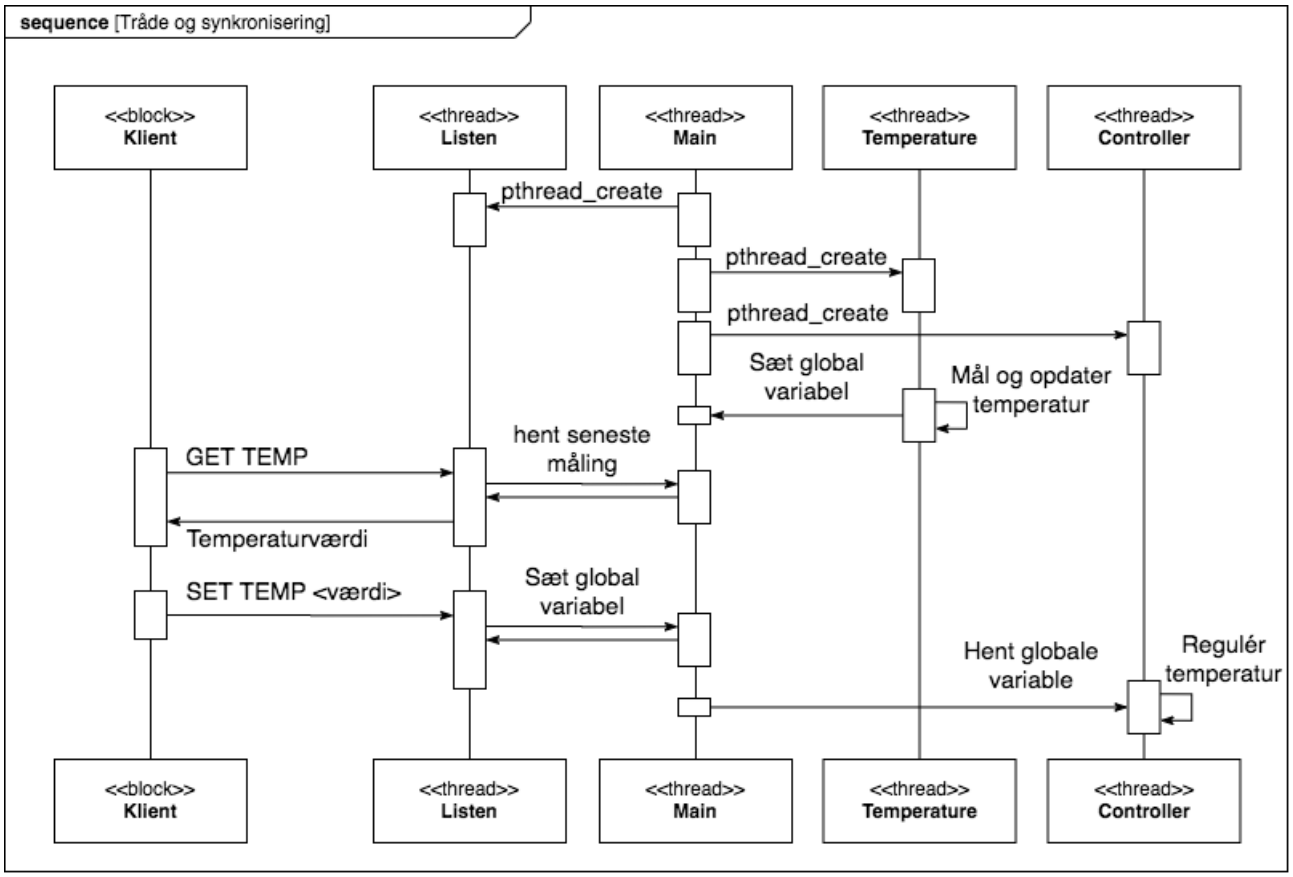
- Server-program kører på Pocketbeagle.
- Klient-program kører på Linux på PC (ej Pocketbeagle).
- Server-klient-forbindelse sker via stream sockets (TCP-sockets).
- Temperatur styres ved afsættelse af variabel effekt i en modstand i nærheden af LM35.
- Ingen krav om performance for servicen.
- Ingen krav om algoritme eller fejlmargin for regulering af temperatur ift. set-point.
- Ingen krav om teknologier anvendt i kommunikationsprotokollen.

3 Systemdesign

Det overordnede systemdesign er som vist nedenfor, at en klient og en server kommunikerer vha. stream sockets (TCP).



Temperaturservicen er en server med "worker" tråde. Der er hhv. en tråd til at håndtere forbindelse (listen), en tråd til at aflæse temperatur (temperature), og en tråd til at regulere temperatur op imod set-point (controller) samt en main-tråd, der opretter de andre tråde, og definerer globale variable. Denne sammenhæng er vist i diagrammet nedenfor.



4 Hardware

4.1 Temperatursensor

Den benyttede temperatursensor er en LM35 i en TO-92-pakke. LM35 har en lineær skaleringsfaktor på $10\,\frac{\mathrm{mV}}{^\circ\mathrm{C}}$. Transducerforholdet er $V_{out}[\mathrm{mV}] = 0\ [\mathrm{mV}] + 10.0\left[\frac{\mathrm{mV}}{^\circ C}\right] \cdot T_{amb}\ [^\circ C]$. Som benyttet i kredsløbet, forsynes sensor med *enkeltforsyning* på 5V (+V_S) og 0V (GND), og relationen gælder da i intervallet fra 2°C til 150°C [2]. Dvs. output er 250 mV ved 25 °C.

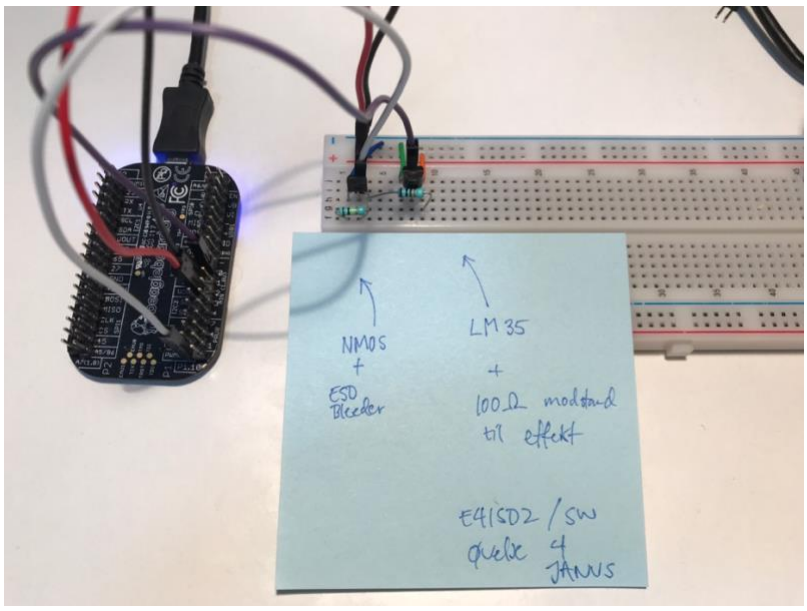
LM35 er en analog sensor, så ADC i Pocketbeagle bruges til at aflæse sensorværdien. ADC'erne på Pocketbeagle er maks. 1,8 V input [1, s. 500]. Sensor kan tilsluttes ADC'en direkte uden beskyttelseskredsløb inden for hele LM35's positive temperaturrating. Hvis BJT-junction i LM35's output stage bryder sammen, så er det dog et problem!

4.2 Kredsløb til variabel effektafsættelse

Der benyttes en ZVN2110A NMOS til kredsløbet til effektafsættelse. Denne benyttes som en PWM-drevet low side driver (kontakt). Den tilsluttede modstand er 100 Ω, forbundet mellem $V_{CC} = 5\ \mathrm{V}$ og stel. Så vha. PWM kan der drives en variabel spænding over, og dermed strøm gennem, den tilsluttete modstand. $R_{DS,on}$ for NMOS'en er mindre end 1 Ω, når fuldt tændt. Så med en PWM duty cycle på 100% afsættes der cirka $P = \frac{V^2}{R} = \frac{(5\ \mathrm{V})^2}{100\ \Omega} = 250\ \mathrm{mW}$. Det er den typiske rating for den "diskrete" slags metal-film-modstande.

4.3 Opstilling

Følgende figur viser opstillingen, som er benyttet ved udvikling og test af løsningen.



Figur 1: Opstilling af hardware til temperaturservice

5 Softwareimplementering

5.1 Synkronisering af tråde

5.1.1 Koordineret nedlukning

Første væsentlige problem, er hvordan en pæn nedlukning af servicen kan koordineres. Af flere grunde duer det ikke bare at lukke den ned ”brutalt”. Bl.a. er det *potentielt* usikkert at lade PWM’en stå ureguleret. Det kan også være problematisk ikke at få frigivet den adresse/port, som serverens socket er bundet til. Den valgte løsning er at benytte en C++ feature, der sikrer atomiske operationer. Variablen, der er garanteret atomisk, er defineret i globalt scope (main.cpp).

```
41 /* Coordinate clean exit :) */
42 std::atomic<bool> exit_now(false);
```

Når SIGHUP-signalet gives til processen, bliver signalhåndtering kaldt, som sætter den atomiske variabel.

```
44 void sig_handler(int signo) {
45
46     // React to caught signal
47     if (signo == SIGHUP) {
48         std::cout << "Received SIGHUP. Bye!" << std::endl;
49
50         // Tell all threads to end now, cleanly
51         exit_now.store(true);
52     }
53 }
```

Alle tråde kører loops, der tjekker denne variabel. Fx i regulator-tråden (controller.cpp):

```
83 void * controller_thread_function(void * value) {
84
85     /* Initialize the PWM with zero duty cycle */
86     enable_pwm(PWM_PERIOD, 0);
87
88     /* difference between set point and current temp */
89     float cur_err = 0;
90     float prev_err = 0;
91     float deriv_err = 0;
92     int new_duty_cycle = 0;
93
94     while( !exit_now.load() ) {
95
96         // Compute controller action
97         cur_err = set_point - cur_temp;
98         deriv_err = cur_err - prev_err;
99
```

Hvor linje 94 tjekker den synkroniserende variabels værdi, og tillader at loop’et brydes, for at få en pæn afslutning på tråden:

```
119         // wait until time to update again
120         sleep(controller_interval);
121     }
122
123     /* Ending - power down PWM*/
124     disable_pwm();
125     pthread_exit(NULL);
126
127 }
```

Linje 121 er afslutningen på while-loop’et. Efter loop’et brydes lukkes PWM ned, og tråden afslutter og kan joines.

Følgende kodestump viser, hvordan tråde oprettes og joines.

```
57      /* Set up signal handler */
58      if ( signal(SIGHUP, sig_handler) == SIG_ERR ) {
59          std::cout << "Can't register SIGINT" << std::endl;
60      }
61
62      pthread_t listen_thread;
63      if ( pthread_create(&listen_thread, NULL, &listen_thread_function, NULL) ) {
64              std::cout << "Could not create thread" << std::endl;
65              return EXIT_FAILURE;
66      }
67
68      pthread_t temp_thread;
69      if ( pthread_create(&temp_thread, NULL, &temp_monitor_thread_function, NULL) ) {
70              std::cout << "Could not create thread" << std::endl;
71              return EXIT_FAILURE;
72      }
73
74      pthread_t control_thread;
75      if ( pthread_create(&control_thread, NULL, &controller_thread_function, NULL) ) {
76              std::cout << "Could not create thread" << std::endl;
77              return EXIT_FAILURE;
78      }
79
80      /* Temperature thread has been stopped by SIGHUP */
81      pthread_join(temp_thread, NULL);
82
83      /* Controller thread has been stopped by SIGHUP */
84      pthread_join(control_thread, NULL);
85
86      /* Safe to kill the listener now */
87      pthread_cancel(listen_thread);
88      pthread_join(listen_thread, NULL);
```

5.1.2 Globale temperatur og set-point værdier

Andet væsentlige problem er synkronisering/kommunikation af temperaturmåling og set-point mellem forskellige tråde. Her vælge en nem løsning at benytte globale variabler. Der benyttes `floats`, og operationer på disse bør være atomiske på en 32-bit-platform som PocketBeagle. Her kunne jeg have brugt den samme løsning som ovenfor, men det ville kræve refactoring af den allerede implementerede løsning fra tidligere uger. Variabler defineres (main.cpp):

```
37 /* These globals are for communication between threads */
38 float cur_temp = 0;
39 float set_point = 0;
```

Og headerfiler benytter extern, så resten overlades til Linker (her fra controller.hpp).

```
18 extern float cur_temp;
19 extern float set_point;
20 extern std::atomic<bool> exit_now;
```

5.2 Aflæsning af temperatur fra LM35

ADC på PocketBeagle er 12-bit, dvs. $2^{12} = 4096$ niveauer i ADC-kode. Værdiområdet på ADC går fra 0 til 4095 og svarer til V_{out} fra LM35 på 0 V til 1,8 V. Det svarer til temperaturværdier 0 °C til 180 °C (men LM35 er selvfølgelig kun rated til maks. 150 °C). ADC-kode konverteres via spændingsniveau til en temperaturværdi i celsius som:

$$T_{amb}\ [°\mathrm{C}] = X_{ADC} \cdot \frac{1{,}8\ [\mathrm{V}] - 0\ [\mathrm{V}]}{2^{12}} \cdot 10^{3} \left[\frac{\mathrm{mV}}{\mathrm{V}}\right] \cdot 10^{-1} \left[\frac{°\mathrm{C}}{\mathrm{mV}}\right] = X_{ADC} \cdot \frac{1{,}8}{4096} \cdot 100\ [°\mathrm{C}]$$

I implementering benyttes `sysfs` direkte til aflæsning af ADC-koden (X_{ADC}). Det er langsomt, så ikke smart, hvis der kræves høj ydeevne. Men det fungerer fint med den krævede samplingsfrekvens på $\frac{1}{15}$ Hz.

Koden er med inspiration fra [1, s. 500-501], og er opdelt in en funktion til at konvertere fra ADC-kode til temperatur, og en funktion til at læse ADC-koden (via `sysfs`). Herunder en kodestump fra temperature.cpp.

```
19 /* We need to get the value from in_voltage0_raw for channel 0 */
20 #define ADC_PATH "/sys/bus/iio/devices/iio:device0/in_voltage"
21 #define SUFFIX "_raw"

48 /* @brief Convert ADC code to temperature in degrees celcius
49  * @param adc_value ADC code (0 to 4095)
50  * @return temperature in C */
51 float conv_temperature(int adc_value) {
52     /* ADC resolution 12-bits (4096), max-value 1.8 V */
53     float cur_voltage = adc_value * (1.80f / 4096.0f);
54
55     /* Convert from voltage reading to celsius */
56     return (cur_voltage * 100.0f);
57 }
58
59 /* @brief Read a value from ADC channel
60  * @param channel the ADC channel 0-7
61  * @return 12-bit ADC code (0-4095) */
62 int read_analog(int channel){
63    stringstream ss;
64    fstream fs;
65    int adc_value;
66
67    /* Make path for the ADC channel fd */
68    ss << ADC_PATH << channel << SUFFIX;
69
70    /* Open ADC fd and read a value */
71    fs.open(ss.str().c_str(), fstream::in);
72    fs >> adc_value;
73    fs.close();
74    return adc_value;
75 }
```

Tråden til løbende aflæsning af temperatur, er som følger

```
23 /* @brief Thread function to monitor temperature in the background
24  * update the global variable CUR_TMP
25  * */
26 void * temp_monitor_thread_function(void * value) {
27
28     int adc_value;
29     float temp;
30
31     while( !exit_now.load() ) {
32
33         // Take a sample and convert to temperature
34         adc_value = read_analog(ADC);
35         temp = conv_temperature(adc_value);
36
37         // Update the global variable
38         cur_temp = temp;
39
40         // Sleep until next sample - this is also a thread cancellation point
41         sleep(TEMP_SAMPLE_INTERVAL);
42     }
43
44     pthread_exit(NULL);
45 }
```

Hvor der igen ses et tjek for koordineret nedlukning.

5.3 Variabel effektafsættelse

Variabel effektafsættelse styres som nævnt via PWM. Her benyttes pin p1.36 (se evt. pin out i bilag). Før brug sættes pin op til PWM vha. `config-pin` utility.

```
janus@beaglebone:~/projects$ config-pin -a p1.36 pwm
janus@beaglebone:~/projects$ config-pin -q p1.36
P1_36 Mode: pwm
```

Den tilhørende PWM er pwmchip0, kanal 0 (dvs. 0:0) [1, s. 286]. PWM'en enables fra C++ ved at skrive til de tilhørende "filer". Dette er vist i kodestumpen herunder (controller.cpp).

```
20 #define PWM_DUTYCYCLE_PATH "/sys/class/pwm/pwmchip0/pwm-0:0/duty_cycle"
21 #define PWM_PERIOD_PATH "/sys/class/pwm/pwmchip0/pwm-0:0/period"
22 #define PWM_ENABLE_PATH "/sys/class/pwm/pwmchip0/pwm-0:0/enable"
23
24 /* @brief Initializes the PWM
25  * @param period (int) the initial period in us
26  * @param duty_cycle (int) the initial duty cycle in us
27  */
28 void enable_pwm(int period, int duty_cycle) {
29     fstream fs;
30
31     /* Write period */
32     fs.open(PWM_PERIOD_PATH, fstream::out);
33     fs << period;
34     fs.close();
35
36     /* Write duty cycle */
37     fs.open(PWM_DUTYCYCLE_PATH, fstream::out);
38     fs << duty_cycle;
39     fs.close();
40
41     /* Write enable */
42     fs.open(PWM_ENABLE_PATH, fstream::out);
43     fs << 1;
44     fs.close();
45 }
```

Den initiale indstilling, er en PWM-periode på 4000 µs og duty-cycle på 0 µs. Regulatoren sætter en ny duty-cycle vha. følgende funktion:

```
69 /* @brief Write duty cycle vaue to PWM
70  * @param duty_cycle (int) duty cycle in us
71  */
72 void write_pwm_duty(int duty_cycle){
73     fstream fs;
74
75     /* Open PWM fd and write a value */
76     fs.open(PWM_DUTYCYCLE_PATH, fstream::out);
77     fs << duty_cycle;
78     fs.close();
79 }
```

Samme princip som før, bare uden overflødig skrivning. Funktionen til nedlukning af PWM svarer til enable_pwm, men med skrivning af 0'er.

5.4 Regulator

Effektafsættelsen skal reguleres, og i denne løsning gøres det udelukkende ved at ændre duty-cycle. Den kan antage heltalsværdier mellem 0 og den satte PWM-periode (4000 µs).

Inspiration til implementering af regulatoren er fra PID-controlleren. Det er en closed-loop regulator med negativt feedback. Reguleringsfejlen er forskellen mellem set-point (+) og den nuværende temperatur (-).

I denne løsning er modellen forsimplet til P og D, med tilhørende parametre K_p og K_d, dvs. justering/gain proportionalt til fejlen (P) og på ændring i fejlen (D). Tuning af parametre foregår typisk ved at K_p justeres indtil der opnås en "hurtig nok" respons, derefter justeres K_d for at nedbringe oscillationer.

Sidste led I med tilhørende parameter K_i benyttes til at eliminere en steady-state-fejl. Dette er udeladt for enkelthedens skyld, da regulatoren ikke er underlagt performance-krav – og regulatoren desuden har et relativt begrænset værdiområde: Den kan selvfølgelig ikke regulere under stuetemperatur (ca. 28 °C her) eller over en maksimalværdi afgjort af stuetemperaturen og begrænsninger fra de fysiske parametre (med 5 V forsyning og 100 Ω-modstand, op til maks. ca. 55-60 °C).

5.4.1 Kode til closed-loop regulator

Implementering af den iterative closed-loop regulator ses nedenfor:

```
94      while( !exit_now.load() ) {
95
96          // Compute controller action
97          cur_err = set_point - cur_temp;
98          deriv_err = cur_err - prev_err;
99
100         // Compute PD
101         new_duty_cycle = new_duty_cycle + (int) (Kp * cur_err + Kd * deriv_err);
102
103         // Limit duty cycle between 0 to PWM_PERIOD
104         if ( new_duty_cycle > PWM_PERIOD) {
105             new_duty_cycle = PWM_PERIOD;
106         } else if (new_duty_cycle < 0) {
107             new_duty_cycle = 0;
108         }
109
110         //output for debugging
111         std::cout << "New duty cycle: " << new_duty_cycle << std::endl;
112
113         // Set the new duty cycle
114         write_pwm_duty(new_duty_cycle);
115
116         // update for next round
117         prev_err = cur_err;
118
119         // wait until time to update again
120         sleep(controller_interval);
121     }
```

Intervallet i afvikling af regulatoren skal være koordineret sampling af temperatur fra LM35. Som det kan ses, benytter regulatoren den globale temperaturværdi.

5.5 Serverklasse (Pocketbeagle)

Serverklassen på PocketBeagle er en iterativ server (modsat en concurrent server – dvs. denne proces håndterer én klient ad gangen). Objektorienteret implementering af serveren i C++ er *stærkt inspireret* fra [1, kap. 11].

Ideen er, at der benyttes en C++ wrapper rundt om sys/socket-systemkaldene. Det giver (i teorien) en praktisk enkapsulering af variabler og state for forbindelsen.

Klassens interface er defineret i socketserver.hpp og implementeret i socketserver.cpp. Interfacet er gengivet i kodestumpen nedenfor:

```
/**
 * @class SocketServer
 * @brief A class that encapsulates a server socket for network communication
 */
class SocketServer {
private:
    int         portNumber;
    int         socketfd, clientSocketfd;
    struct      sockaddr_in   serverAddress;
    struct      sockaddr_in   clientAddress;
    bool        clientConnected;

public:
    SocketServer(int portNumber);
    virtual int listen();
    virtual int send(std::string message);
    virtual std::string receive(int size);
    virtual void closeClient();

    virtual ~SocketServer();
};
```

Implementeringen af listen-metoden (socketserver.cpp) er vist herunder. Som jeg har skrevet i kommentaren, håndterer listen hele processen med fra at sætte socket op til at acceptere en forbindelse.

```
/* listen creates the socket and
 * blocks until an incoming connection established,
 * and then accepts the incoming connection */
int SocketServer::listen(){

    /* Make INET (TCP) socket */
    this->socketfd = socket(AF_INET, SOCK_STREAM, 0);

    /* Check that socket created OK */
    if (this->socketfd < 0){
        perror("Socket Server: error opening socket.\n");
        return 1;
    }

    /* Ensure all zeros in unused part of serverAddress
     * this is an alternative to memset */
    bzero((char *) &serverAddress, sizeof(serverAddress));

    /* Create struct for address */
    serverAddress.sin_family = AF_INET;
    serverAddress.sin_addr.s_addr = INADDR_ANY;
    serverAddress.sin_port = htons(this->portNumber);

    /* Attempt to bind to the socket address */
    if (bind(socketfd, (struct sockaddr *) &serverAddress, sizeof(serverAddress)) < 0) {
        perror("Socket Server: error on binding the socket.\n");
        return 1;
    }

    /* system call listen */
    ::listen(this->socketfd, 5);

    /* Update address of client */
    socklen_t clientLength = sizeof(this->clientAddress);

    /* Accept connection */
    this->clientSocketfd = accept(this->socketfd,
                                  (struct sockaddr *) &this->clientAddress,
                                  &clientLength);

    /* Check that client socket is OK */
    if (this->clientSocketfd < 0){
        perror("Socket Server: Failed to bind the client socket properly.\n");
        return 1;
    }
    return 0;
}
```

Dette var det oprindelige designvalg, og jeg er pt. ikke overbevist om, at det var et godt/skalérbart valg. Men det var et forsøg værd. Det vil give mere mening fremadrettet at "refactor" koden, så disse enkeltfunktioner adskilles i separate metoder.

Som eksempel kan også ses, hvordan send-metoden også er en wrapper for sys/socket-systemkald

```
73 int SocketServer::send(std::string message){
74     const char *writeBuffer = message.data();
75     int length = message.length();
76     int n = write(this->clientSocketfd, writeBuffer, length);
77     if (n < 0){
78        perror("Socket Server: error writing to server socket.");
79        return 1;
80     }
81     return 0;
82 }
```

Receive-metoden fungerer tilsvarende.

5.6 Servertråd (Pocketbeagle)

Selve servertråden findes i listen.hpp og implementeret i listen.cpp. Jeg har valgt at benytte regex til at "afkode" beskeder fra klienten. Det har jeg gjort, fordi det er en ret nem/fleksibel mulighed fra C++11. Hvis menu'en skulle udvides, ville det være forholdsvist nemt at opbygge en mere avanceret "grammatik". Regex-objekter til denne afkodning ses defineret nedenfor:

```
22     /* Regex to match the menu options
23      * https://solarianprogrammer.com/2011/10/12/cpp-11-regex-tutorial/
24      * https://www.informit.com/articles/article.aspx?p=2079020
25      */
26
27     /* Match GET TEMP followed by anything else*/
28     std::regex GET_TEMP("GET TEMP(.*)");
29
30     /* Match e.g. SET TEMP 37.1, with values possible like 37, 37., 37.1, 37.11
31      * But there can only be one decimal point ( [\\.]? )
32      * And then anything else can follow .* */
33     std::regex SET_TEMP("SET TEMP ([0-9]+[\\.]?[0-9]*).*");
```

Disse objekter anvendes til at afkode modtagne beskeder fra klient. Herunder vises det mest "avancerede" eksempel, aflæsning af kommando og værdi:

```
48         /* Attempt to receive a message from the client */
49         std::string rec;
50         rec = server.receive(BUF_SIZE);
51
52         /* Create a regex match object to extract match groups */
53         std::smatch s_match;                                          ///
```

```
62        /* SET TEMP */
63        else if ( std::regex_search(rec, s_match, SET_TEMP) && s_match.size() > 0  ) {
64            /* extract the matched set point value
65             * It is still unsafe, not sanitized */
66            set_point_unsafe = std::stof( s_match.str(1) );
67
68            // Only change if the desired temp is inside valid range
69            if (set_point_unsafe >= TEMP_MIN && set_point_unsafe <= TEMP_MAX) {
70                set_point = set_point_unsafe;
71            } else {
72                std::stringstream message;
73                message << "Invalid temperature range, must be between "
74                        << TEMP_MIN << " and " << TEMP_MAX << ".\n";
75                server.send(message.str());
76            }
77        }
```

Implementeringen benytter, at regex kan uddrage elementer fra et match – fx udtrækkes matchede værdi i linje 66, og konverteres til float (std::stof). Derefter tjekkes, at værdien er inden for et prædefineret interval (20-60 °C). Hvis dette er tilfældet, kan værdien gemmes i den globale set-point-variabel.

Tråd-funktionen er et loop. Overordnet set instantierer tråden en server, starter serveren og looper uendeligt, hvor dne modtager/afkoder/sender beskeder. Her et udsnit, der gerne skulle give overblikket:

```
19
20 void * listen_thread_function(void * value) {
21
```

< her defineres regex >

```
37      std::cout << "Starting server" << std::endl;
38      SocketServer server(PORT_NUM);
39
40      /* Bind and listen for connections
41       * blocks until a connection received
42       * or something failed */
43      server.listen();
44
45      // Loop here until SIGHUP
46      while ( !exit_now.load() ) {                                    ///
```

< her afkodes beskeder – GET TEMP og SET TEMP værdi >

```
79          /* UNRECOGNIZED COMMAND */
80          else {
81              std::stringstream message;
82              message << "Invalid command. Use either \"GET TEMP\" or \"SET TEMP <float value>\"" << ".\n";
83              server.send(message.str());
84          }
85
86      } //SIGHUP received
87
88      pthread_exit(NULL);
89
90  }
```

Det ses, at funktionen igen prøves afsluttet mindre brutalt. En ulempe ved valgene i listen-metoden er, at loop'et blokeres, og ikke kan tjekke exit_now-variablen. Dette er indtil videre løst ved at cancellere-tråden.

6 Test

I figuren til højre vises et udsnit fra en test af koden.

Starttemperaturen er ca. 30 °C (Singapore ☺). Klienten afsender ønske om afsender klient et ønske om at sætte temperaturen til 50 °C. PD-algoritmen, her med $K_p = 50$ og $K_d = 0$, sætter duty-cycle for PWM op, og tempaturen i løbet af få sekunder til omkring 50 °C.

Temperaturen oscillerer omkring dette niveau, som det må forventes, når $K_d = 0$.

Overordnet set virker algoritmenten, om end det kunne være en god forbedring at have strammere temperaturregulering.

```
GET TEMP
Current temperature is 30.1025 deg. C.
SET TEMP 50
GET TEMP
Current temperature is 48.4277 deg. C.
GET TEMP
Current temperature is 48.7793 deg. C.
GET TEMP
Current temperature is 48.9111 deg. C.
GET TEMP
Current temperature is 49.2188 deg. C.
GET TEMP
Current temperature is 50.7568 deg. C.
GET TEMP
Current temperature is 48.999 deg. C.
GET TEMP
Current temperature is 50.0977 deg. C.
GET TEMP
Current temperature is 48.4717 deg. C.
GET TEMP
Current temperature is 49.1748 deg. C.
GET TEMP
Current temperature is 51.3721 deg. C.
GET TEMP
Current temperature is 49.0869 deg. C.
GET TEMP
Current temperature is 48.8672 deg. C.
GET TEMP
Current temperature is 50.0977 deg. C.
```

6.1 Resultater

Regulerings-algoritmen virker efter hensigten. Den er dog i nuværende konfiguration begrænset til værdiområdet ca. 28-55 °C.

Temperaturservicen virker derfor efter hensigten, at der både kan måles/afsendes temperaturdata samt modtages/reguleres op imod et temperatur-set-point.

6.2 Diskussion, forbedringsmuligheder

En række forbedringsmuligheder er identificeret i løsningen af denne opgave:

- Bedre server-arkitektur -> en concurrent-server, med nye tråde til hver ny indkommende forbindelse.
- Bedre temperaturregulering, fx med fuld PID-algoritme, samt bedre tuning af algoritmens parametre.
- Bedre kommunikationsmekanisme mellem tråde end ”rå” ubeskyttede globale variable:
    - Fx: atomic (C++) eller klassisk mutex / beskyttet queue (til flere elementer).
- Bedre protokol til kommunikation med service. Kan fx forbedres ved brug af JSON (JSONRPC).
- Bedre beskyttelse af ADC’en, fx med clamping-dioder.
- Udforske, hvordan regulatorens værdiområde kan øges (nok mest muligt opad i temperatur ☺).

7 Samlet konklusion

Der er i denne opgave implementeret et system, hvor en server agerer temperaturservice for en klient. Kommunikationen sker ved tekstbeskeder over stream-sockets (TCP / INET-domæne).

Vha. en LM35-sensor kan temperaturen i omgivelserne måles, og vha. en anden transducer (PWM-styret N-type MOSFET + 100 Ω-modstand) kan temperaturen i omgivelserne styres op og ned.

En regulator sikrer, at set-pointet overholdes (inden for det mulige interval på ca. 28-55 °C). Regulatoren giver i sin nuværende implementering lidt oscillationer omkring set-point. Det forventes, at disse kunne fjernes ved yderligere tuning af algoritmens paramatre.

8 Referencer

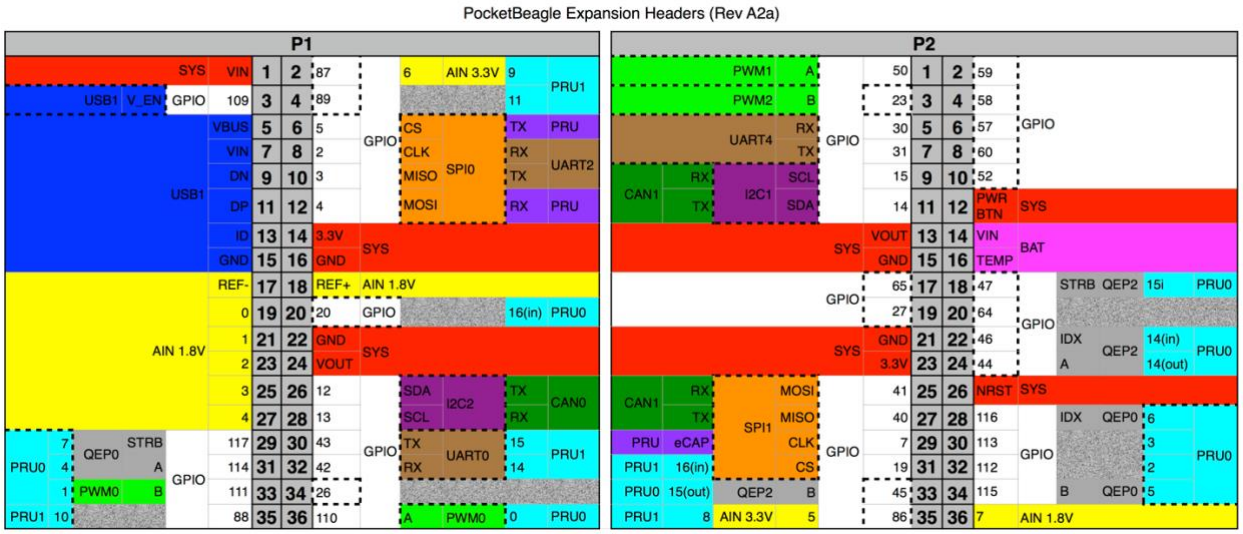
[1] MOLLOY, DEREK. *Exploring BeagleBone*. 2. udgave. Wiley, 2019.

[2] Texas Instruments. LM35 Precision Centigrade Temperature Sensors. Tilgængelig online (sidst set 29/03/2020). URL:
http://www.ti.com/lit/ds/symlink/lm35.pdf

9 Bilag

9.1 Pin-outs på Pocketbeagle

Nedenfor ses pin-outs.



Issues I Rev. A2:

https://github.com/beagleboard/pocketbeagle/wiki/System-Reference-Manual#223_Rev_A2

Power pins:

https://github.com/beagleboard/pocketbeagle/wiki/System-Reference-Manual#54_Power