

INDLEJRET SYSTEMDESIGN 2 EKSAMEN

E4ISD2

Sommer 2020

Janus Bo Andersen (JA67494)

EMNER 2 X 10-12 MIN. + VOTERING

Software (KKO)

1. Generel Linux-programmering
2. Distribueret programmering og IoT
3. Distribuerede systemer: Procesperspektivet
4. Distribuerede systemer: Ressourceperspektivet
5. Distribuerede systemer: Netværks-/Kommunikationsperspektivet
6. Distribuerede systemer: Arkitekturperspektivet

Hardware (SMM)

1. CAN-bus
2. FreeRTOS
3. Co-design og systemarkitektur

SW1: GENEREL LINUX-PROGRAMMERING

Indhold

1. Overblik
2. Gennemgang af:
 1. sigaction/signal
 2. syslog
 3. dæmonisering
3. Eksempel fra distribueret system
(øvelse), og illustration af:
 1. Tråde
 2. Sockets
4. Opsummering

Spørgsmål

Forklar (formål og virkemåde) og vis dit egen kode, som afvikles som en daemon

- forklar sigaction,
- dæmoniseringen,
- syslog,
- tråde og
- sockets.

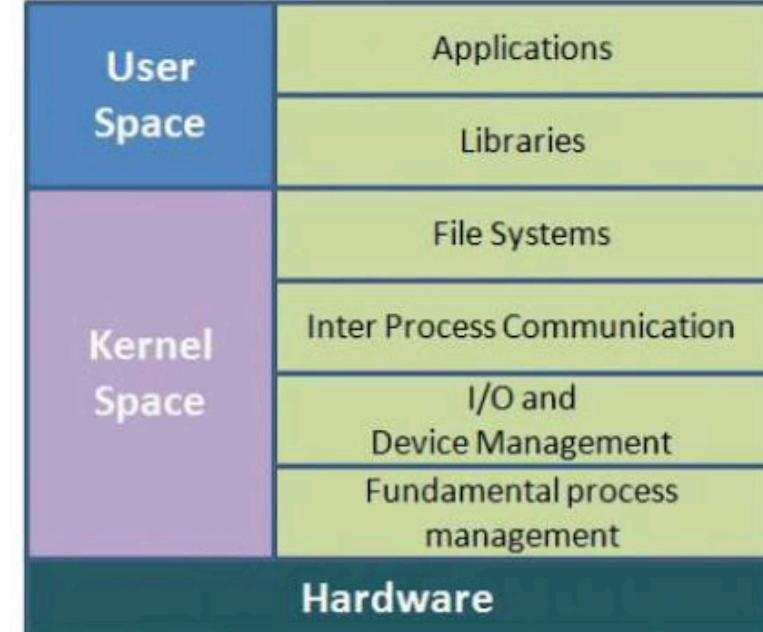
GENEREL LINUX-PROGRAMMERING

OVERBLIK

Formål: Design og udvikling af distribuerede systemer på Linux (gælder ofte også på POSIX-systemer)

Systemprogrammering: Benytte funktionalitet som kernen stiller til rådighed

- Signaler <- responsive applikationer, software interrupts
 - sigaction, signal
- Processer, tråde <- programafvikling og scheduling
 - daemonization, pthreads, syslog
- IPC, netværk <- kommunikation
 - sockets
- Anden I/O



Korrekt brug giver **portable** og **pålidelige** distribuerede applikationer.

SIGACTION

Formål: Opnå responsive systemer, kontrollere og kommunikere med processer

- Software interrupts ved forskellige begivenheder
- Sende signaler til daemon-program, fx
 - SIGTERM, SIGHUP, OSV.

Systemkald `sigaction()`:

- Robust, permanent installeret signal-handler
 - Modsat `signal()`.
- POSIX-standard.

Signal-handler (software interrupt handler) skal:

- Være re-entrant (~thread safe), så pas på med
 - Fælles hukommelse (globale, static, osv.)
 - buffered I/O, stdio.
 - Ikke-atomiske operationer
- Ellers risiko for utilsigtede konsekvenser.

```
1 #include <iostream>
2 #include <csignal>
3 #include <unistd.h>
4
5 void sig_handler(int sig)
6 {
7     if (sig == SIGHUP) {
8         write(STDOUT_FILENO, buf: "Received HANGUP.\n", n: 17);
9         exit(EXIT_SUCCESS);
10    }
11 }
12
13 int main() {
14
15     struct sigaction act; // struct for setup
16     act.sa_handler = sig_handler; // set handler
17     sigemptyset(&act.sa_mask); // Don't ignore any signals
18     act.sa_flags = 0; // No flags
19
20     if (sigaction(SIGHUP, &act, oact: 0) < 0) { // Register HUP signal
21         std::cout << "Could not register sigaction" << std::endl;
22         exit(EXIT_FAILURE);
23     }
24
25     while (1) {
26         std::cout << "Working" << std::endl;
27         sleep(seconds: 2);
28     }
29
30 }
```

The `sigaction` structure is defined as something like:

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
};
```

```
#include <signal.h>
int sigaction(int sig, const struct sigaction *act, struct sigaction *oact);
```

Returner 0 hvis OK
-1 hvis fejl.

Signal, der skal
reageres på

Action der skal
installeres for signal

Overskriv evt.
gammel action

TEST AF SIGACTION

Start proces

Afslut proces med SIGHUP

Kan også afsluttes med bl.a.

- INT (2)
- QUIT (3)
- KILL (9)
- TERM (15)

Test af signalhandler med SIGHUP

```
janus@janus-vm:~/projects/e4isd2/u6_sigaction/cmake-build-debug$ ./u6_sigaction
Working
Working
Working
```

```
0 S janus      14199  3275  0  80   0 - 1479 hrtime 18:45 pts/0    00:00:00 ./u6_sigaction
4 R janus      14200 13877  0  80   0 - 2862 -      18:45 pts/1    00:00:00 ps -elf
janus@janus-vm:~/projects/e4isd2/u6_sigaction/cmake-build-debug$ kill -1 14199
```

```
Working
Received HANGUP.
janus@janus-vm:~/projects/e4isd2/u6_sigaction/cmake-build-debug$ █
```

SIGNAL

Væsentligt simplere

- Til simple use-cases (one time), bare pas på:
 - Signaler installeres muligvis ikke permanent
 - Utilsigtede konsekvenser
- Program og test
 - Fang SIGINT, men afslut ikke.

Program og test

```
1  /*
2   * @author Janus Bo Andersen
3   * @brief C++ example to register a signal handler and catch a signal
4   */
5
6 #include <iostream>
7 #include <csignal>
8 #include <zconf.h>
9
10 void sig_handler(int signo) {
11
12     // React to caught signal
13     if (signo == SIGINT) {
14         std::cout << "Received SIGINT" << std::endl;
15     }
16 }
17
18
19 int main() {
20     std::cout << "Registering a signal handler. Try Ctrl+C!" << std::endl;
21
22     //attempt tp register the signal handler to catch SIGINT
23     //signal returns SIG_ERR if this cannot be registered
24     if ( signal(SIGINT, sig_handler) == SIG_ERR ) {
25         std::cout << "Can't register SIGINT" << std::endl;
26     }
27
28     // Infinite loop while you try to press ctrl+c
29     while(1) {
30         sleep(seconds: 1);
31     }
32
33     return 0;
34 }
```

```
janus@janus-vm:~/projects/e4isd2/u5_signal/cmake-build-debug$ ./u5_signal
Registering a signal handler. Try Ctrl+C!
^CReceived SIGINT
^CReceived SIGINT
```

```
12966 pts/0    S+    0:00 ./u5_signal
12975 pts/1    Ss    0:00 bash
12983 pts/1    R+    0:00 ps a
janus@janus-vm:~/projects/e4isd2/u5_signal/cmake-build-debug$ kill -9 12966
```

SYSLOG

Formål: Skrive og logge lokale systembegivenheder (events)

- **Facility:** Repræsenterer processen, der rapporterer begivenhed.
- **Priority:**
 - LOG_NOTICE er 5,
 - 0 er højest
- Rapporteres som "facility/priority", vha. makro LOG_MAKEPRI, som er `(fac) | (pri)`.
- Kan benytte setlogmask til at ignorere lav-prio, logge op til bestemte niveauer, osv.

Anvendelser:

- Vores services og applikationer
- Systemet selv, drivers, hardware
- syslogd <- sende log-beskeder til andre systemer via UDP-sockets

Important	Program started by User 1000
All	Failed to load module "c tmp-snap.rootfs_1xYRTO.m Sender u5_syslog Time 13:29:14 audit: type=1400 audit(1 Message Program started by User 1000 Audit Session 3 Priority 5
Applications	audit: type=1400 audit(1 kauditd_printk_skb: 6 ca AVC apparmor="DENIED" op Started flatpak document
System	[session uid=1000 pid=13
Security	

Begivenhed kan ses i systemets log
Lidt andet format
Bemærk prio

```
1  /*  
2   * @file    main.cpp  
3   * @author  Janus Bo Andersen  
4   * @brief   Syslog code for E4ISD2 week 5  
5   *  
6  */  
7  
8  #include <iostream>  
9  #include <sys/syslog.h>  
10 #include <unistd.h>  
11  
12 int main() {  
13     std::cout << "Writing an error to syslog. See it in /var/log/syslog" << std::endl;  
14  
15     //Open the log and give an identity  
16     openlog(ident: "e4isd2wk5",  
17             option: LOG_CONS | LOG_PID | LOG_NDELAY, //options  
18             facility: LOG_LOCAL0); //facility  
19  
20     //Log a system event  
21     syslog(pri: LOG_MAKEPRI(LOG_LOCAL0, LOG_NOTICE), fmt: "Program started by User %d", getuid());  
22  
23     //Close the connection  
24     closelog();  
25  
26     //Report to syslog, without opening a connection first.  
27     // Facility: LOG_LOCAL0, Severity: Error. Returns void.  
28     /* syslog(LOG_MAKEPRI(LOG_LOCAL0, LOG_ERR),  
29            "Hello this is an error");  
30     */  
31  
32     return 0;  
33 }
```

Writing an error to syslog. See it in /var/log/syslog
Process finished with exit code 0

Indstillinger
LOG_CONS: Skriver til konsol, hvis /dev/log-socket ikke kan åbnes
LOG_PID: Skriver proces-ID i loggen
LOG_NDELAY: Socket til loggen åbnes med det samme.

Priority
Kombination af Facility og Severity

Begivenhed kan læses i sysloggen
Host, identitet, PID, besked

DAEMONIZATION (DÆMONISERING)

Formål: Gøre en proces til en Linux/UNIX-daemon

- Services, baggrundsprocesser
- Ingen terminal, så
 - Kontrol vha. signaler
 - Output og fejl skrives til syslog

Metode:

- **Installér sighandler**
- Fork parent-proces
- Luk parent-proces
- Indstil daemon:
 - Ændr file mode mask (umask)
 - Åbn syslog
 - Nyt sessions-ID (sid) til procesgruppe for child
 - Skift workdir
 - Luk file descriptors
- Afvikling af daemon/service

Del 1: Installer sighandler

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <csignal>
4  #include <sys/stat.h>
5  #include <unistd.h>
6  #include <sys/syslog.h>
7
8  // signal handler reacts to hangup signal
9  void sig_handler(int signo) {
10     if (signo == SIGHUP) {
11         syslog( pri: LOG_MAKEPRN(LOG_LOCAL0, LOG_NOTICE), fmt: "Daemon received SIGHUP. Stopping.");
12         closelog();
13         exit(EXIT_SUCCESS);
14     }
15 }
16
17 ▶ int main(void) {
18
19     // install signal handler
20     if ( signal(SIGHUP, sig_handler) == SIG_ERR) {
21         std::cout << "Failed to register SIGHUP" << std::endl;
22     }
23
24     std::cout << "Starting Daemonization!" << std::endl;
```

DAEMONIZATION (DÆMONISERING)

Formål: Gøre en proces til en Linux/UNIX-daemon

- Services, baggrundsprocesser
- Ingen terminal, så
 - Kontrol vha. signaler
 - Output og fejl skrives til syslog

Metode:

- Installér sighandler
- **Fork parent-proces**
- **Luk parent-proces**
- **Indstil daemon:**
 - **Ændr file mode mask (umask)**
 - **Åbn syslog**
 - Nyt sessions-ID (sid) til procesgruppe for child
 - Skift workdir
 - Luk file descriptors
- Afvikling af daemon/service

Del 2: Forking og åbn syslog

```
24     std::cout << "Starting Daemonization!" << std::endl;
25
26     pid_t pid, sid;                                // Process ID and Session ID
27
28     pid = fork();                                  // Fork to make copied process
29     if (pid < 0) {
30         exit(EXIT_FAILURE);                      // Failed to fork
31     }
32
33     if (pid > 0) {                                // pid in child is 0
34         exit(EXIT_SUCCESS);                      // Exit parent process, not needed
35     }
36
37     umask( mask: 0);                            // rwx permission on file creation
38
39     // Open log
40     openlog( ident: "daemon-proc",
41               option: LOG_CONS | LOG_PID | LOG_NDELAY,
42               facility: LOG_LOCAL0);
43
```

DAEMONIZATION (DÆMONISERING)

Formål: Gøre en proces til en Linux/UNIX-daemon

- Services, baggrundsprocesser
- Ingen terminal, så
 - Kontrol vha. signaler
 - Output og fejl skrives til syslog

Metode:

- Installér sighandler
- Fork parent-proces
- Luk parent-proces
- Indstil daemon:
 - Ændr file mode mask (umask)
 - Åbn syslog
 - **Nyt sessions-ID (sid) til procesgruppe for child**
 - **Skift workdir**
 - **Luk file descriptors**
- **Afvikling af daemon/service**

Del 3: Sessions-ID og afvikling af arbejde

```
44     sid = setsid();           // new session id
45     if (sid < 0) {
46         syslog( pri: LOG_MAKEPRI(LOG_LOCAL0, LOG_NOTICE), fmt: "Daemon failed to get sid.");
47         exit(EXIT_FAILURE); // failed to sid
48     } else {
49         syslog( pri: LOG_MAKEPRI(LOG_LOCAL0, LOG_NOTICE), fmt: "Daemon got sid: %d.", sid);
50     }
51
52     if ((chdir( path: "/" ) < 0) { // change path if creating files
53         syslog( pri: LOG_MAKEPRI(LOG_LOCAL0, LOG_NOTICE), fmt: "Daemon failed to chdir.");
54         exit(EXIT_FAILURE);
55     }
56
57     // Remove access to terminal file descriptors
58     close(STDIN_FILENO);
59     close(STDOUT_FILENO);
60     close(STDERR_FILENO);
61
62     // Do daemon work
63     syslog( pri: LOG_MAKEPRI(LOG_LOCAL0, LOG_NOTICE), fmt: "Daemon starting work.");
64     while (true) {
65         sleep( seconds: 30 );
66     }
}
```

TEST OG KONTROL OVER DAEMON

Test:

- Start daemon
 - Bekræft
- Test SIGHUP
- Dræb daemon
 - Bekræft

Øvrigt:

- Daemon har et proc-interface som alle andre processer:
 - Kig i proc/pid/-mappen og se evt. fd'er, cmdline, osv.

Test og kontrol

Daemon skriver til syslog

Processen eksisterer med det nye sid (1331 er systemd)

Daemon findes som proces i /proc/, med alt tilhørende

Daemon stoppes med SIGHUP

Daemon rapporter software interrupt til syslog

Daemon er død

```
janus@janus-vm:~/projects/e4isd2/u6_daemonize/cmake-build-debug$ ./u6_daemonize
Starting Daemonization!
janus@janus-vm:~/projects/e4isd2/u6_daemonize/cmake-build-debug$ tail -n 2 /var/log/syslog
Jun 25 16:54:43 janus-vm daemon-proc[12906]: Daemon got sid: 12906.
Jun 25 16:54:43 janus-vm daemon-proc[12906]: Daemon starting work.
janus@janus-vm:~/projects/e4isd2/u6_daemonize/cmake-build-debug$ ps -ef | grep "12906"
janus 12906 1331 0 16:54 ? 00:00:00 ./u6_daemonize
janus 12910 3275 0 16:54 pts/0 00:00:00 grep --color=auto 12906
janus@janus-vm:~/projects/e4isd2/u6_daemonize/cmake-build-debug$ ll /proc/ | grep "12906"
dr-xr-xr-x 9 janus janus 0 Jun 25 16:54 12906/
janus@janus-vm:~/projects/e4isd2/u6_daemonize/cmake-build-debug$ kill -1 12906
janus@janus-vm:~/projects/e4isd2/u6_daemonize/cmake-build-debug$ tail -n 2 /var/log/syslog
Jun 25 16:54:43 janus-vm daemon-proc[12906]: Daemon starting work.
Jun 25 16:55:32 janus-vm daemon-proc[12906]: Daemon received SIGHUP. Stopping.
janus@janus-vm:~/projects/e4isd2/u6_daemonize/cmake-build-debug$ ll /proc/ | grep "12906"
janus@janus-vm:~/projects/e4isd2/u6_daemonize/cmake-build-debug$ ps -ef | grep "12906"
janus 12919 3275 0 16:55 pts/0 00:00:00 grep --color=auto 12906
```

SOCKETS OG TRÅDE

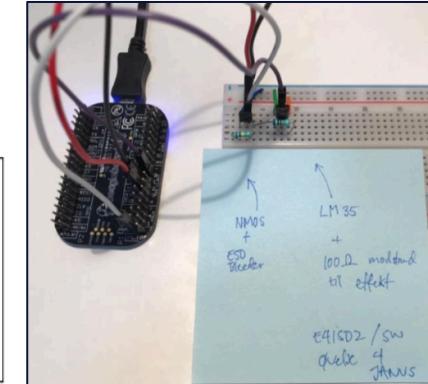
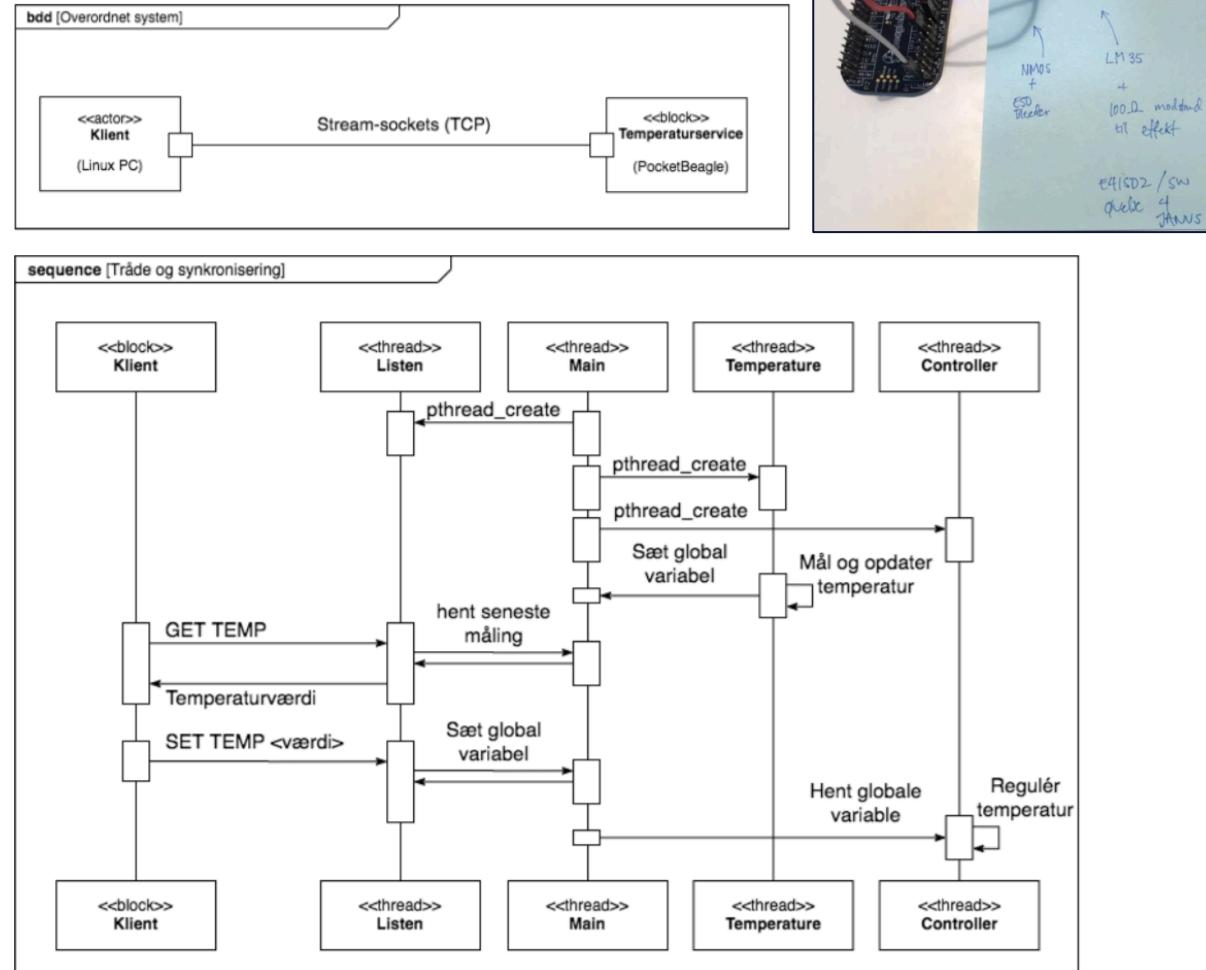
EKSEMPEL FRA TEMP SERVER

Temperaturserver

- Klient-server-arkitektur
 - Server på Beaglebone
 - LM35-temperatursensor
 - TCP-sockets
 - Trådet applikation

Formål med brug af sockets og tråde i arkitekturen:

- Sockets: Kommunikere over netværk (TCP)
- Tråde: Opdele arbejde med forskellige tidsconstraints og I/O-typer.
 - Listen-tråden blokerer på `listen()` og på `read()`/`recv()`.
 - Temperatursensor aflæses med interval.
 - Controller håndterer PI-regulator med interval.



TRÅDE

Opstart og nedlukning af tråde

```
58 int main(void) {
59
60     /* Set up signal handler */
61     if ( signal(SIGHUP, sig_handler) == SIG_ERR ) {
62         std::cout << "Can't register SIGHUP" << std::endl;
63     }
64
65     pthread_t listen_thread;
66     if ( pthread_create(&listen_thread, NULL, &listen_thread_function, NULL) ) {
67         std::cout << "Could not create thread" << std::endl;
68         return EXIT_FAILURE;
69     }
70
71     pthread_t temp_thread;
72     if ( pthread_create(&temp_thread, NULL, &temp_monitor_thread_function, NULL) ) {
73         std::cout << "Could not create thread" << std::endl;
74         return EXIT_FAILURE;
75     }
76
77     pthread_t control_thread;
78     if ( pthread_create(&control_thread, NULL, &controller_thread_function, NULL) ) {
79         std::cout << "Could not create thread" << std::endl;
80         return EXIT_FAILURE;
81     }
82
83     /* Temperature thread has been stopped by SIGHUP */
84     pthread_join(temp_thread, NULL);
85
86     /* Controller thread has been stopped by SIGHUP */
87     pthread_join(control_thread, NULL);
88
89     /* Safe to kill the listener now */
90     pthread_cancel(listen_thread);
91     pthread_join(listen_thread, NULL);
92
93     return 0;
94 }
```

Benytter POSIX-tråde (pthreads)

Opstart af trådfunktioner

Tråde afkobles ikke fra hoved-tråden,
men synkroniseres ved nedlukning.

Når regulator-tråden er død, kan
resten lukkes ned.

Synkronisering af variable og nedlukning

```
44 /* Coordinate clean exit : */
45 std::atomic<bool> exit_now(false);
46
47 void sig_handler(int signo) {
48
49     // React to caught signal
50     if (signo == SIGHUP) {
51         std::cout << "Received SIGHUP. Bye!" << std::endl;
52
53         // Tell all threads to end now, cleanly
54         exit_now.store(true);
55     }
56 }
21 /* These enable communication between threads */
22 extern float cur_temp;
23 extern float set_point;
24 extern std::atomic<bool> exit_now;
25
26 /* @brief Thread function to monitor temperature in the background
27 * update the global variable CUR_TMP
28 */
29 void * temp_monitor_thread_function(void * value) {
30
31     int adc_value;
32     float temp;
33
34     while( !exit_now.load() ) {
35
36         // Take a sample and convert to temperature
37         adc_value = read_analog(ADC);
38         temp = conv_temperature(adc_value);
39
40         // Update the global variable
41         cur_temp = temp;
42
43         // Sleep until next sample - this is also a thread cancellation point
44         sleep(TEMP_SAMPLE_INTERVAL);
45     }
46
47     pthread_exit(NULL);
48 }
```

Atomisk variabel til
koordinere nedlukning.

Globale variable, header fil.
Her kunne overvejes mutex eller
en atomisk variabel.

Tjek for nedlukning.

Opdatér global variabel.

SOCKETS

SocketServer – klasse til serverfunktion

```
16@/**  
17 * @class SocketServer  
18 * @brief A class that encapsulates a server socket for network communication  
19 */  
20@class SocketServer {  
1 private:  
22 int portNumber;  
23 int socketfd, clientSocketfd;  
24 struct sockaddr_in serverAddress;  
25 struct sockaddr_in clientAddress;  
26 bool clientConnected;  
27  
28 public:  
29 SocketServer(int portNumber);  
30 virtual int listen();  
31 virtual int send(std::string message);  
32 virtual std::string receive(int size);  
33 virtual void closeClient();  
34  
35 virtual ~SocketServer();  
36 };
```

Objekt instantieres med portnummer

Wrappers til 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 }  
83  
84@string SocketServer::receive(int size=1024){  
85 char readBuffer[size];  
86 memset(readBuffer, '\0', size);  
87 int n = read(this->clientSocketfd, readBuffer, sizeof(readBuffer));  
88 if (n < 0){  
89 perror("Socket Server: error reading from server socket.");  
90 throw "Socket probably closed.";  
91 }  
92 return string(readBuffer);  
93 }
```

Wrapper de underliggende C-kald, og oversetter til C-strukturer

26.JUNI 2020

Opsætning og indkommende requests (listen())

```
25@ /* listen creates the socket and  
26 * blocks until an incoming connection established,  
27 * and then accepts the incoming connection */  
28@ int SocketServer::listen(){  
29  
30 /* Make INET (TCP) socket */  
31 this->socketfd = socket(AF_INET, SOCK_STREAM, 0);  
32  
33 /* Check that socket created OK */  
34 if (this->socketfd < 0){  
35 perror("Socket Server: error opening socket.\n");  
36 return 1;  
37 }  
38  
39@ /* Ensure all zeros in unused part of serverAddress  
40 * this is an alternative to memset */  
41 bzero((char *) &serverAddress, sizeof(serverAddress));  
42  
43 /* Create struct for address */  
44 serverAddress.sin_family = AF_INET;  
45 serverAddress.sin_addr.s_addr = INADDR_ANY;  
46 serverAddress.sin_port = htons(this->portNumber);  
47  
48 /* Attempt to bind to the socket address */  
49 if (bind(socketfd, (struct sockaddr *) &serverAddress, sizeof(serverAddress)) < 0) {  
50 perror("Socket Server: error on binding the socket.\n");  
51 return 1;  
52 }  
53  
54 /* system call listen */  
55 ::listen(this->socketfd, 5);  
56  
57 /* Update address of client */  
58 socklen_t clientLength = sizeof(this->clientAddress);  
59  
60 /* Accept connection */  
61 this->clientSocketfd = accept(this->socketfd,  
62 (struct sockaddr *) &this->clientAddress,  
63 &clientLength);  
64  
65 /* Check that client socket is OK */  
66 if (this->clientSocketfd < 0){  
67 perror("Socket Server: Failed to bind the client socket properly.\n");  
68 return 1;  
69 }  
70  
71 }
```

Ingen specifik IP-adresse

Sat i headerfil, men kunne også være argument

Backlog, tillader kø på 5 ventende forbindelser

OPSUMMERING OG KONKLUSION

Generel Linux-programmering

Vi har set på alle ingredienserne til distribuerede systemer

- Proceshåndtering
- Oprettelse af dæmoner / services der kører i baggrunden
- Signaler til at håndtere begivenheder
- Syslog til logge begivenheder
- Tråde til afkobling af forskellige opgaver
- Sockets til kommunikation

SW2: DISTRIBUERET PROGRAMMERING OG IOT

Indhold

1. Overblik
2. Eksempler
 1. Klient-server-arkitektur, og struktureret kommunikation
 2. Mere kompliceret system fra PRO4
3. Datalagring og -adgang
4. Realtidshåndtering
5. Opsummering og konklusion

Forklar hvordan distribueret programmering, f.eks. en IoT enhed, kan konstrueres på Linux

Fokuser f.eks. på:

- klient - server konstruktion
- Realtidshåndtering
- Dataopbevaring
- Struktureret kommunikation mellem klient og server

DISTRIBUERET PROGRAMMERING OG IOT

OVERBLIK

Definitioner

Distribueret system: Er et system, hvor...

- *Systemets samlede funktionalitet (forretningslogik) er distribueret ud på flere netværks forbundne enheder.*
- *Ressourcer benyttet af systemet er delt ud over flere computere, der kommunikerer over netværk*

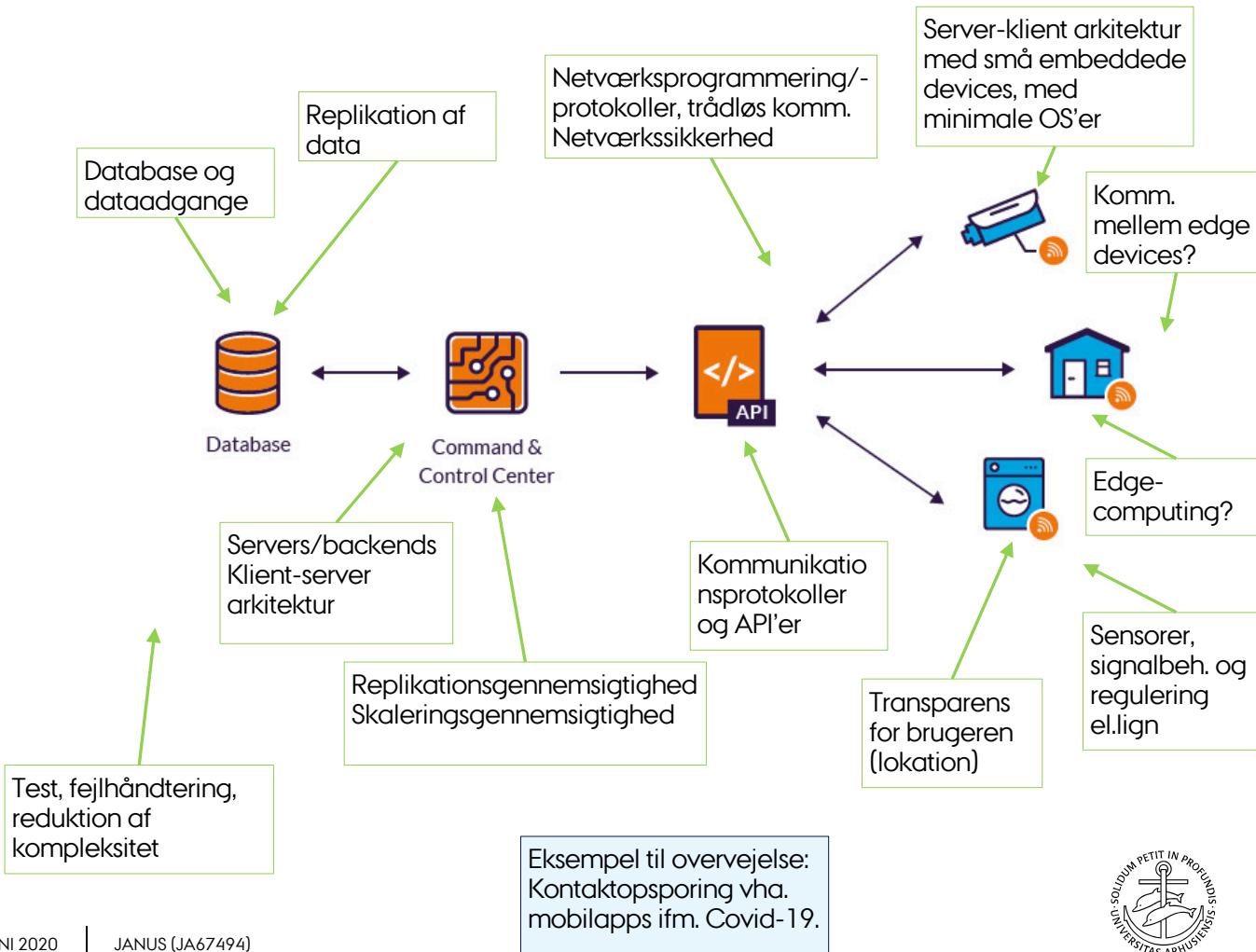
IOT: Internet of Things er et koncept, hvor...

- *Devices er embeddede i hverdagsting, og forbinder og kommunikerer over internettet*
- Cloud computing

IIOT og Industry 4.0: Industrial IOT

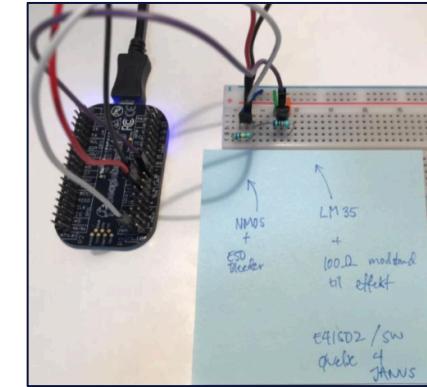
- Forbundne industrienheder, produktionssystemer (trådløst, 5G)
- Fog computing (tættere på edge)

Hvordan relaterer IOT til distribuerede systemer?



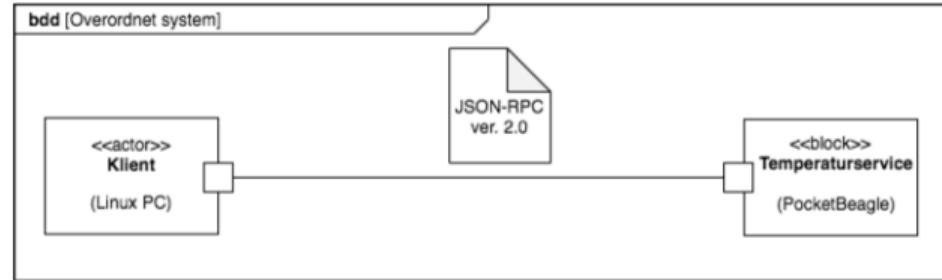
Klient-Server-Konstruktion

EKSEMPEL FRA TEMP SERVER MED RPC



Temperaturserver

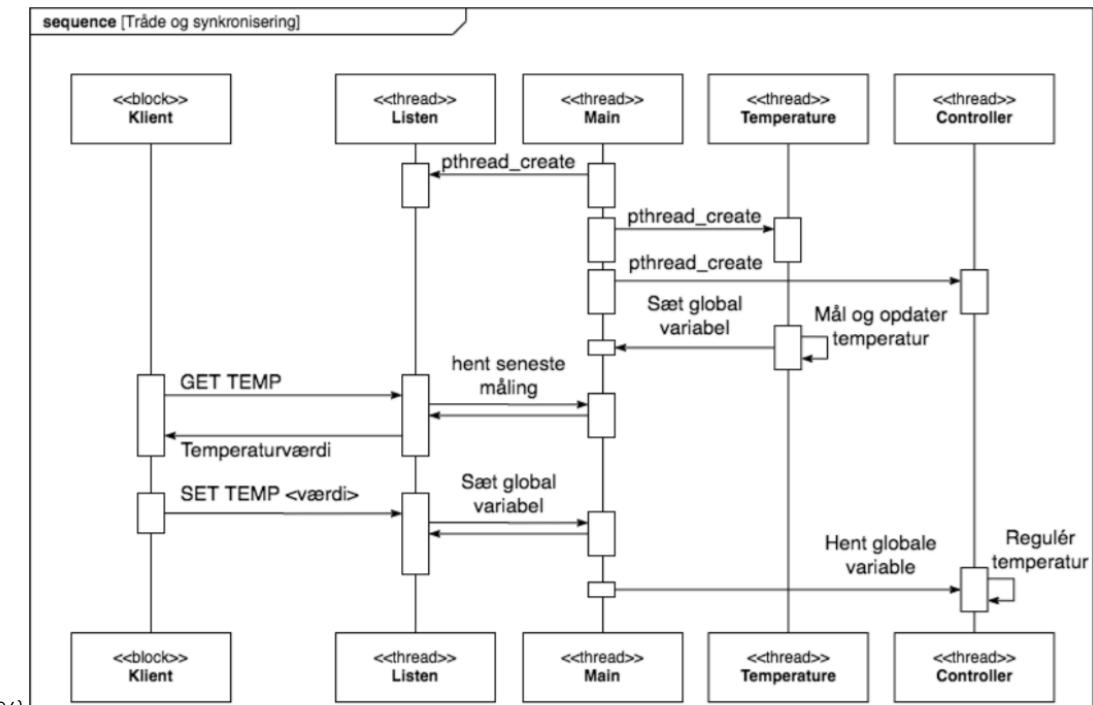
- Klient-server-arkitektur
 - Server på Beaglebone
 - LM35-temperatursensor
 - Protokol: JSON-RPC over TCP-sockets
 - Trådet applikation



Formål med JSON-RPC: Struktureret kommunikation, lettere at parse syntax / grammatik for kommandoer og svar

Formål med brug af sockets og tråde i arkitekturen:

- Sockets: Kommunikere over netværk (TCP)
- Tråde: Opdele arbejde med forskellige tidsconstraints og I/O-typer.
 - Listen-tråden blokerer på `listen()` og på `read()`/`recv()`.
 - Temperatursensor aflæses med interval.
 - Controller håndterer PI-regulator med interval.



EKSEMPEL 1: STRUKTURERET KOMMUNIKATION MELLEM KLIENT OG SERVER

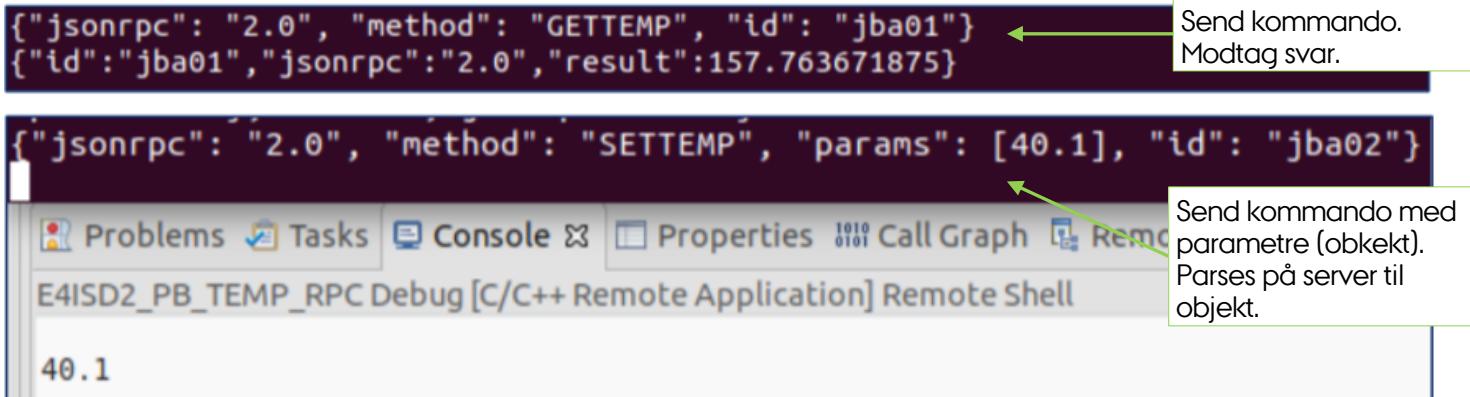
Specifikation: JSON-RPC 2.0

- Specifierer gyldige (og ugyldige) beskeder klient <-> server
- Kan indlejre objekter
- Relativt simpel, parses med gængse JSON-parsere

Andre metoder til at strukturere kommunikation

- REST (stateless)
- SOAP over HTTP, osv
- XML

Eksempel

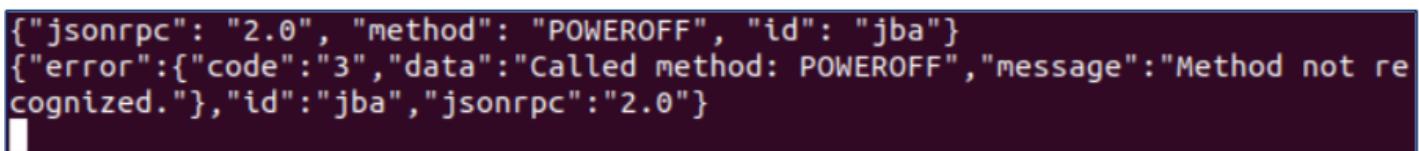


The screenshot shows a C/C++ IDE interface with several tabs: Problems, Tasks, Console, Properties, Call Graph, and Remote. The Remote tab is active, displaying the text "E4ISD2_PB_TEMP_RPC Debug [C/C++ Remote Application] Remote Shell". In the console area, three JSON-RPC messages are shown:

```
{"jsonrpc": "2.0", "method": "GETTEMP", "id": "jba01"}  
{"id": "jba01", "jsonrpc": "2.0", "result": 157.763671875}  
  
{"jsonrpc": "2.0", "method": "SETTEMP", "params": [40.1], "id": "jba02"}  
40.1
```

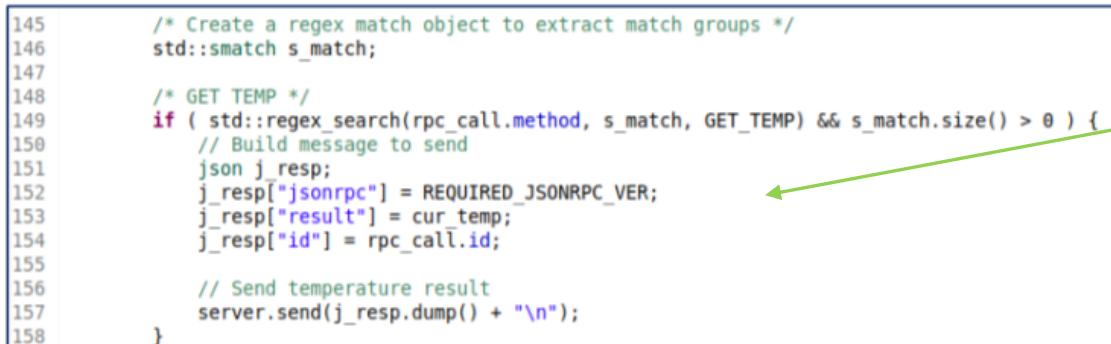
Annotations with arrows point to the messages:

- An arrow points from the first message to a callout box: "Send kommando. Modtag svar."
- An arrow points from the second message to a callout box: "Send kommando med parametre (objekt). Parses på server til objekt."



The screenshot shows a C/C++ IDE interface with the Remote tab active, displaying the text "E4ISD2_PB_TEMP_RPC Debug [C/C++ Remote Application] Remote Shell". In the console area, a JSON-RPC error message is shown:

```
{"jsonrpc": "2.0", "method": "POEROFF", "id": "jba"}  
{"error": {"code": 3, "data": "Called method: POEROFF", "message": "Method not recognized."}, "id": "jba", "jsonrpc": "2.0"}
```



The screenshot shows a C++ code editor with code related to JSON-RPC processing:

```
145     /* Create a regex match object to extract match groups */  
146     std::smatch s_match;  
147  
148     /* GET TEMP */  
149     if ( std::regex_search(rpc_call.method, s_match, GET_TEMP) && s_match.size() > 0 ) {  
150         // Build message to send  
151         json j_resp;  
152         j_resp["jsonrpc"] = REQUIRED_JSONRPC_VER;  
153         j_resp["result"] = cur_temp;  
154         j_resp["id"] = rpc_call.id;  
155  
156         // Send temperature result  
157         server.send(j_resp.dump() + "\n");  
158     }
```

Annotations with arrows point to the code:

- An arrow points from the final line of code to a callout box: "Indpakning af et svar, der overholder JSON-RPC."

EKSEMPEL 2: IOT-AGTIG LØSNING, PRO4

WEBINTERFACE FOR FYSISKE PRØVESTANDE

Containeriseret/
Docker-drevet
system

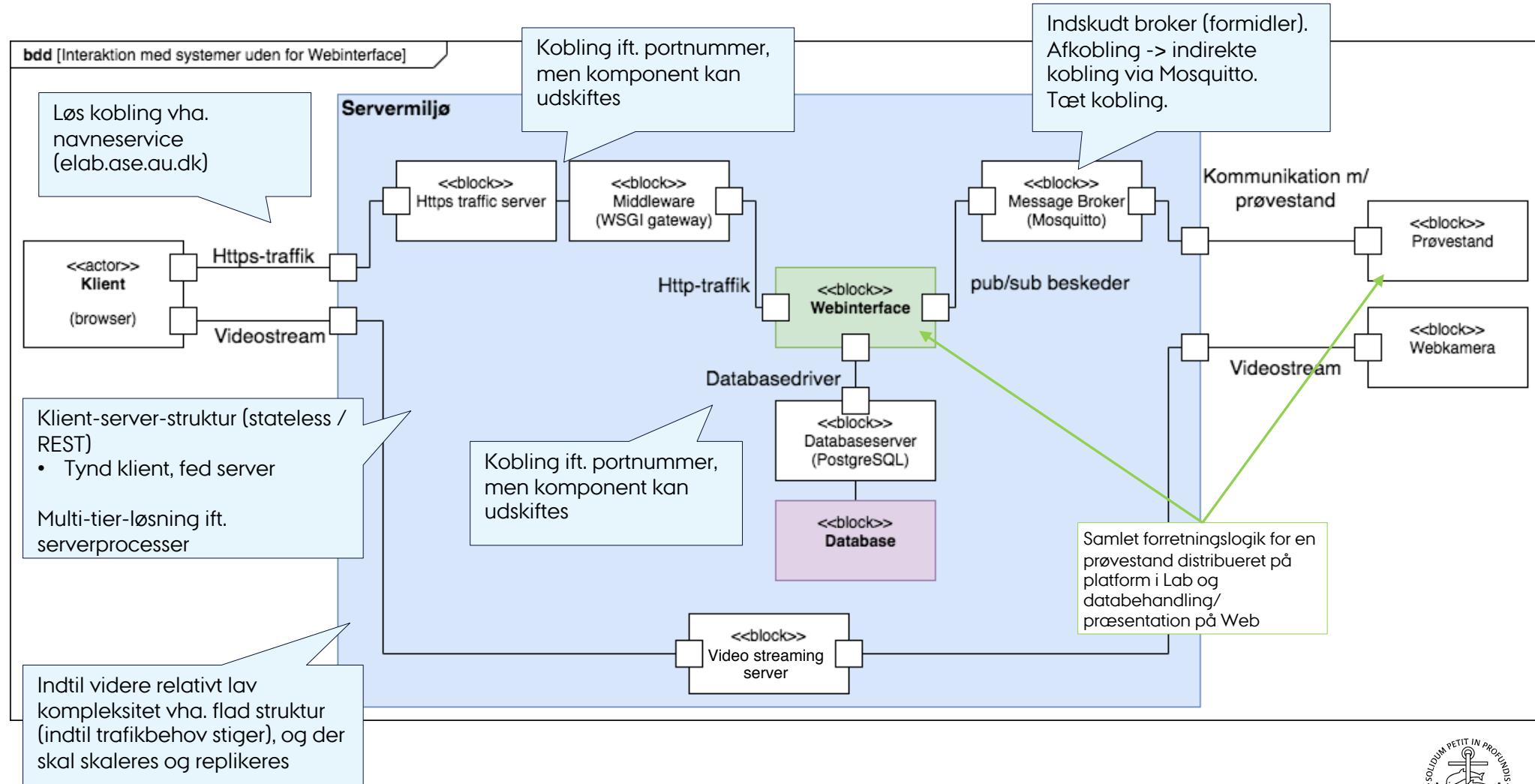
- Mikroservices

Systemet kunne
skaleres op,

Replikation af
services

Mest vanskelige at
skalere er
databasen

- Replikations-
metoder



PROTOKOL FOR KOMMUNIKATION OVER MQTT, PRO4

Mange devices med mange udviklerteams

En enkelt backend-server, et udviklerteam

Protokollen skal strukturere kommunikation til

- Overførsel af kommandoer og data

Fordel ved struktureret kommunikation og
specificeret interface

- Entydigt for udviklere, hvad der skal udvikles
op imod.
- En parser, et sæt objekter på backend,
genbrug, standardisering.
- Databasen tilpasset kommunikationen.

Version:	JSONschema_v1.1
schema	<pre>{ "\$schema": "AUTeam2 JSONschema", "title": "PayloadSchema", "type": "object", "properties": { "protocolVersion": {"type": "number"}, "sentBy": {"type": "string"}, "msgType": {"type": "string"}, "commandList": {"type": "array"}, "statusCode": {"type": "string"}, "parameterObj": {"type": "object"}, "dataObj": {"type": "object"}<br "required":="" "sentby",<br="" ["protocolversion",="" },<br=""/> "msgType", "statusCode"] }</pre>

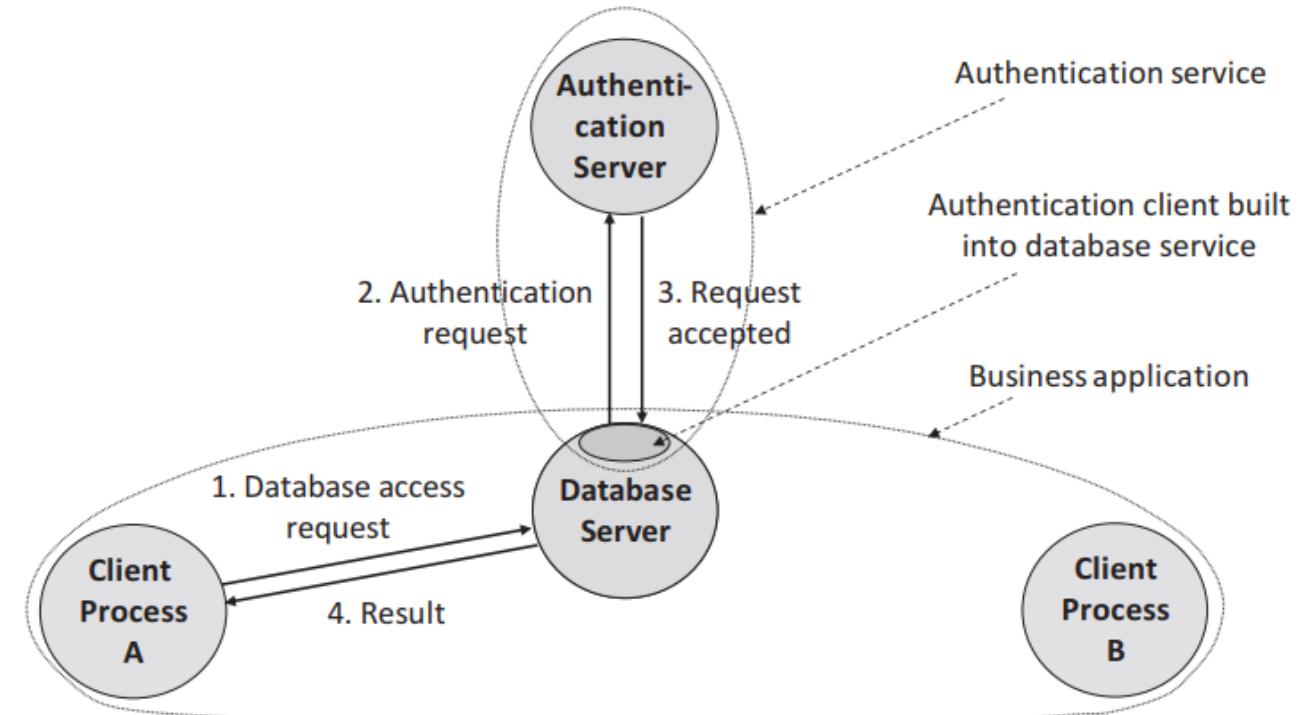
INDSKYDE AUTHENTICATION-KOMPONENTER IFT. DATAAADGANG

Centralisering af authentification

- Akkreditiver udstedes af tredjepart
- Modulært, simplere

Et eksempel er AU's WAYF

- (men det er et single-point of failure)



DATA-REPLIKATION

Hvis der opstod et behov for opskalering, replikation:

- Kan fx få brug for at flytte data og processering tættere på edge (edge computing)
- Mange prøvestande – kan få brug for mere samtidighed

Husk Replikationsgennemsigtighed og ACID-princip

- Atomicity (udelelig adgang)
- Consistency (Konsistens)
- Isolation
- Durability

Løsninger: Replikationsmodeller, transaktioner

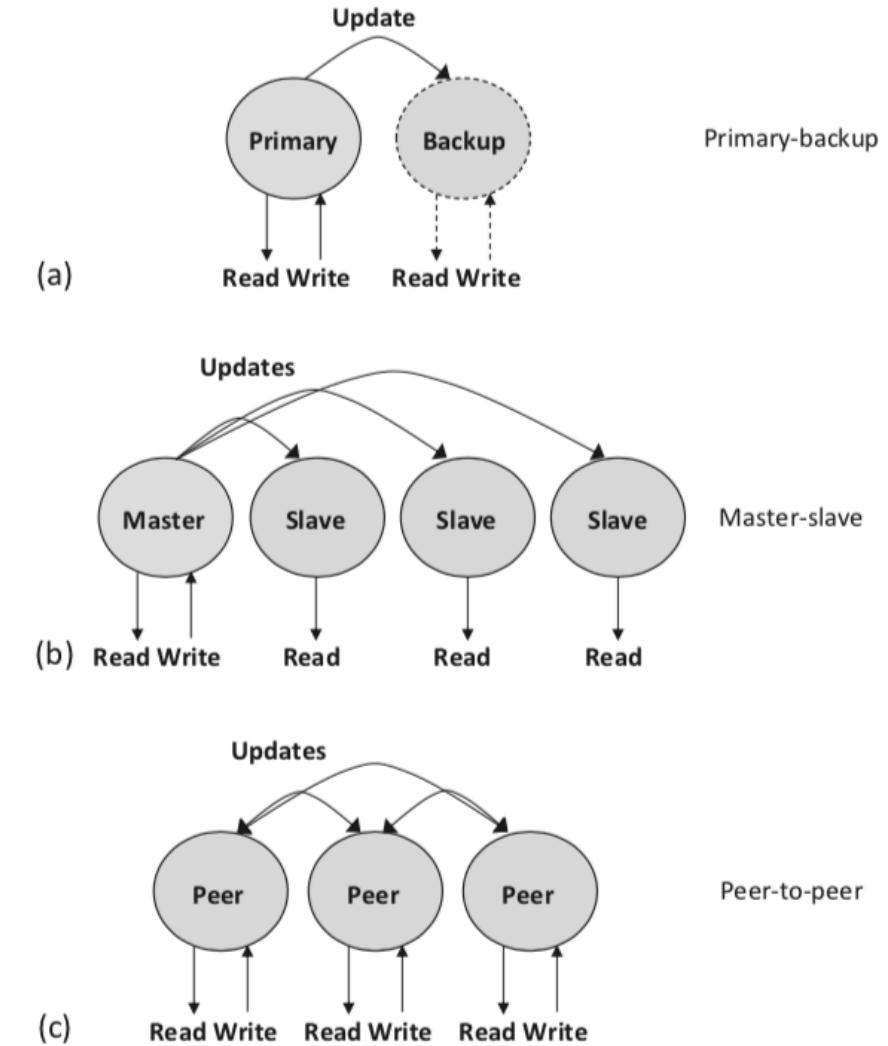


FIGURE 6.3

Some alternative models for server replication.

REALTIDSBEHOV

Realtidskrav

- Hard realtime
- Firm realtime
- Soft realtime

Hastighed på compute skal matche proces, fx ved regulering: Kan delvis løses med edge computing.

Men med **realtidskrav**:

- Når SW ikke er hurtig nok / kan stille garantier -> dedikeret hardware

Eksempel:

- 200 MHz PRU'er på BeagleBone i stedet for processering i Linux på CPU, firkantsignal ind.
- Assembler-kode på PRU, op til 5 MHz
- C-kode i Linux, op til 2-2,5 kHz
- ~ faktor 2000 forskel i frekvens, der kan håndteres

Sammenligning af performance

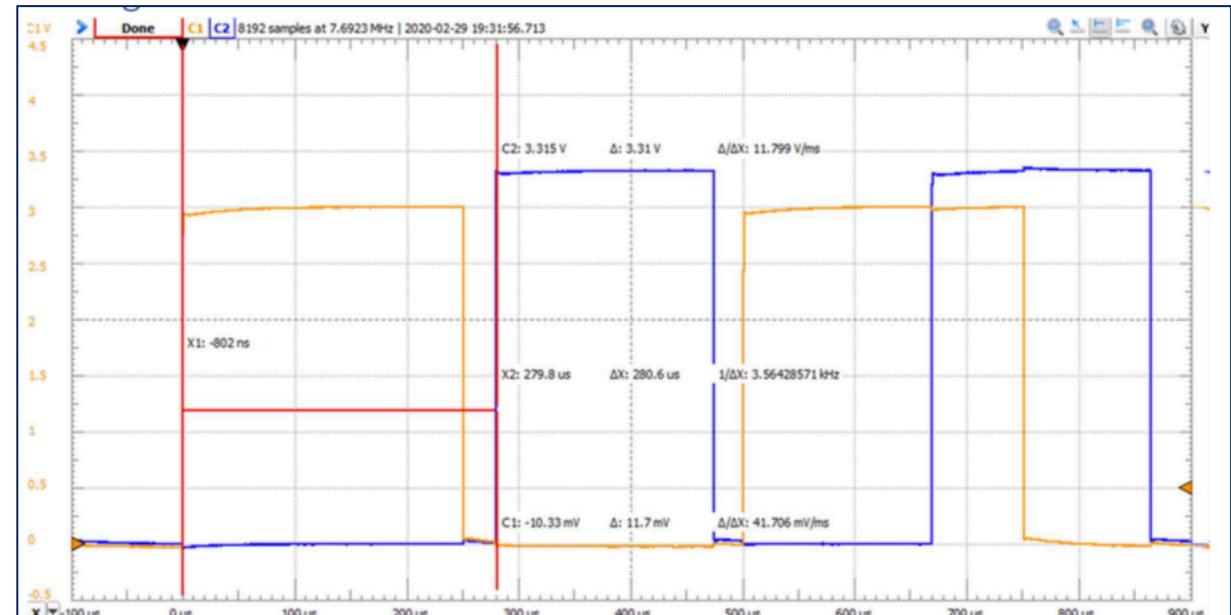
LATENCY VED 1 MHZ FIRKANTSIGNAL

Ved 1 MHz ligger latency
stadic omkring 40 ns

PRU kan stadic følge med!

I nærheden af HF-delen i RF-
båndet...

- Ikke-ideelle effekter begynder at spille ind
- Selv-induktive effekter i ledninger
- Kapacitans i breadboard, m.m.



Figur 4 - Måling af OS-styret in/output delay ved 2KHz

Ved 2Khz ses et delay på $280\mu s$.

OPSUMMERING OG KONKLUSION

BILAG TIL TRÅDE OG SOCKETS

TRÅDE

Opstart og nedlukning af tråde

```
58 int main(void) {
59
60     /* Set up signal handler */
61     if ( signal(SIGHUP, sig_handler) == SIG_ERR ) {
62         std::cout << "Can't register SIGHUP" << std::endl;
63     }
64
65     pthread_t listen_thread;
66     if ( pthread_create(&listen_thread, NULL, &listen_thread_function, NULL) ) {
67         std::cout << "Could not create thread" << std::endl;
68         return EXIT_FAILURE;
69     }
70
71     pthread_t temp_thread;
72     if ( pthread_create(&temp_thread, NULL, &temp_monitor_thread_function, NULL) ) {
73         std::cout << "Could not create thread" << std::endl;
74         return EXIT_FAILURE;
75     }
76
77     pthread_t control_thread;
78     if ( pthread_create(&control_thread, NULL, &controller_thread_function, NULL) ) {
79         std::cout << "Could not create thread" << std::endl;
80         return EXIT_FAILURE;
81     }
82
83     /* Temperature thread has been stopped by SIGHUP */
84     pthread_join(temp_thread, NULL);
85
86     /* Controller thread has been stopped by SIGHUP */
87     pthread_join(control_thread, NULL);
88
89     /* Safe to kill the listener now */
90     pthread_cancel(listen_thread);
91     pthread_join(listen_thread, NULL);
92
93     return 0;
94 }
```

Benytter POSIX-tråde (pthreads)

Opstart af trådfunktioner

Tråde afkobles ikke fra hoved-tråden,
men synkroniseres ved nedlukning.

Når regulator-tråden er død, kan
resten lukkes ned.

Synkronisering af variable og nedlukning

```
44 /* Coordinate clean exit : */
45 std::atomic<bool> exit_now(false);
46
47 void sig_handler(int signo) {
48
49     // React to caught signal
50     if (signo == SIGHUP) {
51         std::cout << "Received SIGHUP. Bye!" << std::endl;
52
53         // Tell all threads to end now, cleanly
54         exit_now.store(true);
55     }
56 }
21 /* These enable communication between threads */
22 extern float cur_temp;
23 extern float set_point;
24 extern std::atomic<bool> exit_now;
25
26 /* @brief Thread function to monitor temperature in the background
27 * update the global variable CUR_TMP
28 */
29
30 void * temp_monitor_thread_function(void * value) {
31
32     int adc_value;
33     float temp;
34
35     while( !exit_now.load() ) {
36
37         // Take a sample and convert to temperature
38         adc_value = read_analog(ADC);
39         temp = conv_temperature(adc_value);
40
41         // Update the global variable
42         cur_temp = temp;
43
44         sleep(TEMP_SAMPLE_INTERVAL);
45     }
46
47     pthread_exit(NULL);
48 }
```

Atomisk variabel til
koordinere nedlukning.

Globale variable, header fil.
Her kunne overvejes mutex eller
en atomisk variabel.

Tjek for nedlukning.

Opdatér global variabel.

SOCKETS

SocketServer – klasse til serverfunktion

```
16@/**  
17 * @class SocketServer  
18 * @brief A class that encapsulates a server socket for network communication  
19 */  
20@class SocketServer {  
1 private:  
22 int portNumber;  
23 int socketfd, clientSocketfd;  
24 struct sockaddr_in serverAddress;  
25 struct sockaddr_in clientAddress;  
26 bool clientConnected;  
27  
28 public:  
29 SocketServer(int portNumber);  
30 virtual int listen();  
31 virtual int send(std::string message);  
32 virtual std::string receive(int size);  
33 virtual void closeClient();  
34  
35 virtual ~SocketServer();  
36 };
```

Objekt instantieres med portnummer

Wrappers til 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 }  
83  
84@string SocketServer::receive(int size=1024){  
85 char readBuffer[size];  
86 memset(readBuffer, '\0', size);  
87 int n = read(this->clientSocketfd, readBuffer, sizeof(readBuffer));  
88 if (n < 0){  
89 perror("Socket Server: error reading from server socket.");  
90 throw "Socket probably closed.";  
91 }  
92 return string(readBuffer);  
93 }
```

Wrapper de underliggende C-kald, og oversetter til C-strukturer

Opsætning og indkommende requests (listen())

```
25@ /* listen creates the socket and  
26 * blocks until an incoming connection established,  
27 * and then accepts the incoming connection */  
28@ int SocketServer::listen(){  
29  
30 /* Make INET (TCP) socket */  
31 this->socketfd = socket(AF_INET, SOCK_STREAM, 0);  
32  
33 /* Check that socket created OK */  
34 if (this->socketfd < 0){  
35 perror("Socket Server: error opening socket.\n");  
36 return 1;  
37 }  
38  
39@ /* Ensure all zeros in unused part of serverAddress  
40 * this is an alternative to memset */  
41 bzero((char *) &serverAddress, sizeof(serverAddress));  
42  
43 /* Create struct for address */  
44 serverAddress.sin_family = AF_INET;  
45 serverAddress.sin_addr.s_addr = INADDR_ANY;  
46 serverAddress.sin_port = htons(this->portNumber);  
47  
48 /* Attempt to bind to the socket address */  
49 if (bind(socketfd, (struct sockaddr *) &serverAddress, sizeof(serverAddress)) < 0) {  
50 perror("Socket Server: error on binding the socket.\n");  
51 return 1;  
52 }  
53  
54 /* system call listen */  
55 ::listen(this->socketfd, 5);  
56  
57 /* Update address of client */  
58 socklen_t clientLength = sizeof(this->clientAddress);  
59  
60 /* Accept connection */  
61 this->clientSocketfd = accept(this->socketfd,  
62 (struct sockaddr *) &this->clientAddress,  
63 &clientLength);  
64  
65 /* Check that client socket is OK */  
66 if (this->clientSocketfd < 0){  
67 perror("Socket Server: Failed to bind the client socket properly.\n");  
68 return 1;  
69 }  
70  
71 }
```

Ingen specifik IP-adresse

Sat i headerfil, men kunne også være argument

Backlog, tillader kø på 5 ventende forbindelser

BILAG TIL PRU

SOFTWARE - ØVELSE 3

REALTID, PRU

E4ISD2

Janus Bo Andersen (JA67494)

ØVELSE 3: REALTIDSPROCESSERING DESIGN

Formål: Afdæk hvor hurtigt et firkantsignal kan propagere gennem PRU'erne, så de "stadic kan følge med".

Design:

- Målrettet at opnå lav latency gennem PRU'erne:
- PRU0 aflæser GPIO-input direkte
- Scratchpad (SPAD) benyttes til at overføre data fra PRU0 til PRU1
 - XOUT / XIN operationer
- PRU1 sætter GPIO-output direkte
- Kode skrevet/optimeret i assembler.

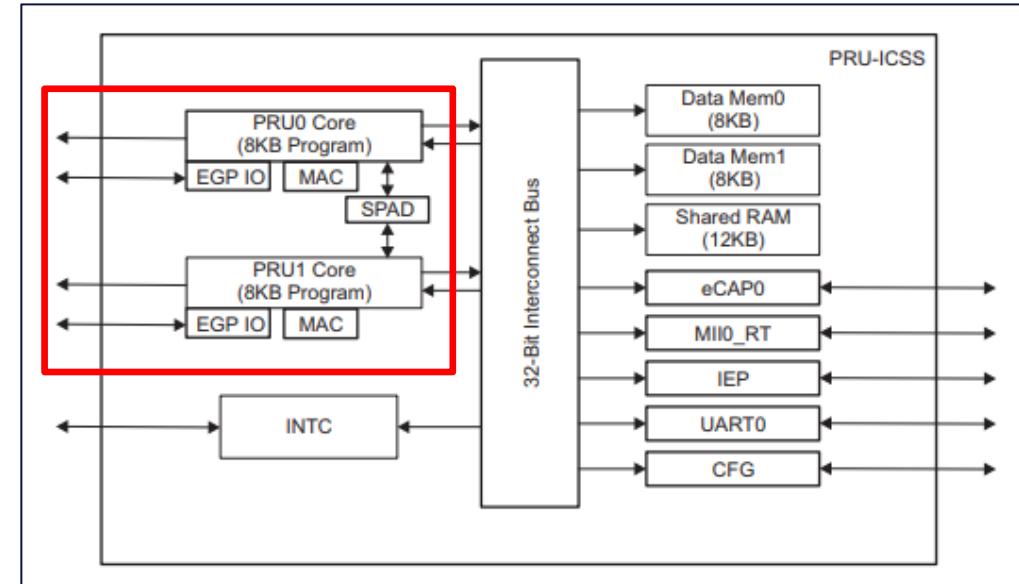
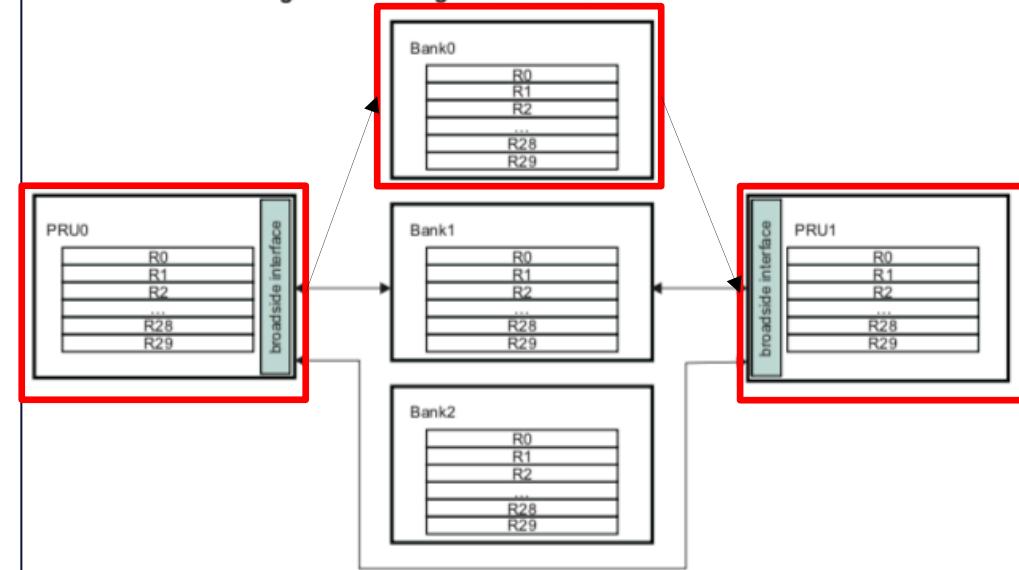
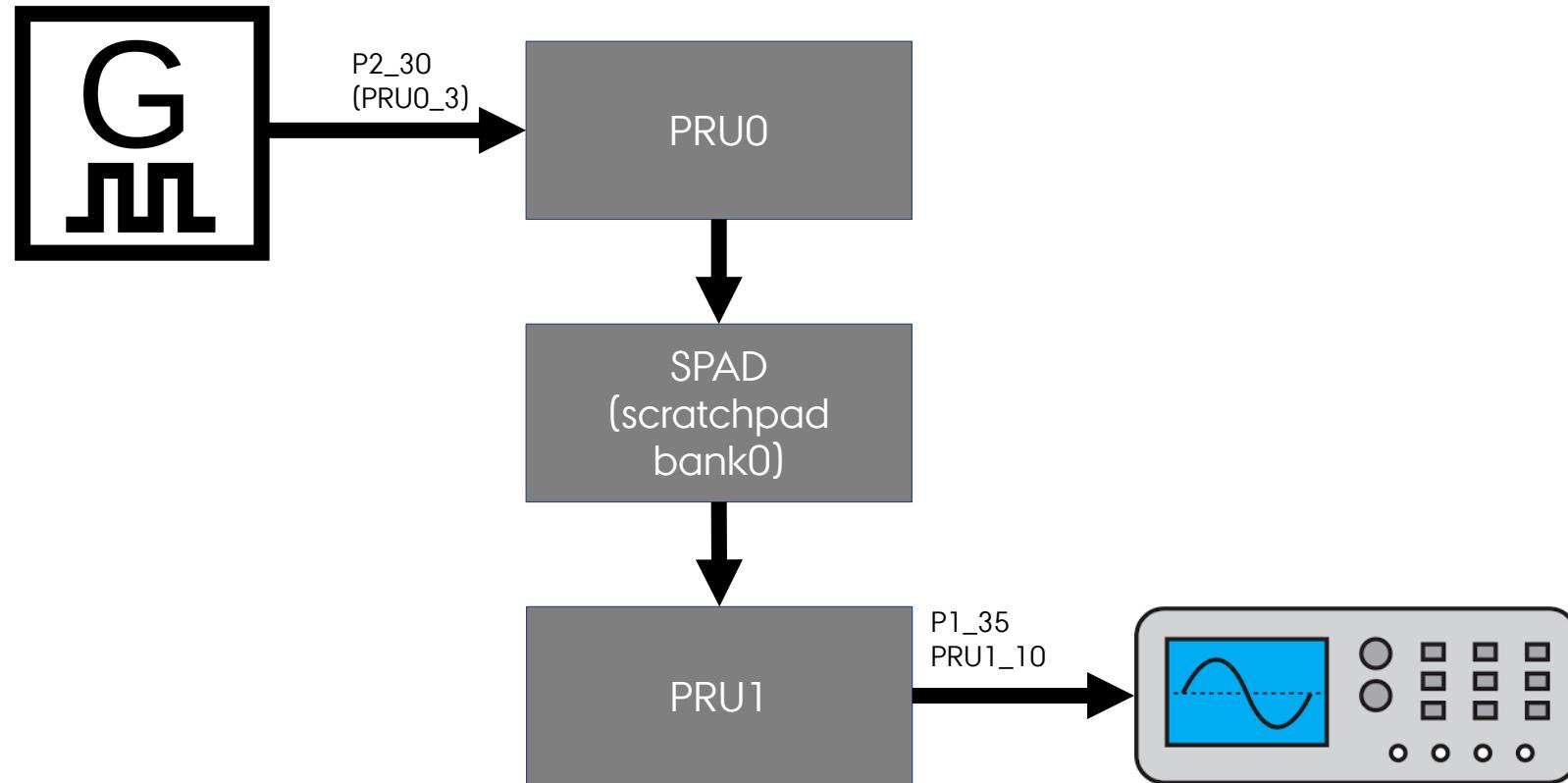


Figure 4-16. Integration of PRU and Scratch Pad



BLOKDIAGRAM

FIRKANTINPUT PÅ PRU0, OUTPUT FRA PRU1



TESTSETUP

Pins for PRU0 og PRU1 sættes op til hhv. in- og output

```
config_pru_pins.sh
1 # input pin PRU0_3, i.e. pr1_pru0_r31_3
1 config-pin P2_30 pruin
2
3 # output pin PRU1_10, i.e. pr1_pru1_r30_10
4 config-pin P1_35 pruout
```

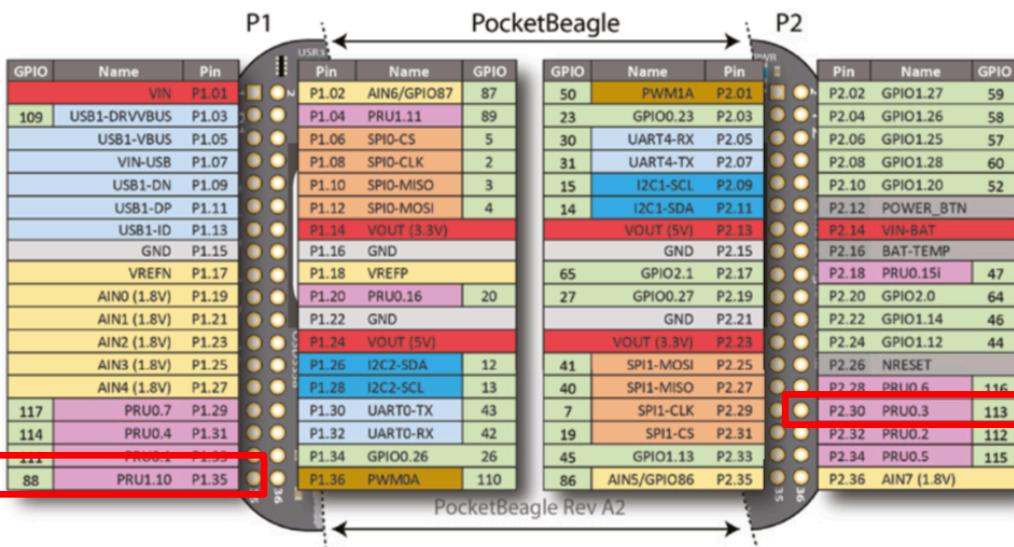
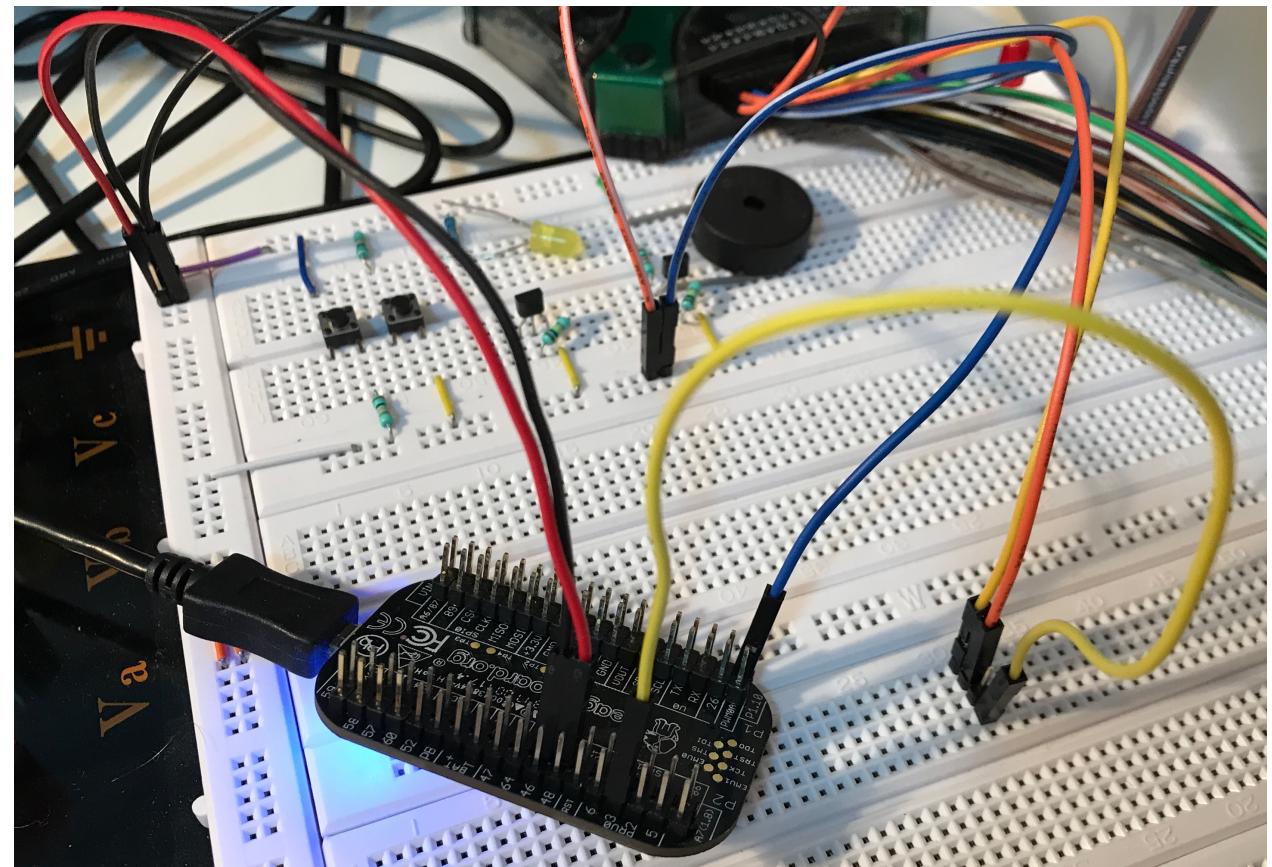


Figure 6-2: The PocketBeagle P1/P2 headers with pin names, which describe each pin's default functionality



Elektrisk:

- Fælles stelplan for PocketBeagle og Analog Discovery.
- Wavegen1 (gul) og oscilloskop kanal 1 (orange) ved P2_30 (pruin).
- Oscilloskop kanal 2 (blå) ved P1_35 (pruout).

ASSEMBLER

Instruktioner til PRU0

```
1      .cdecls "perfTestContainer.c"
2
3      .clink
4      .global start
5      start: CLR    r30, r30.t5      ; turn off the LED
6      SET    r0, r0.t10      ; Initialize value to send to PRU1
7      CHECK: WBS    r31, 3      ; wait for bit set on GPIO in
8      XOUT   10, &r0, 4      ; Send value to PRU1
9      CLR    r0, r0.t10      ; Prepare the next value to send to PRU1
10     WBC    r31, 3      ; wait until the bit is cleared again
11     XOUT   10, &r0, 4      ; Send value to PRU1
12     SET    r0, r0.t10      ; Prepare value for next round to PRU1
13     JMP    CHECK        ; Repeat
14
15     HALT
```

WBS og WBC er
pseudoinstruktioner, der
indefholder loop og compare
(busy polling)

Afsender besked via SPAD så
hurtigt som muligt, og
forbereder registeropdatering
til næste "edge" bagefter.

Instruktioner til PRU1

```
1      .cdecls "perfTestContainer.c"
2
3      .clink
4      .global start
5      start: CLR    r30, r30.t10      ; start with GPIO low
6      CHECK: XIN    10, &r0, 4      ; read from scratchpad bank 0
7      MOV    r30, r0        ; Move value from scratchpad into GPIO out
8      JMP    CHECK        ; loop
9
10     HALT
```

Aflæser kontinuerligt
scratchpad, og indsætter
værdien i GPIO out registret.

Det kunne være smukt at
advisere PRU1 via interrupt
om at værdien er opdateret,
men dette her virker
umiddelbart hurtigt

PERFORMANCE TEST - 1 PULS

Kanal 1: Input til PRU0 (knap)

Kanal 2: Output fra PRU1

- 40 ns fra input høj til output høj
- => 8 clock cycles @200 MHz

Regnskab over ticks:

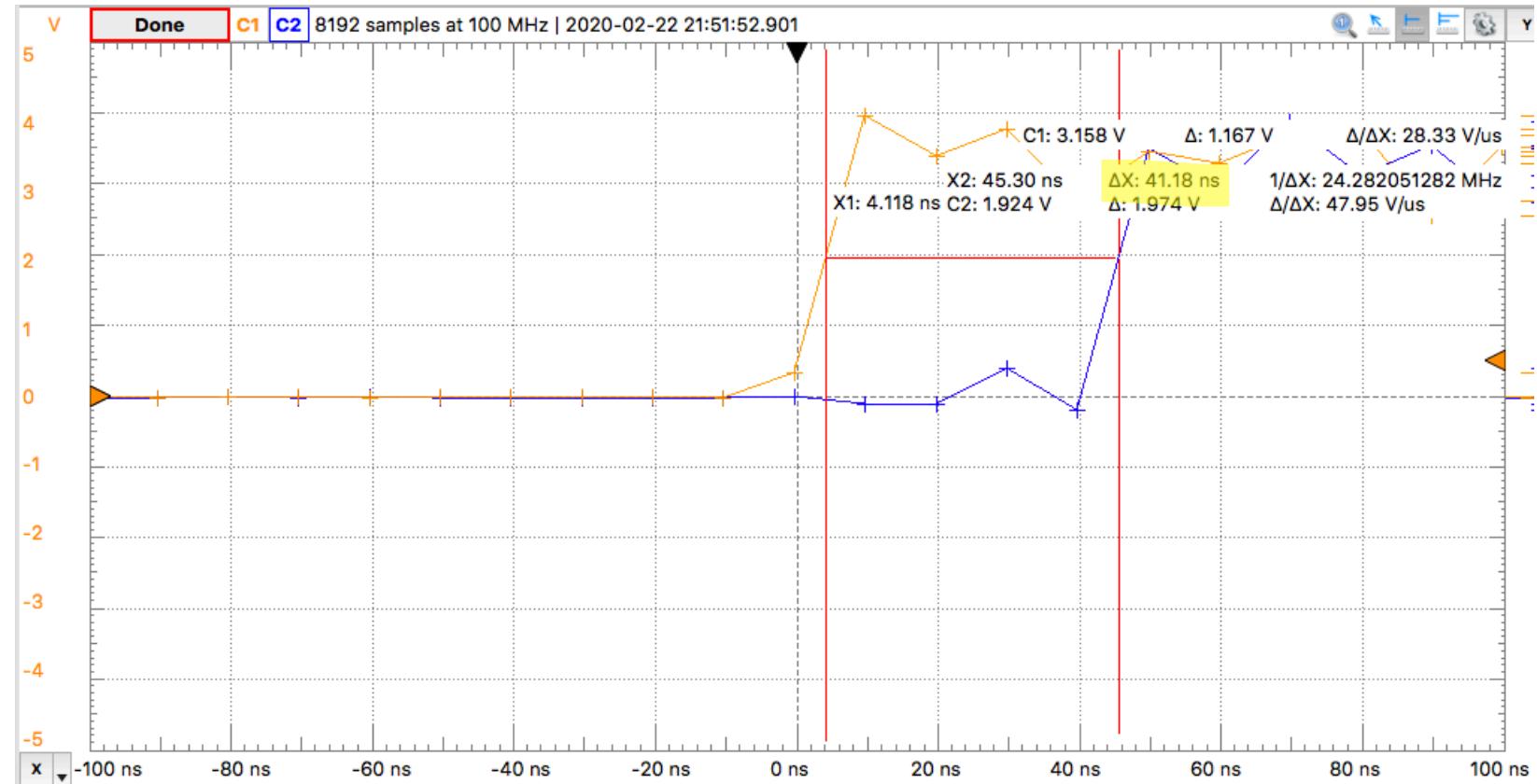
- Propagation delay/internal delay?

PRU0 latency:

- 3-4 ticks WBS/WBC-'spinning' indtil GPIO aflæst og registreretændret
- 1 tick til at overføre værdi til SPAD

PRU1 latency:

- 2-3 ticks i loop til at læse SPAD, opdatere GPIO out, JMP



PERFORMANCE TEST - WAVE GENERATOR

Setup: Se t.h.

- Definition:
- **OK latency: ≤ 50 ns**

Frekvens på firkantsignalet øges:

- Se næste billeder
- OK op til RF-frekvenser
- Herefter giver det ikke mening at måle mere



LATENCY VED 1 MHZ FIRKANTSIGNAL

Ved 1 MHz ligger latency
stadic omkring 40 ns

PRU kan stadic følge med!

I nærheden af HF-delen i RF-
båndet...

- Ikke-ideelle effekter begynder at spille ind
 - Selv-induktive effekter i ledninger
 - Kapacitans i breadboard, m.m.



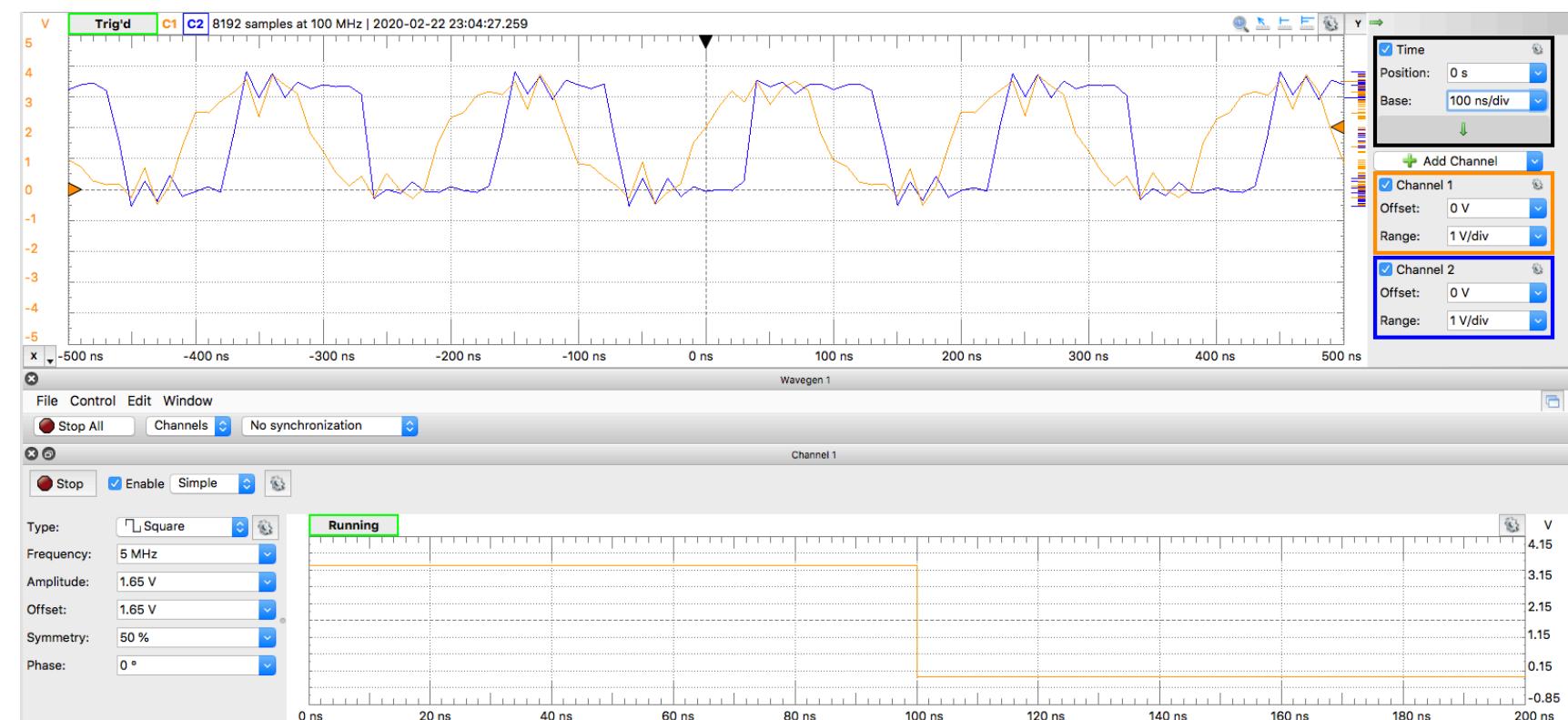
LATENCY VED 5 MHZ FIRKANTSIGNAL

Latency gennem PRU'erne stadig
< 50 ns ved 5 MHz:

- Signalkvalitet ind er væsentligt forringet
- Kraftig ringning på signal ud

Stopper forsøget her, og konkluderer, at måleudstyret er outperformed af PRU'erne

- Skal benytte BNC-prober for at komme det nærmere.



KONKLUSION

Pga. meget **lav** og **deterministisk** (worst-case) **latency**, er PRU'erne velegnede til:

- Hård-realtime:
 - Garanterede deadlines vha. deterministisk afvikling ved 200 MHz / 5 ns ticks.
- Håndtere højfrekvent input/output (i hvert fald op til HF), fx
 - Afstandsmåling, timing, osv.
 - Stabil/præcis signalgenerator.
- Afkoble og synkronisere behandling af signaler
 - Input-signal til én PRU og output-signaler fra en anden
 - Synkronisering via delte memory-områder med meget lave read latencies (1 tick? for SPAD, 3 for Shared DRAM).
- Arbejde i realtid uafhængigt af Linux-kernen.
- Synkronisere via beskeder til/fra Linuxkernen vha. remoteproc.

APPENDIKS: BEST-CASE LATENCY (PRU LOKALT)

A.1 AM335x

Table 1 through Table 3 are considered "best-case" read latency values for the PRU on AM335x.

Table 1. AM335x PRU Read Latencies - Local PRU Subsystem Resources

MMRs	Read Latency (PRU cycles @ 200 MHz)
PRU CTRL	4
PRU CFG	3
PRU INTC	3
PRU DRAM	3
PRU Shared DRAM	3
PRU ECAP	4
PRU UART	14
PRU IEP	12
PRU R31 (GPI)	1

<http://www.ti.com/lit/an/sprace8a/sprace8a.pdf>

SW3: DISTRIBUEREDE SYSTEMER: PROCESPERSPEKTIVET

Indhold

1. Overblik
2. Processer i Linux
3. Procestilstande og scheduling
4. Perspektivering
5. Opsummering og konklusion

Forklar indholdet af nedennævnte perspektiv og relationen til andre perspektiver

Procesperspektivet

DE FIRE PERSPEKTIVER (VIEWS) OVERBLIK

Forskellige perspektiver på hvordan *krav* og *designmål* kan opnås

Forskellige indgange til fornuftigt anvendelse af *teknologi* i design af distr. systemer.

Proces

- Proceshåndtering, tråde, samtidighed
- IPC, kommunikation på computerniv., sockets og porte
- Scheduling, blocking
- Meddelelsesbuffering

Design af distribuerede systemer

Gennemgående: Teknikker til at opnå gennemsigtighed/gennemsuelighed (transparency).

Arkitektur

- Opdeling på funktionelle komponenter
- Modeller for arkitektur
 - Betydning for non-funktionelle kvalitetskrav
- Struktur, kompleksitet

Ressourcer

- Rationel / økonomisk anvendelse af knappe ressourcer
 - Processeringskapacitet (delt mellem lokale processer)
 - Båndbredde (fælles ressource)
 - Hukommelse (til prog. og data) vs. omkostning

Kommunikation / netværk

- Netværk og protokoller
- Socket API
- RPC, struktureret kommunikation

PROCESSPERSPEKTIVET OVERBLIK

Dette perspektiv omfatter

- Proceshåndtering
 - Processer og tilstande
 - Schedulering og -blocking
- Hvordan **processer og kommunikation spiller sammen**
 - OS'ets håndtering af kommunikation på computer-niveau
 - Processer og tråde
 - IPC: Pipes, sockets, signaler
 - Sockets, binding til porte

PROCESSER OVERBLIK

Processer i Linux

Def. proces:

- Kørende instans af et program (instruktioner)
 - Ikke-tomt finit mængde af tråde, med samme (virtuelle) adress space
- Proces omfatter “image“ og “state“
 - Kan inspicere processers tilstand med `ps`.
 - Mapping i `/proc/`
- Når program afvikles skabes (minimum) 1 proces

IPC: Mekanismer og metoder hvormed processer kan kommunikere.

- Fx named/anonyme pipes, sockets, signaler, delt hukommelse, osv.

Procestilstand
S: Interruptible sleep
I: Idle kernel thread
R: Running
OSV...

Scheduling class/policy
TS: Other
FF: FIFO
RR: Round Robin
OSV...

Prioritet
Nice value: -20 (højeste prio.) til 19 (laveste). Mest nice bliver kørt oftest ☺
Realtime priority: 140 mulige værdier

ps -eflc

F	S	SUID	PID	PPID	CLS	PRI	ADDR	SZ	WCHAN	STIME	TTY	TIME	CMD
4	S	root	1	0	TS	19	-	25582	-	16:14	?	00:00:05	/sbin/init splash
1	S	root	2	0	TS	19	-	0	-	16:14	?	00:00:00	[kthreadd]
1	I	root	3	2	TS	39	-	0	-	16:14	?	00:00:00	[rcu_gp]
1	I	root	4	2	TS	39	-	0	-	16:14	?	00:00:00	[rcu_par_gp]
1	I	root	6	2	TS	39	-	0	-	16:14	?	00:00:00	[kworker/0:0H-kblockd]
1	I	root	8	2	TS	39	-	0	-	16:14	?	00:00:00	[mm_percpu_wq]
1	S	root	9	2	TS	19	-	0	-	16:14	?	00:00:00	[ksoftirqd/0]
1	I	root	10	2	TS	19	-	0	-	16:14	?	00:00:01	[rcu_sched]
1	S	root	11	2	FF	139	-	0	-	16:14	?	00:00:00	[migration/0]
5	S	root	12	2	FF	90	-	0	-	16:14	?	00:00:00	[idle_inject/0]
1	I	root	13	2	TS	19	-	0	-	16:14	?	00:00:01	[kworker/0:1-events]
1	S	root	14	2	TS	19	-	0	-	16:14	?	00:00:00	[cpuhp/0]
1	S	root	15	2	TS	19	-	0	-	16:14	?	00:00:00	[cpuhp/1]
5	S	root	16	2	FF	90	-	0	-	16:14	?	00:00:00	[idle_inject/1]
1	S	root	17	2	FF	139	-	0	-	16:14	?	00:00:00	[migration/1]
1	S	root	18	2	TS	19	-	0	-	16:14	?	00:00:00	[ksoftirqd/1]
1	I	root	20	2	TS	39	-	0	-	16:14	?	00:00:00	[kworker/1:0H-kblockd]
5	S	root	21	2	TS	19	-	0	-	16:14	?	00:00:00	[kdevtmpfs]
1	I	root	22	2	TS	39	-	0	-	16:14	?	00:00:00	[netns]
1	S	root	23	2	TS	19	-	0	-	16:14	?	00:00:00	[rcu_tasks_kthre]

<https://man7.org/linux/man-pages/man1/ps.1.html>

PROCESSER

PROCESTILSTANDE

Schedulering af processer

Fordeling af processor-tid

- Fordi kun 1 proces kan afvikles ad gangen på en CPU-kerne
 - Typisk: #processer > #kerner
- OS'et (hvis der er sådan et) håndterer brug af CPU som ressource, afgør hvilke processer afvikles hvornår
 - Linux: Completely Fair Scheduler: MLFQ-baseret
 - Enkelt-trådete processer (lavet med fork)
 - Multi-trådete processer: Linux schedulerer trådene (pthreads)
 - Særlige kernel threads, kworkers, osv.

Typer af scheduling:

- Preemptive (tvungen tidsdeling)
- Collaborative (frivillig tidsdeling)
- Batch-processering

Overvejelser til at dispatche / swappe processer ind og ud:

- **General purpose-system** (Linux:niceness, prioritet, fairness, tidsdeling, prioritet, tilbageværende afviklingstid, I/O-brug, osv.) vs.
- **Realtime-system** (deadlines, maks. afviklingstid, periodicitet).

Tilstande for en proces

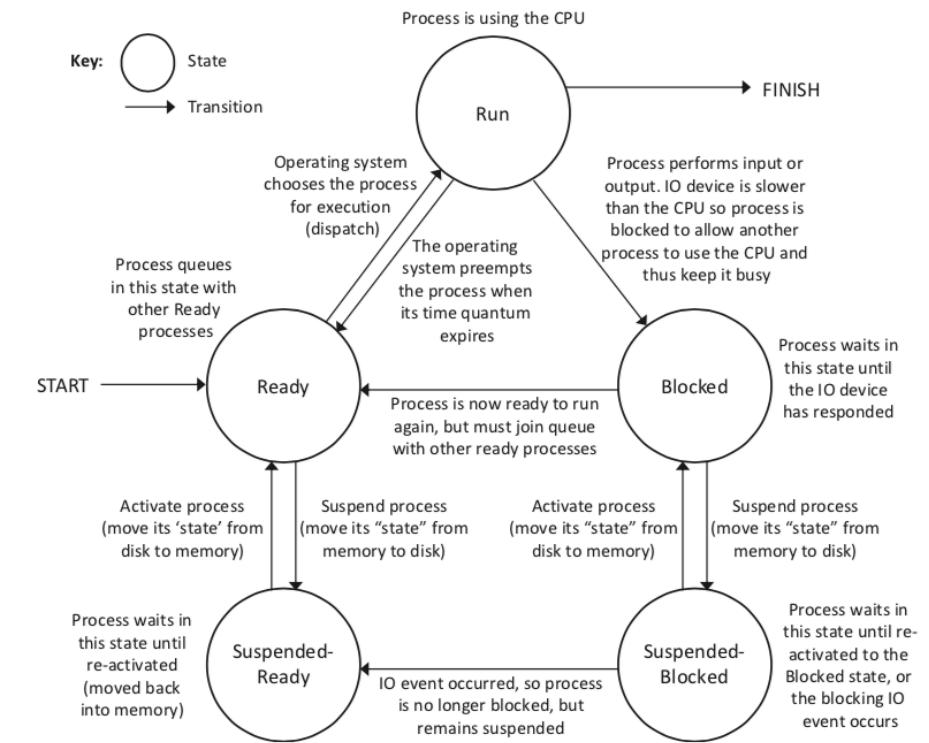


FIGURE 2.19

The extended process state transition model including suspended process states.

REALTIDSBEHOV

Realtidskrav -> OS-schedulering er ikke-deterministisk

- Hard realtime
- Firm realtime
- Soft realtime

Hastighed på compute skal matche proces, fx ved regulering: Kan delvis løses med edge computing.

Men med **realtidskrav**:

- Når SW ikke er hurtig nok / kan stille garantier -> dedikeret hardware

Eksempel:

- 200 MHz PRU'er på BeagleBone i stedet for processering i Linux på CPU, firkantsignal ind.
- Assembler-kode på PRU, op til 5 MHz
- C-kode i Linux, op til 2-2,5 kHz
- ~ faktor 2000 forskel i frekvens, der kan håndteres

Sammenligning af performance

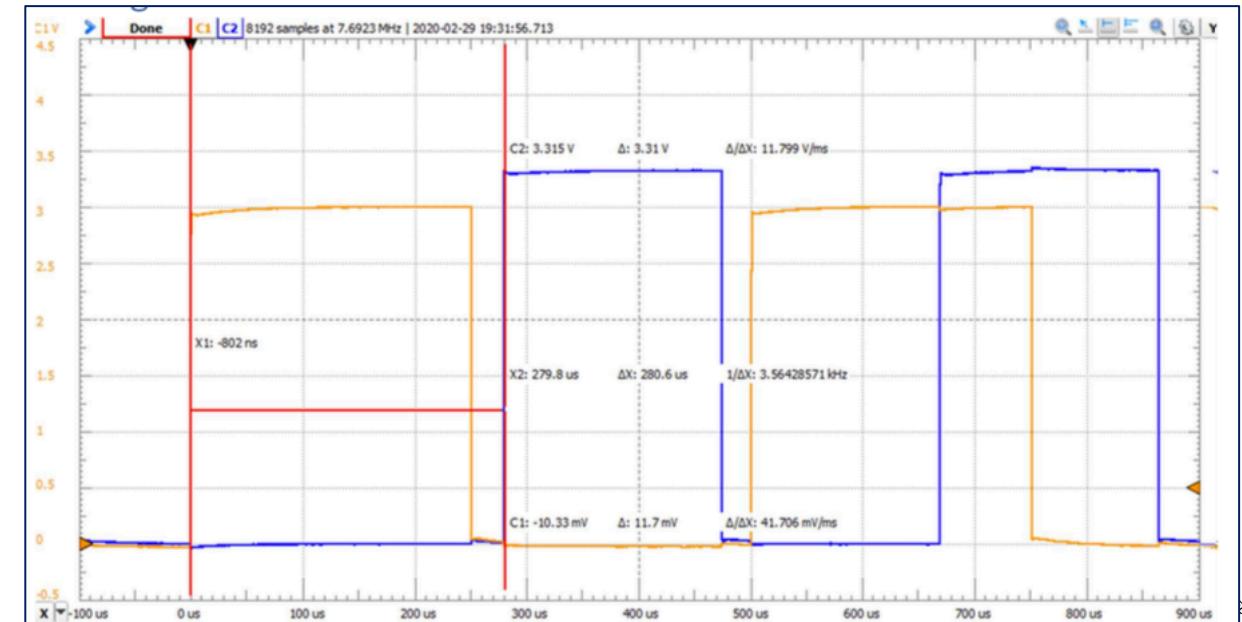
LATENCY VED 1 MHZ FIRKANTSIGNAL

Ved 1 MHz ligger latency
stadic omkring 40 ns

PRU kan stadic følge med!

I nærheden af HF-delen i RF-
båndet...

- Ikke-ideelle effekter begynder at spille ind
- Selv-induktive effekter i ledninger
- Kapacitans i breadboard, m.m.



Figur 4 - Måling af OS-styret in/output delay ved 2KHz

Ved 2Khz ses et delay på $280\mu s$.

DAEMONIZATION (DÆMONISERING)

Formål: Gøre en proces til en Linux/UNIX-daemon

- Ofte benyttet i distribuerede systemer
 - Services, baggrundsprocesser
- Ingen terminal, så
 - Kontrol vha. signaler
 - Output og fejl skrives til syslog

Metode:

- **Installér sighandler**
- Fork parent-proces
- Luk parent-proces
- Indstil daemon:
 - Ændr file mode mask (umask)
 - Åbn syslog
 - Nyt sessions-ID (sid) til procesgruppe for child
 - Skift workdir
 - Luk file descriptors
- Afvikling af daemon/service

Del 1: Installer sighandler

```
1  #include <iostream>
2  #include <cstdlib>
3  #include <csignal>
4  #include <sys/stat.h>
5  #include <unistd.h>
6  #include <sys/syslog.h>
7
8  // signal handler reacts to hangup signal
9  void sig_handler(int signo) {
10     if (signo == SIGHUP) {
11         syslog( pri: LOG_MAKEPRN(LOG_LOCAL0, LOG_NOTICE), fmt: "Daemon received SIGHUP. Stopping.");
12         closelog();
13         exit(EXIT_SUCCESS);
14     }
15 }
16
17 ▶ int main(void) {
18
19     // install signal handler
20     if ( signal(SIGHUP, sig_handler) == SIG_ERR) {
21         std::cout << "Failed to register SIGHUP" << std::endl;
22     }
23
24     std::cout << "Starting Daemonization!" << std::endl;
```

DAEMONIZATION (DÆMONISERING)

Metode:

- Installér sighandler
- **Fork parent-proces**
- Luk parent-proces
- **Indstil daemon:**
 - **Ændr file mode mask (umask)**
 - **Åbn syslog**
 - Nyt sessions-ID (sid) til procesgruppe for child
 - Skift workdir
 - Luk file descriptors
- Afgang af daemon/service

Del 2: Forking og åbn syslog

```
24         std::cout << "Starting Daemonization!" << std::endl;
25
26         pid_t pid, sid;                                // Process ID and Session ID
27
28         pid = fork();                                 // Fork to make copied process
29         if (pid < 0) {
30             exit(EXIT_FAILURE);                      // Failed to fork
31         }
32
33         if (pid > 0) {                               // pid in child is 0
34             exit(EXIT_SUCCESS);                     // Exit parent process, not needed
35         }
36
37         umask( mask: 0);                            // rwx permission on file creation
38
39         // Open log
40         openlog( ident: "daemon-proc",
41                   option: LOG_CONS | LOG_PID | LOG_NDELAY,
42                   facility: LOG_LOCAL0);
43
```

DAEMONIZATION (DÆMONISERING)

Metode:

- Installér sighandler
- Fork parent-proces
- Luk parent-proces
- Indstil daemon:
 - Ændr file mode mask (umask)
 - Åbn syslog
 - **Nyt sessions-ID (sid) til procesgruppe for child**
 - **Skift workdir**
 - **Luk file descriptors**
- **Afvikling af daemon/service**

Del 3: Sessions-ID og afvikling af arbejde

```
44     sid = setsid();           // new session id
45     if (sid < 0) {
46         syslog( pri: LOG_MAKEPRI(LOG_LOCAL0, LOG_NOTICE), fmt: "Daemon failed to get sid.");
47         exit(EXIT_FAILURE); // failed to sid
48     } else {
49         syslog( pri: LOG_MAKEPRI(LOG_LOCAL0, LOG_NOTICE), fmt: "Daemon got sid: %d.", sid);
50     }
51
52     if ((chdir( path: "/" )) < 0) { // change path if creating files
53         syslog( pri: LOG_MAKEPRI(LOG_LOCAL0, LOG_NOTICE), fmt: "Daemon failed to chdir.");
54         exit(EXIT_FAILURE);
55     }
56
57     // Remove access to terminal file descriptors
58     close(STDIN_FILENO);
59     close(STDOUT_FILENO);
60     close(STDERR_FILENO);
61
62     // Do daemon work
63     syslog( pri: LOG_MAKEPRI(LOG_LOCAL0, LOG_NOTICE), fmt: "Daemon starting work.");
64     while (true) {
65         sleep( seconds: 30 );
66     }

```

TEST OG KONTROL OVER DAEMON

Test:

- Start daemon
 - Bekræft
- Test SIGHUP
- Dræb daemon
 - Bekræft

Øvrigt:

- Daemon har et proc-interface som alle andre processer:
 - Kig i proc/pid/-mappen og se evt. fd'er, cmdline, osv.

Test og kontrol

Daemon skriver til syslog

Processen eksisterer med det nye sid (1331 er systemd)

Daemon findes som proces i /proc/, med alt tilhørende

Daemon stoppes med SIGHUP

Daemon rapporter software interrupt til syslog

Daemon er død

```
janus@janus-vm:~/projects/e4isd2/u6_daemonize/cmake-build-debug$ ./u6_daemonize
Starting Daemonization!
janus@janus-vm:~/projects/e4isd2/u6_daemonize/cmake-build-debug$ tail -n 2 /var/log/syslog
Jun 25 16:54:43 janus-vm daemon-proc[12906]: Daemon got sid: 12906.
Jun 25 16:54:43 janus-vm daemon-proc[12906]: Daemon starting work.
janus@janus-vm:~/projects/e4isd2/u6_daemonize/cmake-build-debug$ ps -ef | grep "12906"
janus 12906 1331 0 16:54 ? 00:00:00 ./u6_daemonize
janus 12910 3275 0 16:54 pts/0 00:00:00 grep --color=auto 12906
janus@janus-vm:~/projects/e4isd2/u6_daemonize/cmake-build-debug$ ll /proc/ | grep "12906"
dr-xr-xr-x 9 janus janus 0 Jun 25 16:54 12906/
janus@janus-vm:~/projects/e4isd2/u6_daemonize/cmake-build-debug$ kill -1 12906
janus@janus-vm:~/projects/e4isd2/u6_daemonize/cmake-build-debug$ tail -n 2 /var/log/syslog
Jun 25 16:54:43 janus-vm daemon-proc[12906]: Daemon starting work.
Jun 25 16:55:32 janus-vm daemon-proc[12906]: Daemon received SIGHUP. Stopping.
janus@janus-vm:~/projects/e4isd2/u6_daemonize/cmake-build-debug$ ll /proc/ | grep "12906"
janus@janus-vm:~/projects/e4isd2/u6_daemonize/cmake-build-debug$ ps -ef | grep "12906"
janus 12919 3275 0 16:55 pts/0 00:00:00 grep --color=auto 12906
```

SOCKETS OG TRÅDE

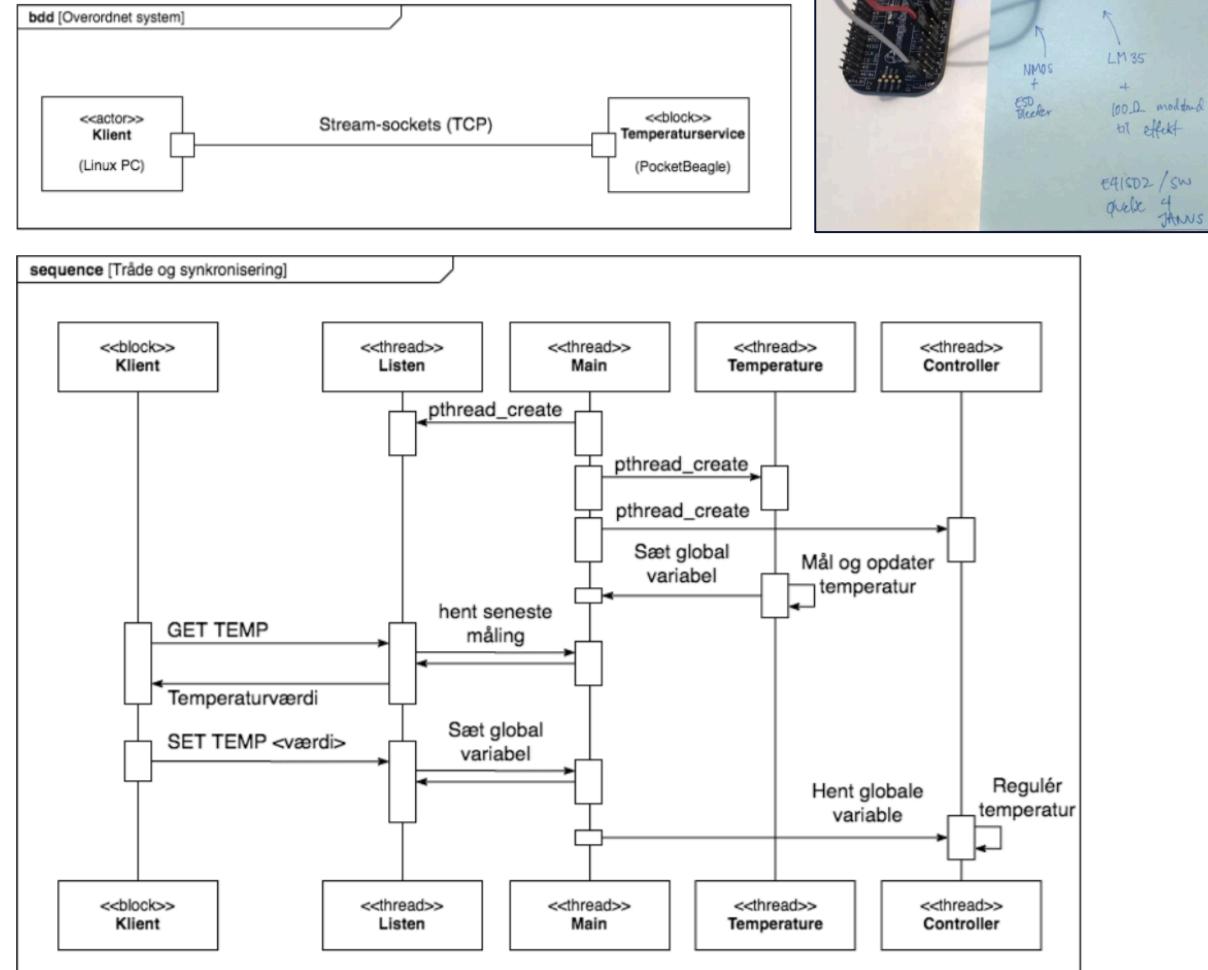
EKSEMPEL FRA TEMP SERVER

Temperaturserver

- Klient-server-arkitektur
 - Server på Beaglebone
 - LM35-temperatursensor
 - TCP-sockets
 - Trådet applikation

Formål med brug af sockets og tråde i arkitekturen:

- Sockets: Kommunikere over netværk (TCP)
- Tråde: Opdele arbejde med forskellige tidsconstraints og I/O-typer.
 - Listen-tråden blokerer på `listen()` og på `read()`/`recv()`.
 - Temperatursensor aflæses med interval.
 - Controller håndterer PI-regulator med interval.

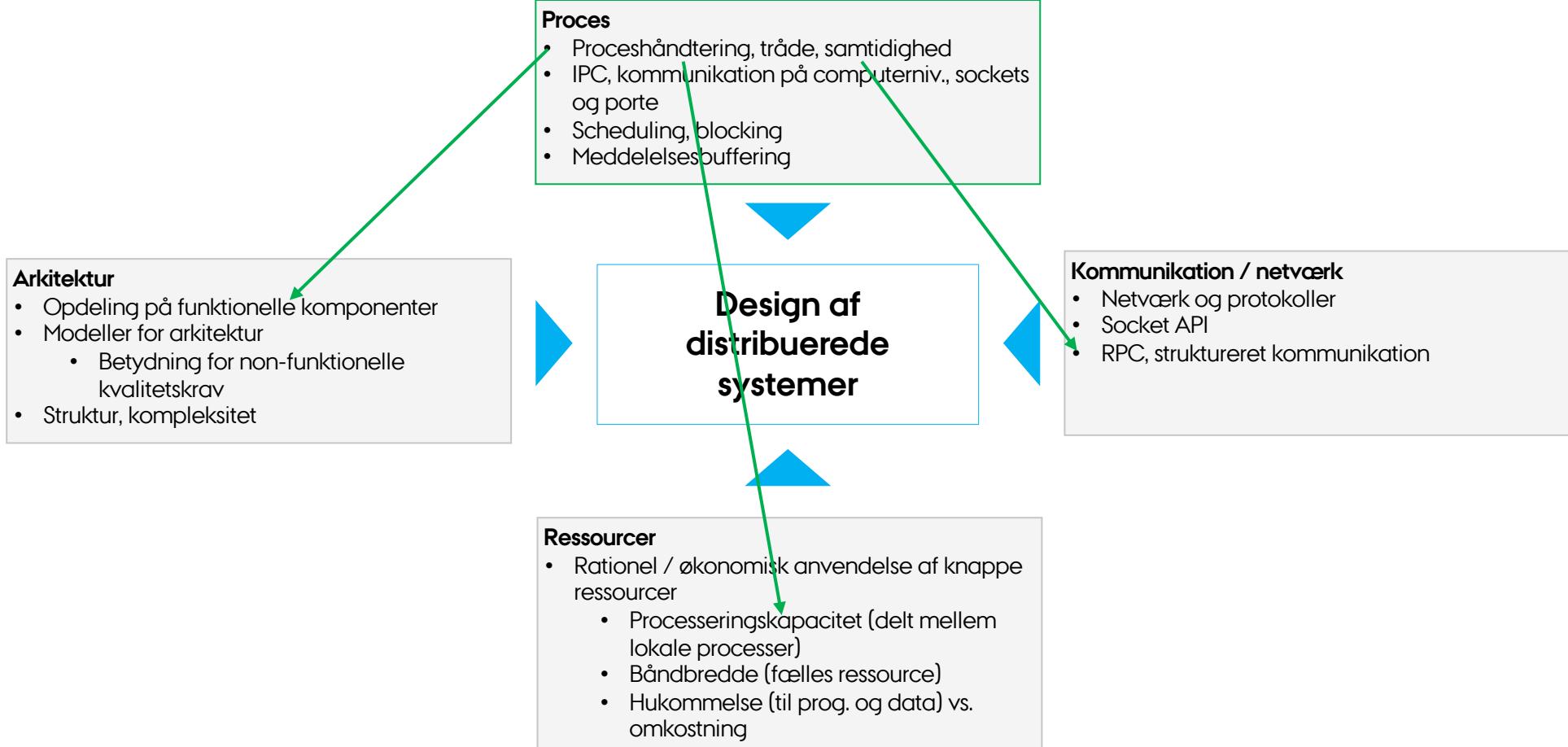


DE FIRE PERSPEKTIVER (VIEWS) PERSPEKTIVERING

Forskellige perspektiver på hvordan *krav* og *designmål* kan opnås

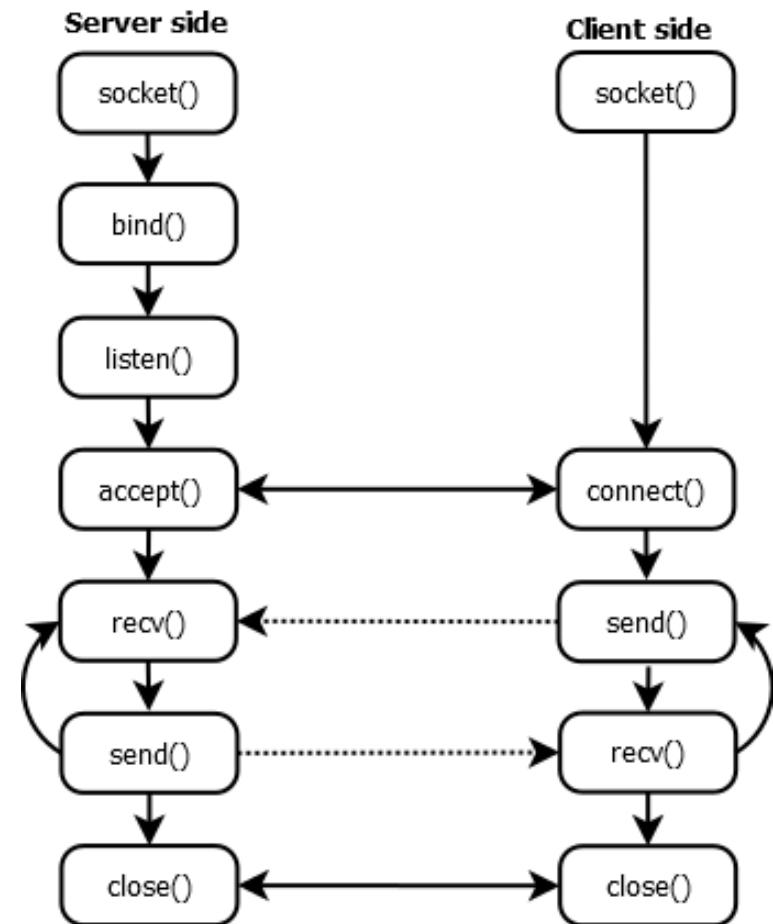
Forskellige indgange til fornuftigt anvendelse af *teknologi* i design af distr. systemer.

Gennemgående: Teknikker til at opnå gennemsigtighed/gennemsuelighed (transparency).



OPSUMMERING OG KONKLUSION

BILAG



TRÅDE

Opstart og nedlukning af tråde

```
58 int main(void) {
59
60     /* Set up signal handler */
61     if ( signal(SIGHUP, sig_handler) == SIG_ERR ) {
62         std::cout << "Can't register SIGHUP" << std::endl;
63     }
64
65     pthread_t listen_thread;
66     if ( pthread_create(&listen_thread, NULL, &listen_thread_function, NULL) ) {
67         std::cout << "Could not create thread" << std::endl;
68         return EXIT_FAILURE;
69     }
70
71     pthread_t temp_thread;
72     if ( pthread_create(&temp_thread, NULL, &temp_monitor_thread_function, NULL) ) {
73         std::cout << "Could not create thread" << std::endl;
74         return EXIT_FAILURE;
75     }
76
77     pthread_t control_thread;
78     if ( pthread_create(&control_thread, NULL, &controller_thread_function, NULL) ) {
79         std::cout << "Could not create thread" << std::endl;
80         return EXIT_FAILURE;
81     }
82
83     /* Temperature thread has been stopped by SIGHUP */
84     pthread_join(temp_thread, NULL);
85
86     /* Controller thread has been stopped by SIGHUP */
87     pthread_join(control_thread, NULL);
88
89     /* Safe to kill the listener now */
90     pthread_cancel(listen_thread);
91     pthread_join(listen_thread, NULL);
92
93     return 0;
94 }
```

Benytter POSIX-tråde (pthreads)

Opstart af trådfunktioner

Tråde afkobles ikke fra hoved-tråden,
men synkroniseres ved nedlukning.

Når regulator-tråden er død, kan
resten lukkes ned.

Synkronisering af variable og nedlukning

```
44 /* Coordinate clean exit : */
45 std::atomic<bool> exit_now(false);
46
47 void sig_handler(int signo) {
48
49     // React to caught signal
50     if (signo == SIGHUP) {
51         std::cout << "Received SIGHUP. Bye!" << std::endl;
52
53         // Tell all threads to end now, cleanly
54         exit_now.store(true);
55     }
56 }
21 /* These enable communication between threads */
22 extern float cur_temp;
23 extern float set_point;
24 extern std::atomic<bool> exit_now;
25
26 /* @brief Thread function to monitor temperature in the background
27 * update the global variable CUR_TMP
28 */
29 void * temp_monitor_thread_function(void * value) {
30
31     int adc_value;
32     float temp;
33
34     while( !exit_now.load() ) {
35
36         // Take a sample and convert to temperature
37         adc_value = read_analog(ADC);
38         temp = conv_temperature(adc_value);
39
40         // Update the global variable
41         cur_temp = temp;
42
43         // Sleep until next sample - this is also a thread cancellation point
44         sleep(TEMP_SAMPLE_INTERVAL);
45     }
46
47     pthread_exit(NULL);
48 }
```

Atomisk variabel til
koordinere nedlukning.

Globale variable, header fil.
Her kunne overvejes mutex eller
en atomisk variabel.

Tjek for nedlukning.

Opdatér global variabel.

SOCKETS

SocketServer – klasse til serverfunktion

```
16@/**  
17 * @class SocketServer  
18 * @brief A class that encapsulates a server socket for network communication  
19 */  
20@class SocketServer {  
1 private:  
22 int portNumber;  
23 int socketfd, clientSocketfd;  
24 struct sockaddr_in serverAddress;  
25 struct sockaddr_in clientAddress;  
26 bool clientConnected;  
27  
28 public:  
29 SocketServer(int portNumber);  
30 virtual int listen();  
31 virtual int send(std::string message);  
32 virtual std::string receive(int size);  
33 virtual void closeClient();  
34  
35 virtual ~SocketServer();  
36 };
```

Objekt instantieres med portnummer

Wrappers til 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 }  
83  
84@string SocketServer::receive(int size=1024){  
85 char readBuffer[size];  
86 memset(readBuffer, '\0', size);  
87 int n = read(this->clientSocketfd, readBuffer, sizeof(readBuffer));  
88 if (n < 0){  
89 perror("Socket Server: error reading from server socket.");  
90 throw "Socket probably closed.";  
91 }  
92 return string(readBuffer);  
93 }
```

Wrapper de underliggende C-kald, og oversetter til C-strukturer

Opsætning og indkommende requests (listen())

```
25@ /* listen creates the socket and  
26 * blocks until an incoming connection established,  
27 * and then accepts the incoming connection */  
28@ int SocketServer::listen(){  
29  
30 /* Make INET (TCP) socket */  
31 this->socketfd = socket(AF_INET, SOCK_STREAM, 0);  
32  
33 /* Check that socket created OK */  
34 if (this->socketfd < 0){  
35 perror("Socket Server: error opening socket.\n");  
36 return 1;  
37 }  
38  
39@ /* Ensure all zeros in unused part of serverAddress  
40 * this is an alternative to memset */  
41 bzero((char *) &serverAddress, sizeof(serverAddress));  
42  
43 /* Create struct for address */  
44 serverAddress.sin_family = AF_INET;  
45 serverAddress.sin_addr.s_addr = INADDR_ANY;  
46 serverAddress.sin_port = htons(this->portNumber);  
47  
48 /* Attempt to bind to the socket address */  
49 if (bind(socketfd, (struct sockaddr *) &serverAddress, sizeof(serverAddress)) < 0) {  
50 perror("Socket Server: error on binding the socket.\n");  
51 return 1;  
52 }  
53  
54 /* system call listen */  
55 ::listen(this->socketfd, 5);  
56  
57 /* Update address of client */  
58 socklen_t clientLength = sizeof(this->clientAddress);  
59  
60 /* Accept connection */  
61 this->clientSocketfd = accept(this->socketfd,  
62 (struct sockaddr *) &this->clientAddress,  
63 &clientLength);  
64  
65 /* Check that client socket is OK */  
66 if (this->clientSocketfd < 0){  
67 perror("Socket Server: Failed to bind the client socket properly.\n");  
68 return 1;  
69 }  
70  
71 }
```

Ingen specifik IP-adresse

Sat i headerfil, men kunne også være argument

Backlog, tillader kø på 5 ventende forbindelser

SW4: DISTRIBUEREDE SYSTEMER: RESSOURCEPERSPEKTIVET

Indhold

1. Overblik
2. Processeringskraft som begrænset ressource
 1. Realtidsbehov
3. Hukommelse som begrænset ressource
4. Netværket som begrænset ressource
 1. Eksempel på protokol
5. Perspektivering
6. Opsummering og konklusion

Forklar indholdet af nedennævnte perspektiv og relationen til andre perspektiver

Ressourceperspektiv

DE FIRE PERSPEKTIVER (VIEWS) OVERBLIK

Forskellige perspektiver på hvordan *krav* og *designmål* kan opnås

Forskellige indgange til fornuftigt anvendelse af *teknologi* i design af distr. systemer.

Proces

- Proceshåndtering, tråde, samtidighed
- IPC, kommunikation på computerniv., sockets og porte
- Scheduling, blocking
- Meddelelsesbuffering

Design af distribuerede systemer

Kommunikation / netværk

- Netværk og protokoller
- Socket API
- RPC, struktureret kommunikation

Ressourcer

- Rationel / økonomisk anvendelse af knappe ressourcer
 - Processeringskapacitet (delt mellem lokale processer)
 - Båndbredde (fælles ressource)
 - Hukommelse (til prog. og data) vs. omkostning

Gennemgående: Teknikker til at opnå gennemsigtighed/gennemsuelighed (transparency).

RESSOURCEPERSPEKTIVET

OVERBLIK

Dette perspektiv omfatter

- Ressourcer i et computersystem, og deres betydning ifm. design og kommunikation i distribuerede systemer
- Rationel/økonomisk anvendelse af knappe ressourcer
 - Processor-ressourcen (compute)
 - Hukommelses-ressourcen
 - Netværks-ressourcen, båndbredde
 - Virtuelle ressourcer
 - (fx sockets (endpoints) porte (binding af service og PID), IP-adresser (binding til fysisk MAC-adresse)
- Designovervejelser omkring brug af ressourcer

PROCESSSERINGSKRAFT SOM BEGRÆNSET RESSOURCE

Distribuerede systemer

- SW skal evt. tilpasses tilgængelig ressourcer på begrænede platforme

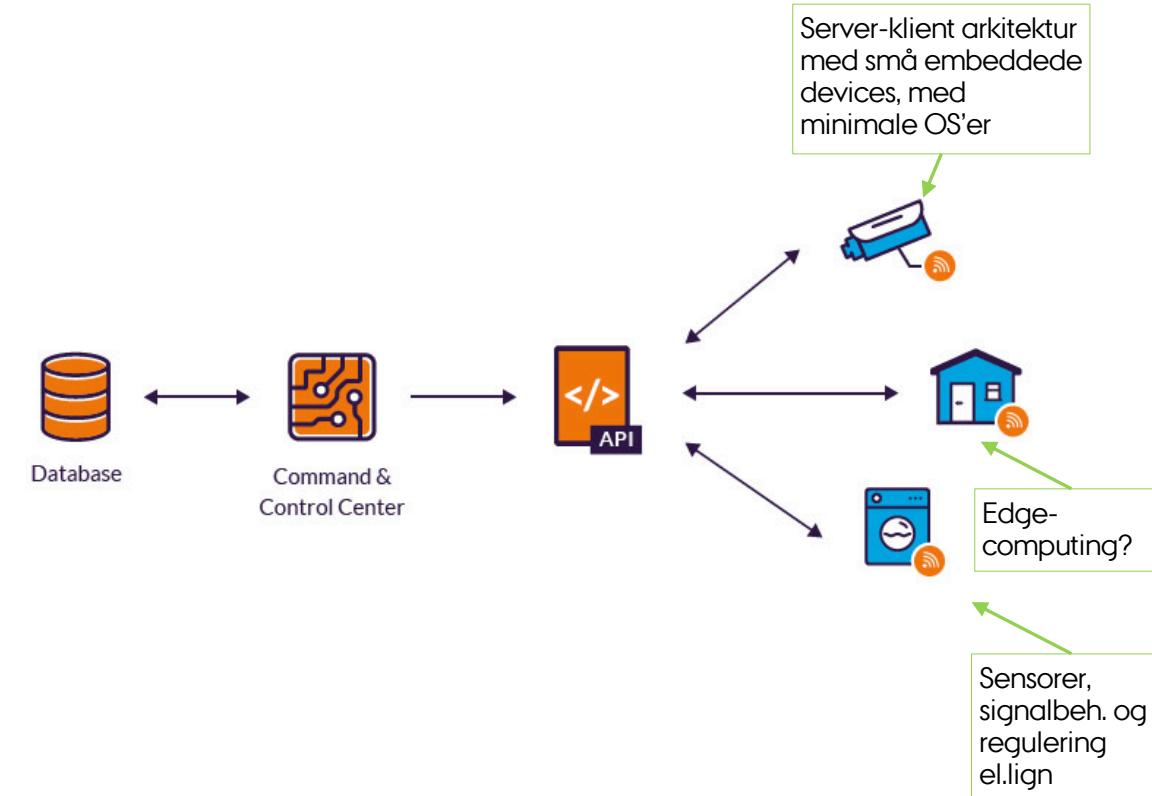
Specifikt for IOT-enheder

- Typiske små, constrained devices
- Dataprocessering på cloud (eller edge: fog/dew)
- Reeltidsbehov: Specifik hardware til DSP, osv.
- Trade-off i at gemme og sende rå data eller komprimeret / processeret

Mulige løsninger:

- Bedre programmeringsparadigme; fx SW/HW-interrupts, undgå busy-waiting/polling, undgå I/O blocking (fx non-blocking sockets), osv.
- Dedikerede hardware-peripherals
- Edge-computing.

Hvordan relaterer IOT til distribuerede systemer?



REALTIDSBEHOV

Realtidskrav

- Hard realtime
- Firm realtime
- Soft realtime

Hastighed på compute skal matche proces, fx ved regulering: Kan delvis løses med edge computing.

Men med **realtidskrav**:

- Når SW ikke er hurtig nok / kan stille garantier -> dedikeret hardware

Eksempel:

- 200 MHz PRU'er på BeagleBone i stedet for processering i Linux på CPU, firkantsignal ind.
- Assembler-kode på PRU, op til 5 MHz
- C-kode i Linux, op til 2-2,5 kHz
- ~ faktor 2000 forskel i frekvens, der kan håndteres

Sammenligning af performance

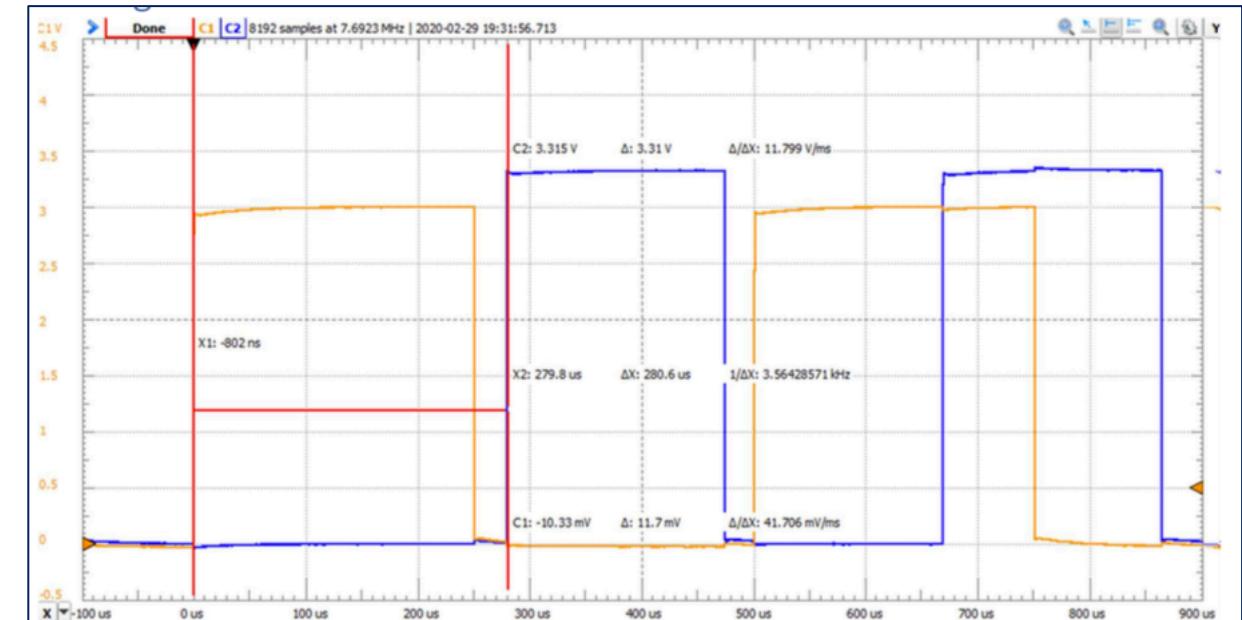
LATENCY VED 1 MHZ FIRKANTSIGNAL

Ved 1 MHz ligger latency
stadic omkring 40 ns

PRU kan stadic følge med!

I nærheden af HF-delen i RF-
båndet...

- Ikke-ideelle effekter begynder at spille ind
- Selv-induktive effekter i ledninger
- Kapacitans i breadboard, m.m.



Figur 4 - Måling af OS-styret in/output delay ved 2KHz

Ved 2Khz ses et delay på $280\mu s$.

HUKOMMELSE SOM BEGRAÆNSET RESSOURCE

Fysisk:

- Constraints relativt til systemkrav og økonomi (co-design ☺)
 - Hurtigt / stor mængde -> dyr løsning
 - Meget constrained -> måske dyr udvikling
 - Low level assembler, HW-specifikt, proprietary DSP, osv.
- Memory-model:
 - Må der allokeres dynamisk eller kun statisk?
 - Statisk (stack) allokeres med det samme, permanent. Dynamisk (heap), mindre umiddelbart pladskrav.
 - Fx FreeRTOS med meget begrænset plads, risiko for fragmentering.
 - Kritiske embedded systemer:
 - Må man free efter flyet er lettet? Må man free i en pacemaker?
 - Hvad hvis man laver en access violation / segfault?

OS:

- User-space
 - Lad være at lave memory leaks ☺
 - Tråde/delte ressourcer: Undgå data races. ACID (brug låse/mutexes, atomic access).
- Kernel-space (mere begrænset), fx ift. buffers/ IO-strømme
- Virtuel (swapning ind/ud, langsommere)

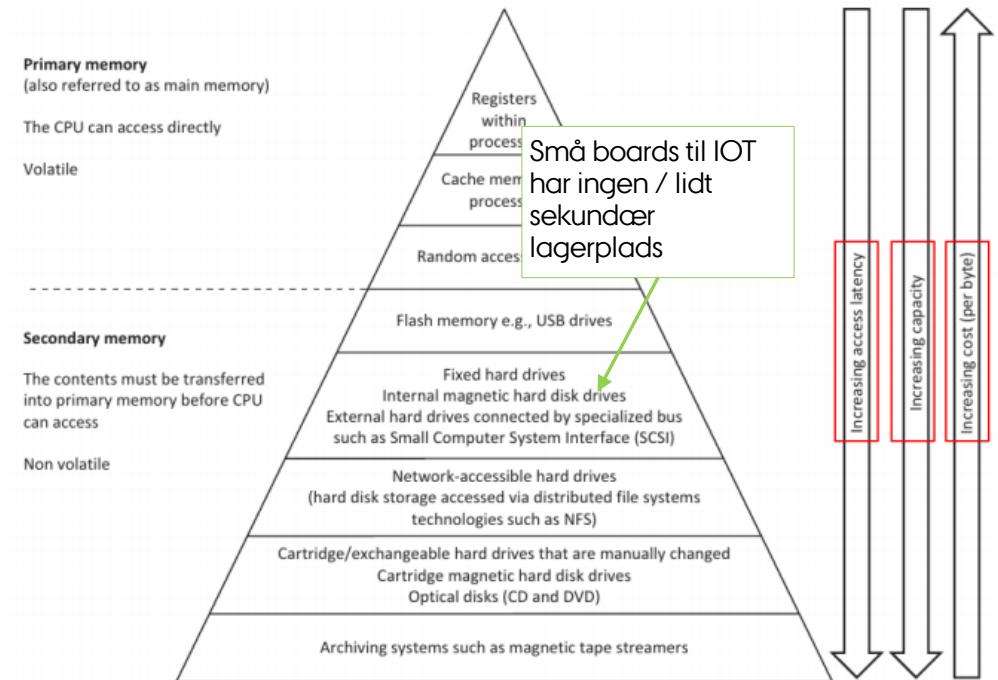


FIGURE 4.7

The memory hierarchy.

NETVÆRKET SOM BEGRÆNSET RESSOURCE

Problemer / udfordringer:

- Båndbredde tilgængelig, effektivt throughput
- Latency, forsinkelse på netværk
- Transmissionsfejl / Package loss

Design-overvejelser

- Behov for garanteret levering?
 - TCP vs UDP (mindre header, hurtigere)
 - UDP kan broadcaste
- Behov for kryptering / sikkehed (TLS)?
- Overfør kun nødvendig data
 - Komprimeret versus rå
 - Downsampling
- Hvis stort package loss observeres (upålideligt netværk, dårlig dækning):
 - Overvej mindre pakkestørrelse vs. overhead (TCP)
- Store pakker fragmenteres
 - Ethernet-eksempel
 - Max frame på ethernet er 1518 bytes, heraf er 18 overhead -> MTU 1500 bytes

Frame 2: 1452 bytes on wire (11616 bits), 1452 bytes captured (11616 bits) on interface en0, id 0
Interface id: 0 (en0)
Encapsulation type: Ethernet (1)
Arrival Time: Jun 26, 2020 15:48:39.537322000 +08
[Time shift for this packet: 0.000000000 seconds]
Epoch Time: 1593157719.537322000 seconds
[Time delta from previous captured frame: 0.128622000 seconds]
[Time delta from previous displayed frame: 0.128622000 seconds]
[Time since reference or first frame: 0.128622000 seconds]
Frame Number: 2
Frame Length: 1452 bytes (11616 bits)
Capture Length: 1452 bytes (11616 bits)
[Frame is marked: False]
[Frame is ignored: False]
[Protocols in frame: eth:ethertype:ip:tcp:tls]
[Coloring Rule Name: TCP]
[Coloring Rule String: tcp]
Ethernet II, Src: Arcadyan_4d:da:a2 (e4:3e:d7:4d:da:a2), Dst: Apple_2a:9a:a1 (c8:2a:14:2a:9a:a1)
Destination: Apple_2a:9a:a1 (c8:2a:14:2a:9a:a1)
Source: Arcadyan_4d:da:a2 (e4:3e:d7:4d:da:a2)
Type: IPv4 (0x0800)
Internet Protocol Version 4, Src: 172.67.75.43, Dst: 192.168.1.7
0100 = Version: 4
.... 0101 = Header Length: 20 bytes (5)
Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
Total Length: 1438
Identification: 0xc4b7 (50359)
Flags: 0x4000, Don't fragment
Fragment offset: 0
Time to live: 53
Protocol: TCP (6)
Header checksum: 0xc284 [validation disabled]
[Header checksum status: Unverified]
Source: 172.67.75.43
Destination: 192.168.1.7
Transmission Control Protocol, Src Port: 443, Dst Port: 59062, Seq: 1, Ack: 1, Len: 1398
Transport Layer Security

PROTOKOL FOR KOMMUNIKATION OVER MQTT, PRO4

Case: Overføre kommandoer og data mellem Webinterface og fysiske prøvestande i laboratorium.

Payload i en MQTT-besked (max 256 MiB)

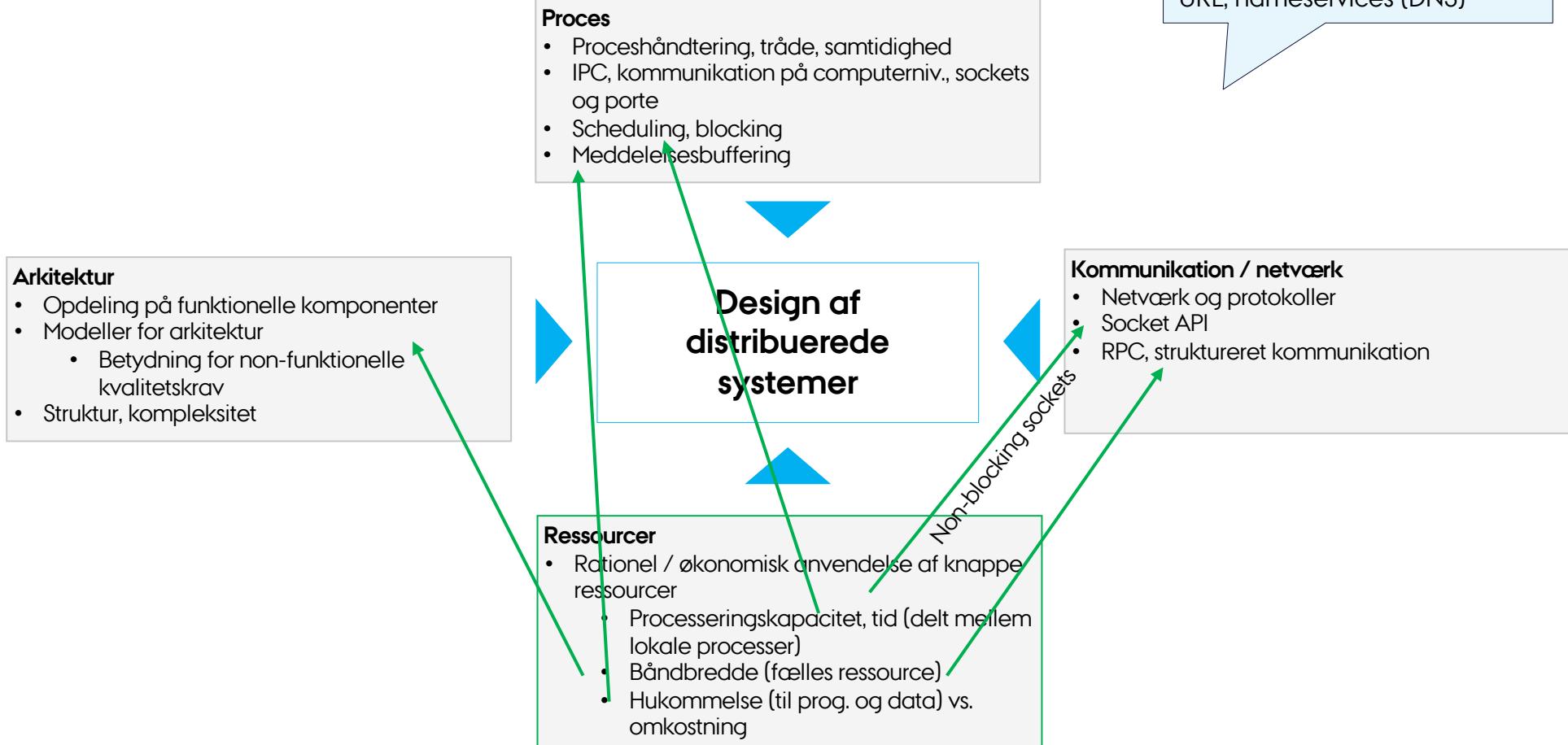
- Protokol-indpakning: UTF-8, JSON, MQTT, TCP
 - -> også data
- Variabel længde PDU (protocol data unit)
- Fordeler:
 - Fleksibel struktur
 - Matchet til database
- Ulemper:
 - Fylder meget mere end en tætpakket, specifik binærrepræsentation kunne (hvis man vidste præcis, hvad der skulle sendes)
 - Kræver serialisering og parsing (CPU-tid)

Version:	JSONschema_v1.1
schema	<pre>{ "\$schema": "AUTeam2 JSONschema", "title": "PayloadSchema", "type": "object", "properties": { "protocolVersion": {"type": "number"}, "sentBy": {"type": "string"}, "msgType": {"type": "string"}, "commandList": {"type": "array"}, "statusCode": {"type": "string"}, "parameterObj": {"type": "object"}, "dataObj": {"type": "object"} }, "required": ["protocolVersion", "sentBy", "msgType", "statusCode"] }</pre>

DE FIRE PERSPEKTIVER (VIEWS) OVERBLIK

Forskellige perspektiver på hvordan *krav* og *designmål* kan opnås

Forskellige indgange til fornuftigt anvendelse af *teknologi* i design af distr. systemer.



Gennemgående: Teknikker til at opnå gennemsigtighed/gennemsuelighed (transparency).

Lokation, fx ved brug af URI, URL, nameservices (DNS)

OPSUMMERING OG KONKLUSION

SW5: DISTRIBUEREDE SYSTEMER: NETVÆRKS-/KOMMUNIKATIONSPERSPEKTIVET

Forklar indholdet af nedennævnte perspektiv og relationen til andre perspektiver

[Netværksperspektivet](#)

Indhold

1. Overblik
2. Begreber
 1. Kommunikationstyper
 2. Kommunikationsmodeller
3. Anvendelse på eksempler
 1. Temperaturserver
 2. System fra PRO4
4. Perspektivering
5. Opsummering og konklusion

DE FIRE PERSPEKTIVER (VIEWS) OVERBLIK

Forskellige perspektiver på hvordan *krav* og *designmål* kan opnås

Forskellige indgange til fornuftigt anvendelse af *teknologi* i design af distr. systemer.

Proces

- Proceshåndtering, tråde, samtidighed
- IPC, kommunikation på computerniv., sockets og porte
- Scheduling, blocking
- Meddelelsesbuffering

Design af distribuerede systemer

Arkitektur

- Opdeling på funktionelle komponenter
- Modeller for arkitektur
 - Betydning for non-funktionelle kvalitetskrav
- Struktur, kompleksitet

Kommunikation / netværk

- Netværk og protokoller
- Socket API
- RPC, struktureret kommunikation

Ressourcer

- Rationel / økonomisk anvendelse af knappe ressourcer
 - Processeringskapacitet (delt mellem lokale processer)
 - Båndbredde (fælles ressource)
 - Hukommelse (til prog. og data) vs. omkostning

Gennemgående: Teknikker til at opnå gennemsigtighed/gennemsuelighed (transparency).

NETVÆRKS- /KOMMUNIKATIONSPERSPEKTIVET OVERBLIK

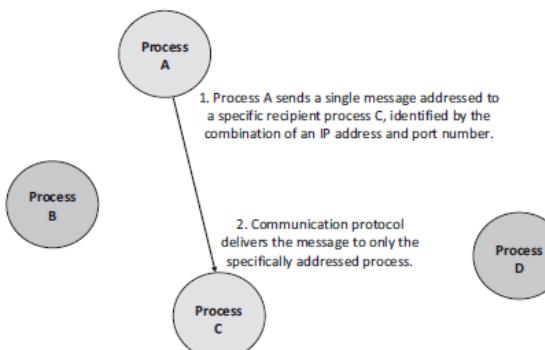
Dette perspektiv omfatter

- Netværk og netværksprotokoller, herunder TCP og UDP
- Kommunikationstyper (unicast, multicast, broadcast)
- Socket API
- Remote Procedure Calls (RPC)

KOMMUNIKATIONSTYPER / SENDETYPER

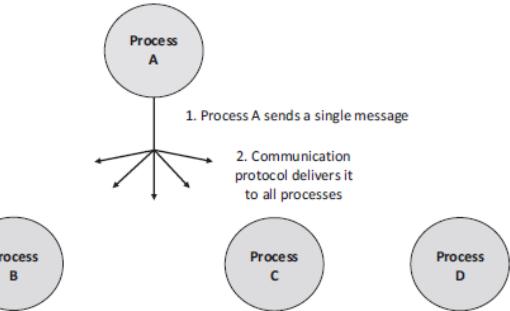
Unicast

- Send til 1 specifik adresse



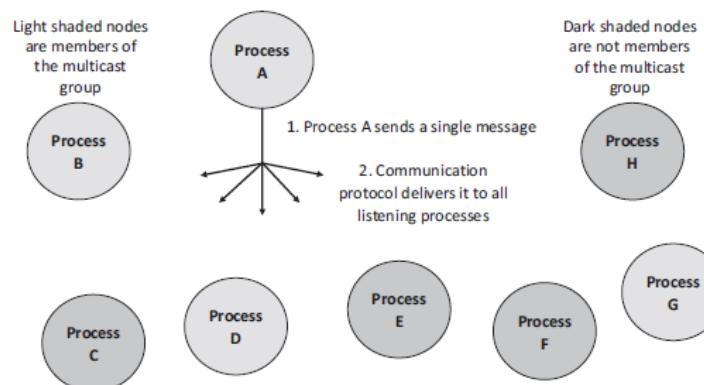
Multicast

- En adresse/destination dækker en **gruppe** af processer
 - > Lokationsgennemsigtighed



Broadcast

- Send til alle som lytter (på bestemt subnet)
- En adresse/destination dækker **alle** processer
 - Fx broadcasting på ethernet



Anycast

- Mindst én skal modtage besked

FIGURE 3.8
Broadcast communication.

FIGURE 3.7
Unicast communication.

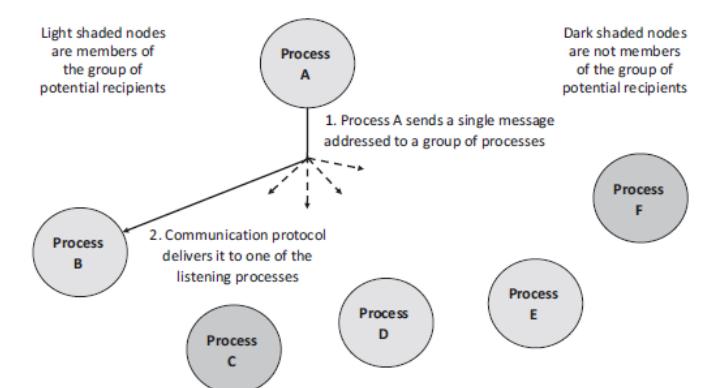


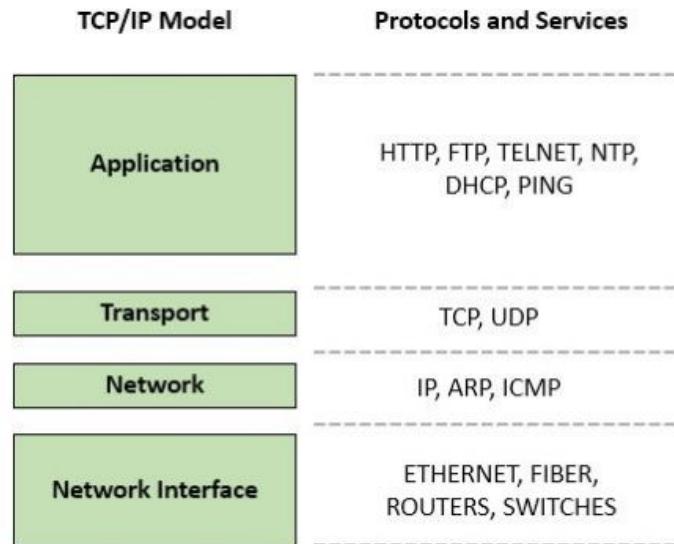
FIGURE 3.11
Anycast communication.

FIGURE 3.10
Multicast communication.

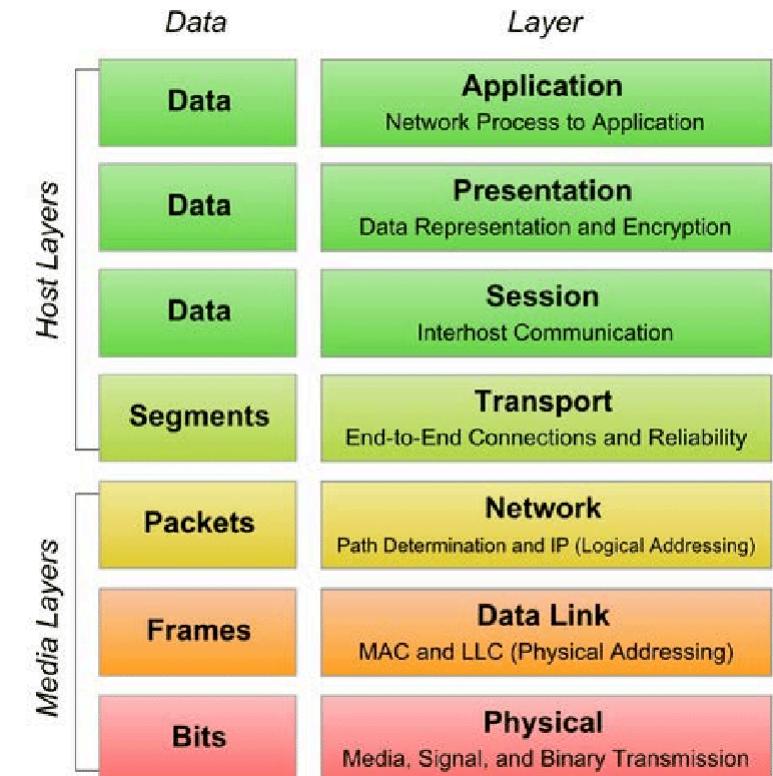
LAGOPDELT KOMMUNIKATION

ABSTRAKTIONER OG ANSVARSFORDELING

TCP/IP-modellen



OSI-modellen



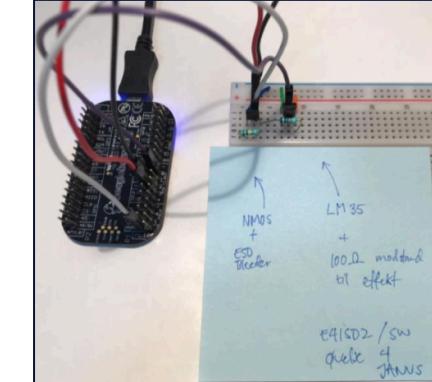
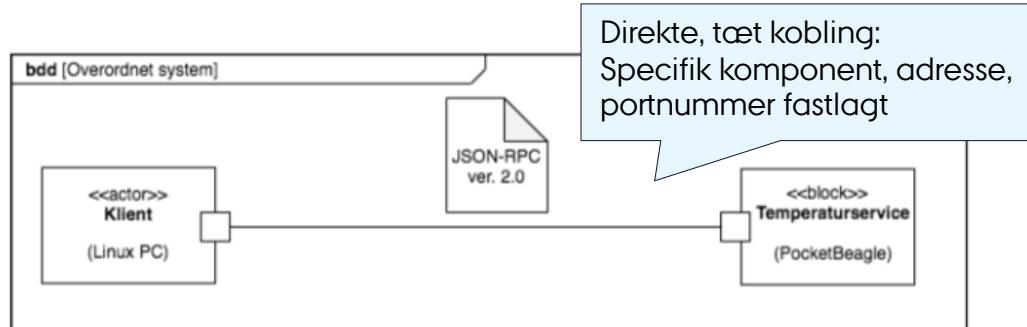
Indpakning med headers og frames

EKSEMPEL 1: TEMPERATURSERVER

KLIENT-SERVER-KONSTRUKTION, TEMP SERVER MED RPC

Temperaturserver

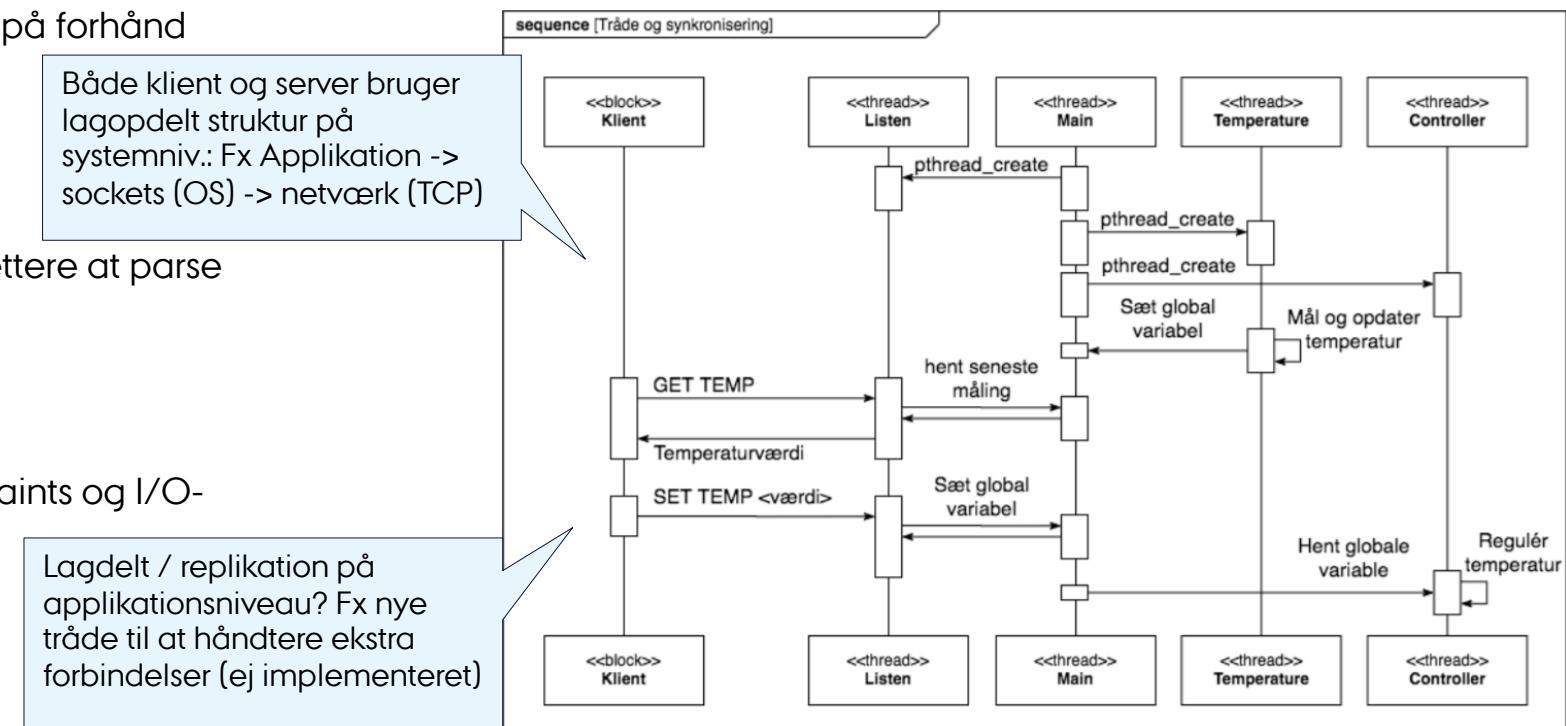
- Tætkoblet klient-server-arkitektur
 - Server på Beaglebone, adresse/port fastlagt på forhånd
 - Protokol: JSON-RPC over TCP-sockets
 - Trådet applikation



Formål med JSON-RPC: Struktureret kommunikation, lettere at parse syntax / grammatik for kommandoer og svar

Formål med brug af sockets og tråde i arkitekturen:

- Sockets: Kommunikere over netværk (TCP)
- Tråde: Opdele arbejde med forskellige tidsconstraints og I/O-typer.



EKSEMPEL 1: STRUKTURERET KOMMUNIKATION MELLEM KLIENT OG SERVER

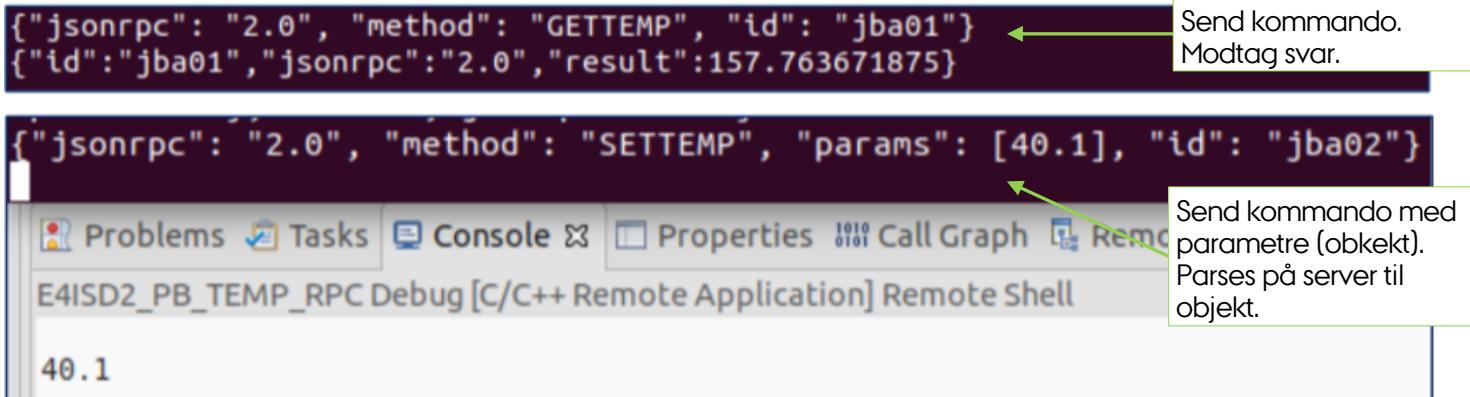
Specifikation: JSON-RPC 2.0

- Specifierer gyldige (og ugyldige) beskeder klient <-> server
- Kan indlejre objekter
- Relativt simpel, parses med gængse JSON-parsere

Andre metoder til at strukturere kommunikation

- REST (stateless)
- SOAP over HTTP, osv
- XML

Eksempel

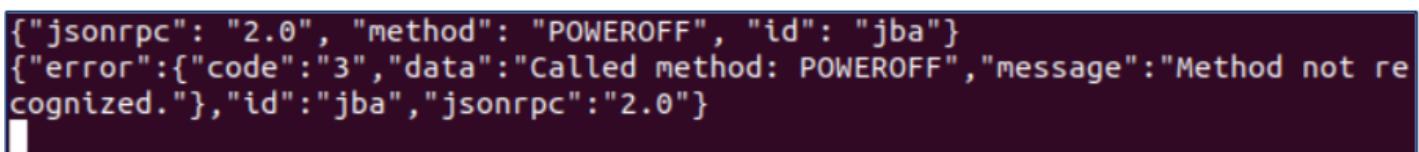


The screenshot shows a C/C++ IDE interface with several tabs: Problems, Tasks, Console, Properties, Call Graph, and Remote. The Remote tab is active, displaying the text "E4ISD2_PB_TEMP_RPC Debug [C/C++ Remote Application] Remote Shell". In the console area, three JSON-RPC messages are shown:

```
{"jsonrpc": "2.0", "method": "GETTEMP", "id": "jba01"}  
{"id": "jba01", "jsonrpc": "2.0", "result": 157.763671875}  
  
{"jsonrpc": "2.0", "method": "SETTEMP", "params": [40.1], "id": "jba02"}  
40.1
```

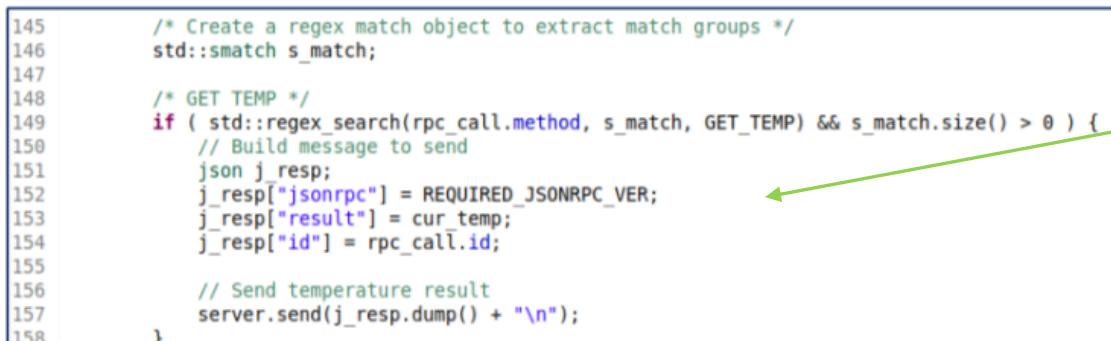
Annotations with arrows point to the messages:

- An arrow points from the first message to the text "Send kommando. Modtag svar."
- An arrow points from the second message to the text "Send kommando med parametre (objekt). Parses på server til objekt."



The screenshot shows a C/C++ IDE interface with the Remote tab active, displaying the text "E4ISD2_PB_TEMP_RPC Debug [C/C++ Remote Application] Remote Shell". In the console area, a JSON-RPC error message is shown:

```
{"jsonrpc": "2.0", "method": "POEROFF", "id": "jba"}  
{"error": {"code": 3, "data": "Called method: POEROFF", "message": "Method not recognized."}, "id": "jba", "jsonrpc": "2.0"}
```



The screenshot shows a C++ code editor with code related to JSON-RPC processing:

```
145     /* Create a regex match object to extract match groups */  
146     std::smatch s_match;  
147  
148     /* GET TEMP */  
149     if ( std::regex_search(rpc_call.method, s_match, GET_TEMP) && s_match.size() > 0 ) {  
150         // Build message to send  
151         json j_resp;  
152         j_resp["jsonrpc"] = REQUIRED_JSONRPC_VER;  
153         j_resp["result"] = cur_temp;  
154         j_resp["id"] = rpc_call.id;  
155  
156         // Send temperature result  
157         server.send(j_resp.dump() + "\n");  
158     }
```

Annotations with arrows point to the code:

- An arrow points from the final line of code to the text "Indpakning af et svar, der overholder JSON-RPC."

EKSEMPEL 2: FRA PRO4

WEBINTERFACE FOR FYSISKE PRØVESTANDE

Containeriseret/
Docker-drevet
system

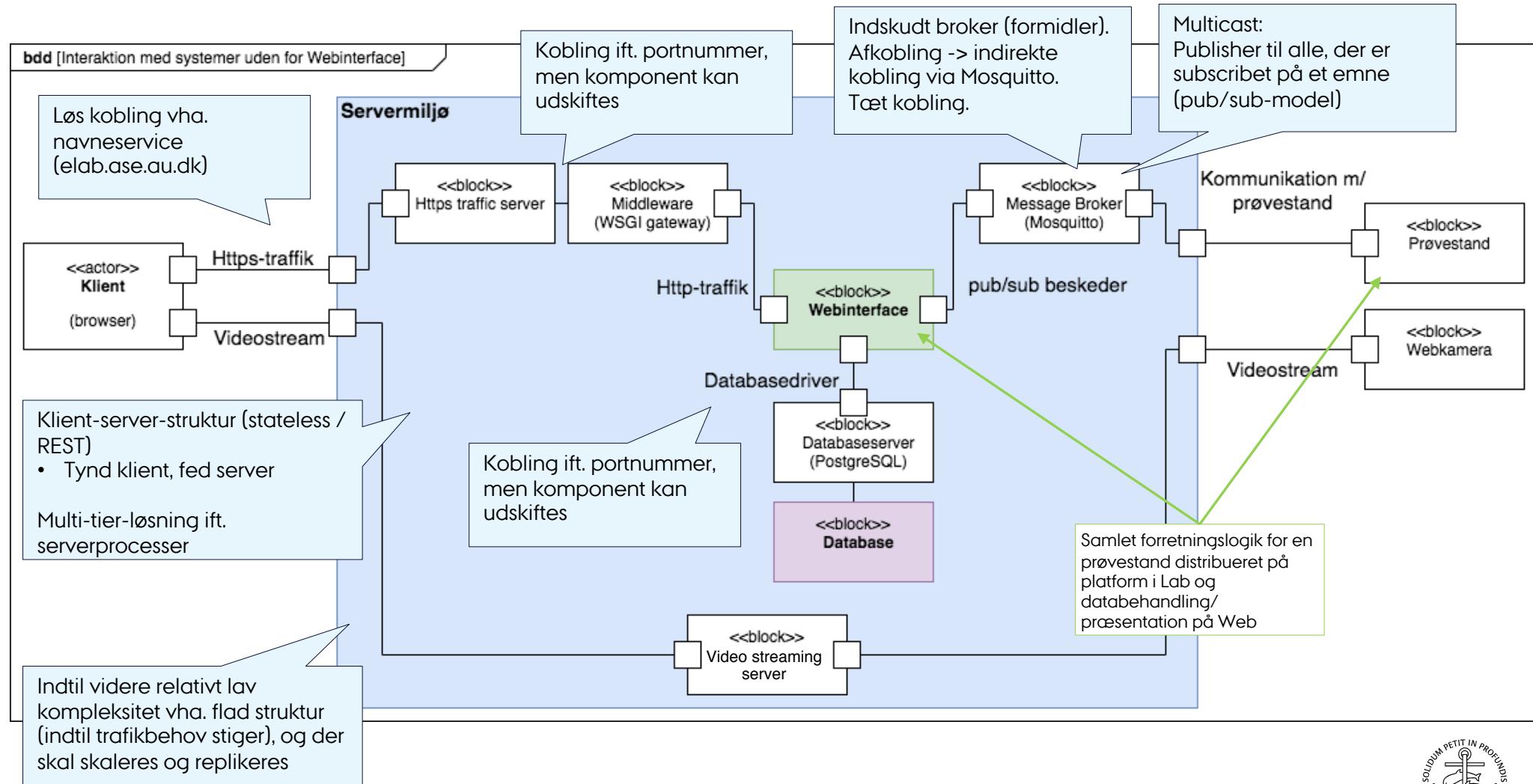
- Mikroservices

Systemet kunne
skaleres op,

Replikation af
services

Mest vanskelige at
skalere er
databasen

- Replikations-
metoder



PROTOKOL FOR KOMMUNIKATION OVER MQTT, PRO4

Mange devices med mange udviklerteams

En enkelt backend-server, et udviklerteam

Protokollen skal strukturere kommunikation til

- Overførsel af kommandoer og data

Fordel ved struktureret kommunikation og
specificeret interface

- Entydigt for udviklere, hvad der skal udvikles
op imod.
- En parser, et sæt objekter på backend,
genbrug, standardisering.
- Databasen tilpasset kommunikationen.

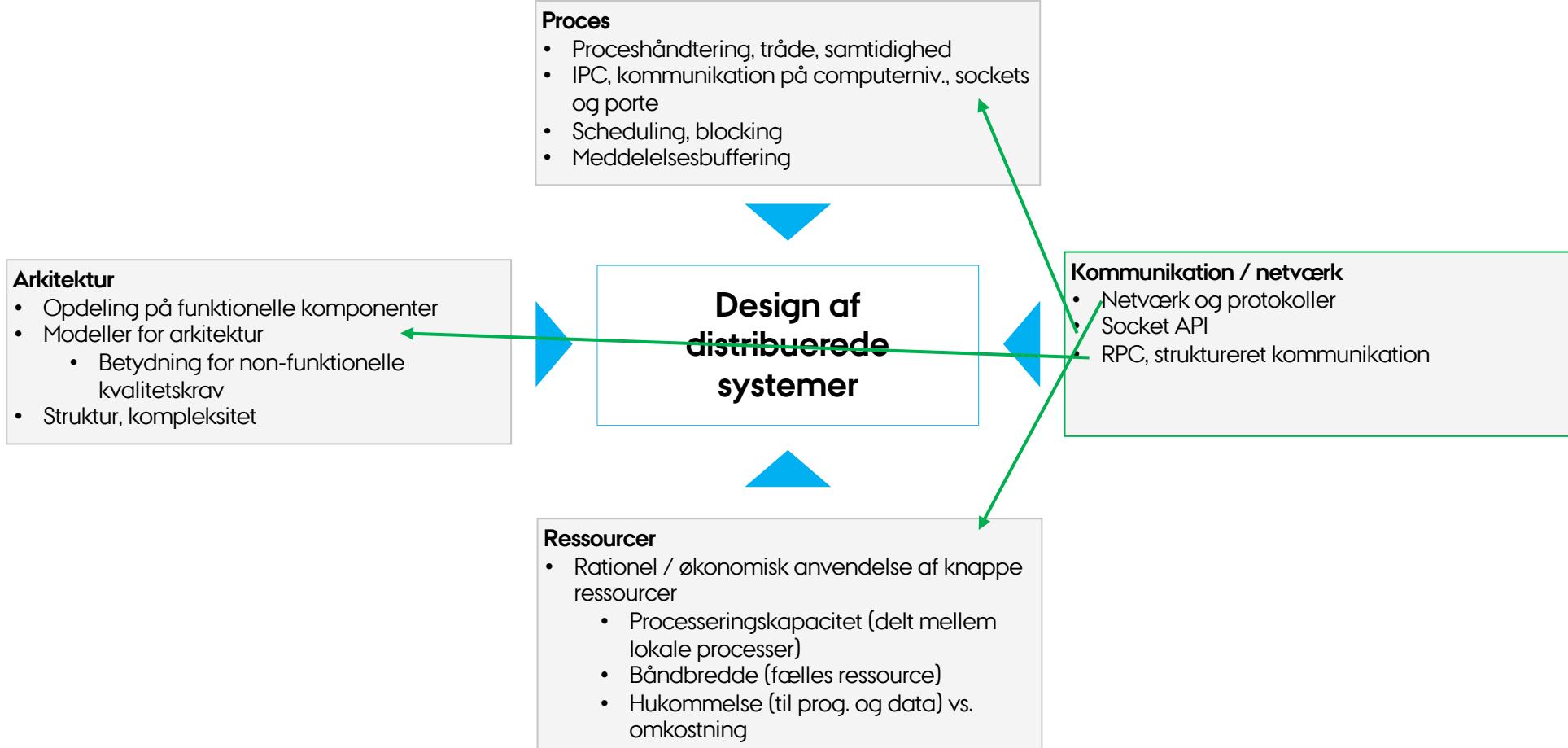
Version:	JSONschema_v1.1
schema	<pre>{ "\$schema": "AUTeam2 JSONschema", "title": "PayloadSchema", "type": "object", "properties": { "protocolVersion": {"type": "number"}, "sentBy": {"type": "string"}, "msgType": {"type": "string"}, "commandList": {"type": "array"}, "statusCode": {"type": "string"}, "parameterObj": {"type": "object"}, "dataObj": {"type": "object"}<br "msgtype",="" "required":="" "sentby",<br="" "statuscode"]<br="" ["protocolversion",="" },<br=""/>}</pre>

DE FIRE PERSPEKTIVER (VIEWS) PERSPEKTIVERING

Forskellige perspektiver på hvordan *krav* og *designmål* kan opnås

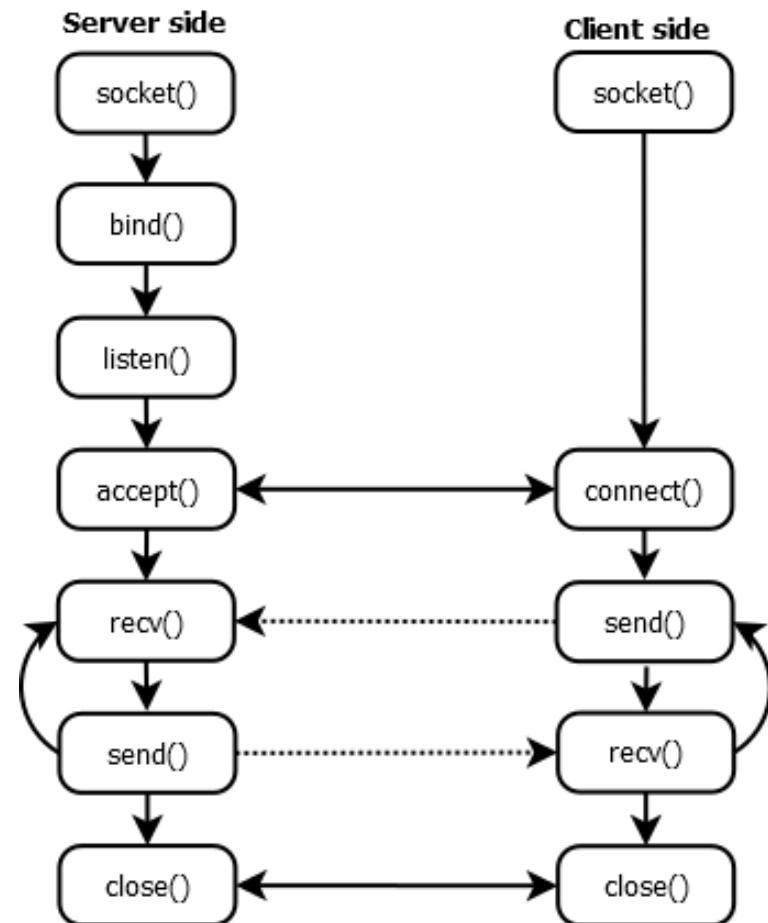
Forskellige indgange til fornuftigt anvendelse af *teknologi* i design af distr. systemer.

Gennemgående: Teknikker til at opnå gennemsigtighed/gennemsuelighed (transparency).



OPSUMMERING OG KONKLUSION

BILAG TIL SOCKETS OG TRÅDE



SOCKETS

SocketServer – klasse til serverfunktion

```
16@/**  
17 * @class SocketServer  
18 * @brief A class that encapsulates a server socket for network communication  
19 */  
20@class SocketServer {  
1 private:  
22 int portNumber;  
23 int socketfd, clientSocketfd;  
24 struct sockaddr_in serverAddress;  
25 struct sockaddr_in clientAddress;  
26 bool clientConnected;  
27  
28 public:  
29 SocketServer(int portNumber);  
30 virtual int listen();  
31 virtual int send(std::string message);  
32 virtual std::string receive(int size);  
33 virtual void closeClient();  
34  
35 virtual ~SocketServer();  
36 };
```

Objekt instantieres med portnummer

Wrappers til 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 }  
83  
84@string SocketServer::receive(int size=1024){  
85 char readBuffer[size];  
86 memset(readBuffer, '\0', size);  
87 int n = read(this->clientSocketfd, readBuffer, sizeof(readBuffer));  
88 if (n < 0){  
89 perror("Socket Server: error reading from server socket.");  
90 throw "Socket probably closed.";  
91 }  
92 return string(readBuffer);  
93 }
```

Wrapper de underliggende C-kald, og oversetter til C-strukturer

Opsætning og indkommende requests (listen())

```
25@ /* listen creates the socket and  
26 * blocks until an incoming connection established,  
27 * and then accepts the incoming connection */  
28@ int SocketServer::listen(){  
29  
30 /* Make INET (TCP) socket */  
31 this->socketfd = socket(AF_INET, SOCK_STREAM, 0);  
32  
33 /* Check that socket created OK */  
34 if (this->socketfd < 0){  
35 perror("Socket Server: error opening socket.\n");  
36 return 1;  
37 }  
38  
39@ /* Ensure all zeros in unused part of serverAddress  
40 * this is an alternative to memset */  
41 bzero((char *) &serverAddress, sizeof(serverAddress));  
42  
43 /* Create struct for address */  
44 serverAddress.sin_family = AF_INET;  
45 serverAddress.sin_addr.s_addr = INADDR_ANY;  
46 serverAddress.sin_port = htons(this->portNumber);  
47  
48 /* Attempt to bind to the socket address */  
49 if (bind(socketfd, (struct sockaddr *) &serverAddress, sizeof(serverAddress)) < 0) {  
50 perror("Socket Server: error on binding the socket.\n");  
51 return 1;  
52 }  
53  
54 /* system call listen */  
55 ::listen(this->socketfd, 5);  
56  
57 /* Update address of client */  
58 socklen_t clientLength = sizeof(this->clientAddress);  
59  
60 /* Accept connection */  
61 this->clientSocketfd = accept(this->socketfd,  
62 (struct sockaddr *) &this->clientAddress,  
63 &clientLength);  
64  
65 /* Check that client socket is OK */  
66 if (this->clientSocketfd < 0){  
67 perror("Socket Server: Failed to bind the client socket properly.\n");  
68 return 1;  
69 }  
70  
71 }
```

Ingen specifik IP-adresse

Sat i headerfil, men kunne også være argument

Backlog, tillader kø på 5 ventende forbindelser

TRÅDE

Opstart og nedlukning af tråde

```
58 int main(void) {
59
60     /* Set up signal handler */
61     if ( signal(SIGHUP, sig_handler) == SIG_ERR ) {
62         std::cout << "Can't register SIGHUP" << std::endl;
63     }
64
65     pthread_t listen_thread;
66     if ( pthread_create(&listen_thread, NULL, &listen_thread_function, NULL) ) {
67         std::cout << "Could not create thread" << std::endl;
68         return EXIT_FAILURE;
69     }
70
71     pthread_t temp_thread;
72     if ( pthread_create(&temp_thread, NULL, &temp_monitor_thread_function, NULL) ) {
73         std::cout << "Could not create thread" << std::endl;
74         return EXIT_FAILURE;
75     }
76
77     pthread_t control_thread;
78     if ( pthread_create(&control_thread, NULL, &controller_thread_function, NULL) ) {
79         std::cout << "Could not create thread" << std::endl;
80         return EXIT_FAILURE;
81     }
82
83     /* Temperature thread has been stopped by SIGHUP */
84     pthread_join(temp_thread, NULL);
85
86     /* Controller thread has been stopped by SIGHUP */
87     pthread_join(control_thread, NULL);
88
89     /* Safe to kill the listener now */
90     pthread_cancel(listen_thread);
91     pthread_join(listen_thread, NULL);
92
93     return 0;
94 }
```

Benytter POSIX-tråde (pthreads)

Opstart af trådfunktioner

Tråde afkobles ikke fra hoved-tråden,
men synkroniseres ved nedlukning.

Når regulator-tråden er død, kan
resten lukkes ned.

Synkronisering af variable og nedlukning

```
44 /* Coordinate clean exit : */
45 std::atomic<bool> exit_now(false);
46
47 void sig_handler(int signo) {
48
49     // React to caught signal
50     if (signo == SIGHUP) {
51         std::cout << "Received SIGHUP. Bye!" << std::endl;
52
53         // Tell all threads to end now, cleanly
54         exit_now.store(true);
55     }
56 }
21 /* These enable communication between threads */
22 extern float cur_temp;
23 extern float set_point;
24 extern std::atomic<bool> exit_now;
25
26 /* @brief Thread function to monitor temperature in the background
27 * update the global variable CUR_TMP
28 */
29
30 void * temp_monitor_thread_function(void * value) {
31
32     int adc_value;
33     float temp;
34
35     while( !exit_now.load() ) {
36
37         // Take a sample and convert to temperature
38         adc_value = read_analog(ADC);
39         temp = conv_temperature(adc_value);
40
41         // Update the global variable
42         cur_temp = temp;
43
44         sleep(TEMP_SAMPLE_INTERVAL);
45     }
46
47     pthread_exit(NULL);
48 }
```

Atomisk variabel til
koordinere nedlukning.

Globale variable, header fil.
Her kunne overvejes mutex eller
en atomisk variabel.

Tjek for nedlukning.

Opdatér global variabel.

SW6: DISTRIBUEREDE SYSTEMER: ARKITEKTURPERSPEKTIVET

Indhold

1. Overblik
2. Begreber og koncepter
 1. Lagopdeling og struktur
 2. Kobling
3. Anvendelse af begreber på eksempler
 1. Temperaturserver
 2. System fra PRO4
4. Perspektivering
5. Opsummering og konklusion

Forklar indholdet af nedennævnte perspektiv og relationen til andre perspektiver

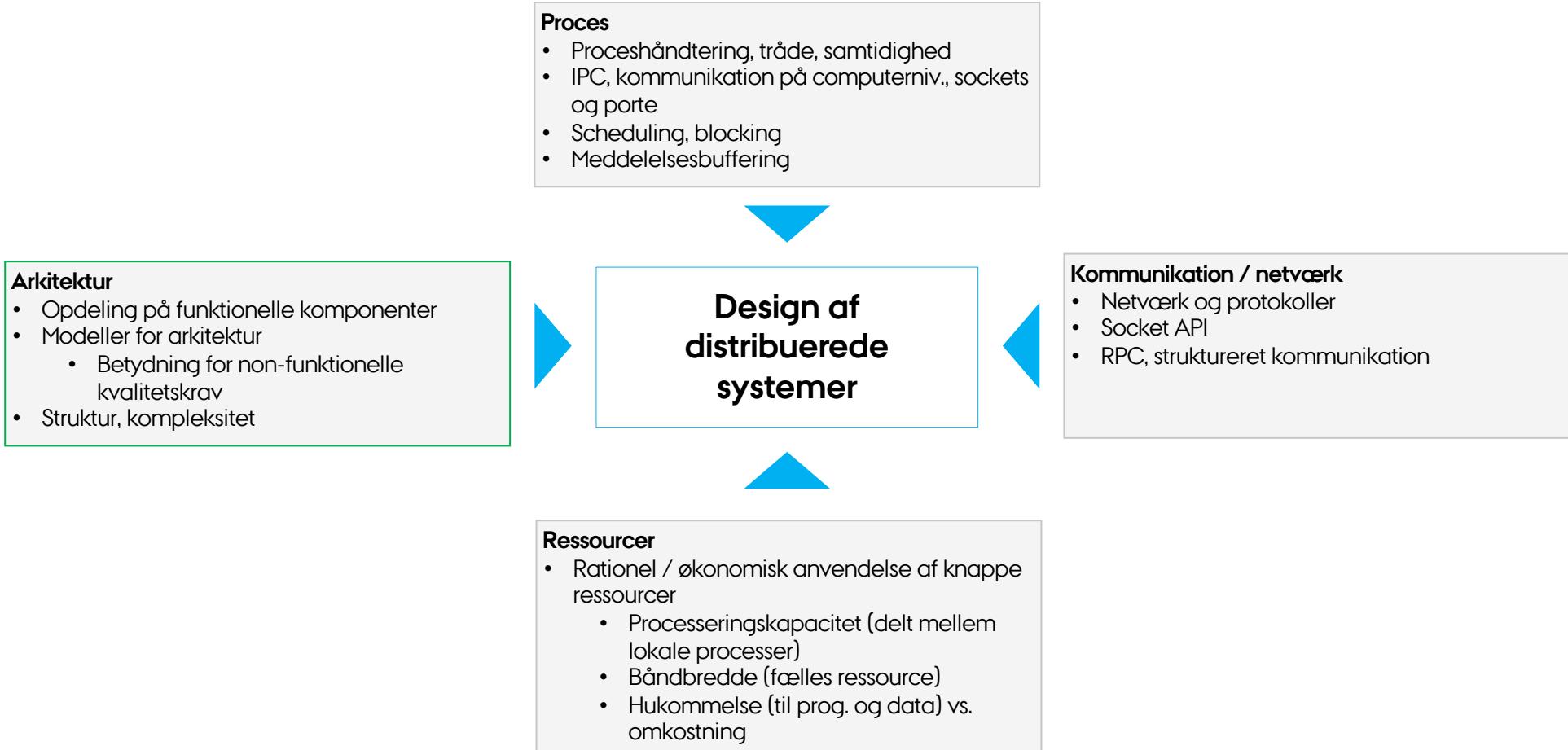
Arkitekturperspektivet

DE FIRE PERSPEKTIVER (VIEWS) OVERBLIK

Forskellige perspektiver på hvordan *krav* og *designmål* kan opnås

Forskellige indgange til fornuftigt anvendelse af *teknologi* i design af distr. systemer.

Gennemgående: Teknikker til at opnå gennemsigtighed/gennemsuelighed (transparency).



ARKITEKTURPRESPEKTIVET

OVERBLIK

Def.: Arkitektur er beskrivelse af placering og forbindelse mellem komponenter i et system. Logisk og fysisk.

Arkitekturperspektivet indeholder

- Opdeling af applikationer på funktionelle komponenter
- Modeller for arkitektur <- design for at opnå krav (funktionelle og non-funktionelle)
 - Placering, forbindelser og typer af komponenter: Peer-to-peer, klient-server, multi-tier, hybrid
 - Betydning for *non-funktionelle krav* / kvalitet:
 - Skalérbarhed, Robusthed, Tilgængelighed, Ydeevne/Efficiens, Usability/brugbarhed, Transparens.
 - Transparens er vigtigt i distribuerede systemer: Detaljer omkring distribuering, multiplicitet og kompleksitet er gemt for brugeren af systemet. Fremstår som et enkelt, samlet lokalt system.

LAGOPDELING OG STRUKTUR

Lagopdeling

- Udnyttelse af services i OS (softwarebiblioteker)
- Giver struktur (og genbrug) at lagopdele
- Fordeling af ansvar
- Standardiseret adfærd

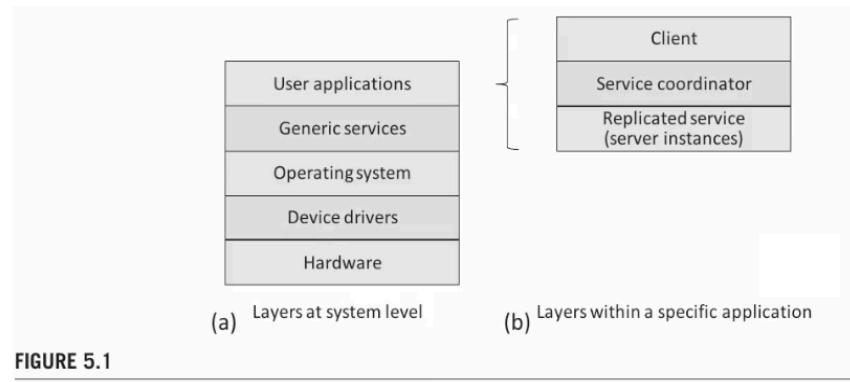
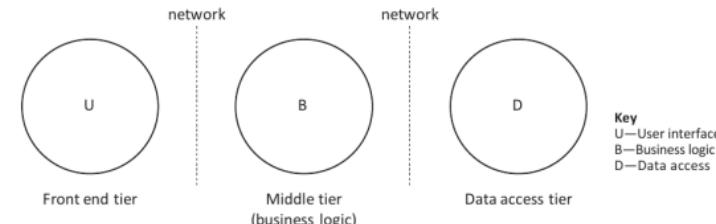
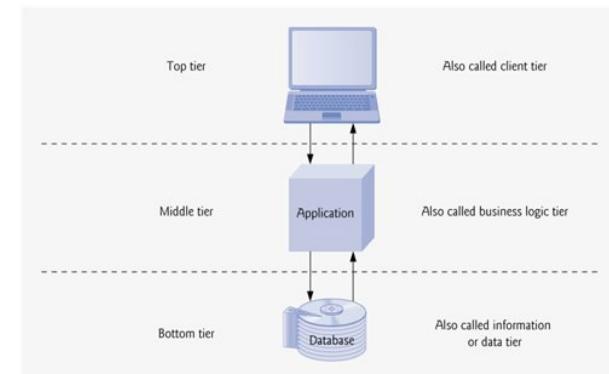


FIGURE 5.1

Struktur for komponenter

- Peer-to-peer (to af samme type)
- Klient-server (aktiv-reakтив)
- Multi-tier (adskillige separate komp.)
- Hybrid (blanding, hvor fx klienterne også taler sammen)

Trade-off:
Gennemsigtighed kræver
mere kompleksitet



Taksonomi af "klasser"

Heterogenitet af komponenter

KOBLINGER

Kobling mellem komponenter

- **Tæt**: Kobling til spec. komp. def. på designtidspunkt
- **Løs**: Ej. def. på designtidspunkt.
 - Discovery, nameservices, osv.
- **Direkte**: Typisk TCP eller UDP.
- **Indirekte**: Gennem broker eller middleware
- Isoleret:

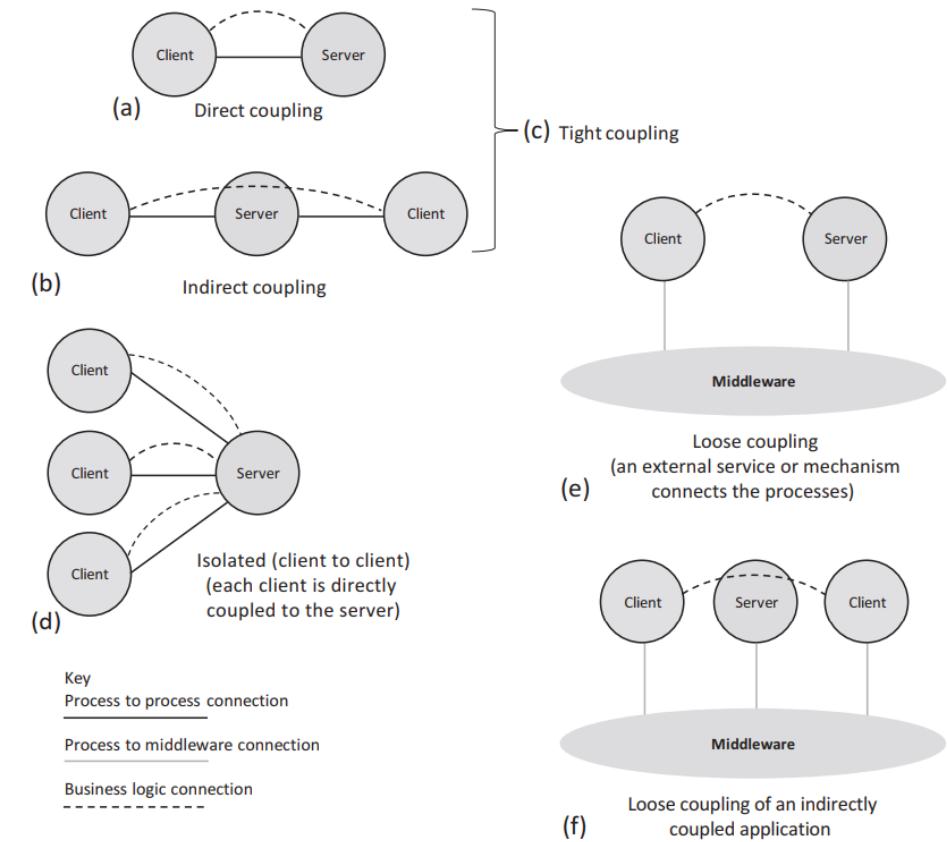


FIGURE 5.10

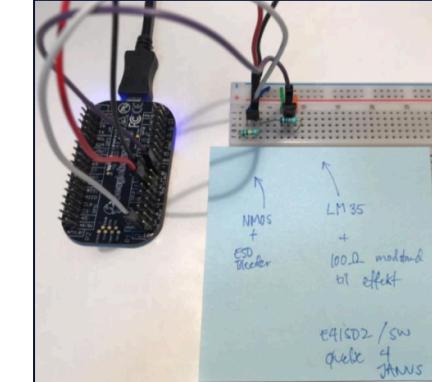
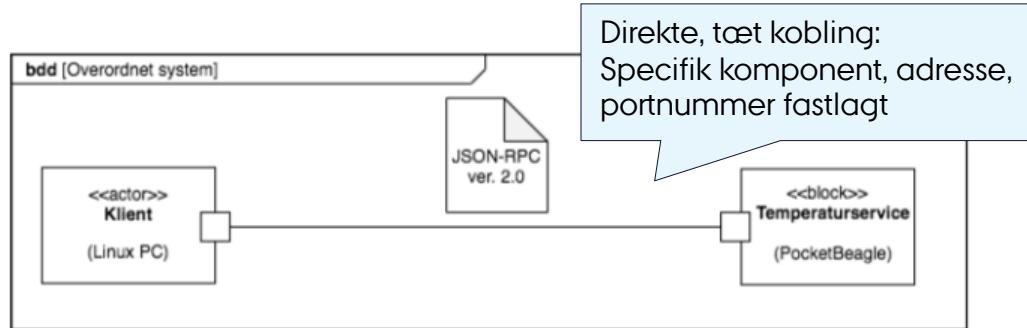
Coupling variations, illustrated in the context of a client-server application.

EKSEMPEL 1: TEMPERATURSERVER

KLIENT-SERVER-KONSTRUKTION, TEMP SERVER MED RPC

Temperaturserver

- Tætkoblet klient-server-arkitektur
 - Server på Beaglebone, adresse/port fastlagt på forhånd
 - Protokol: JSON-RPC over TCP-sockets
 - Trådet applikation

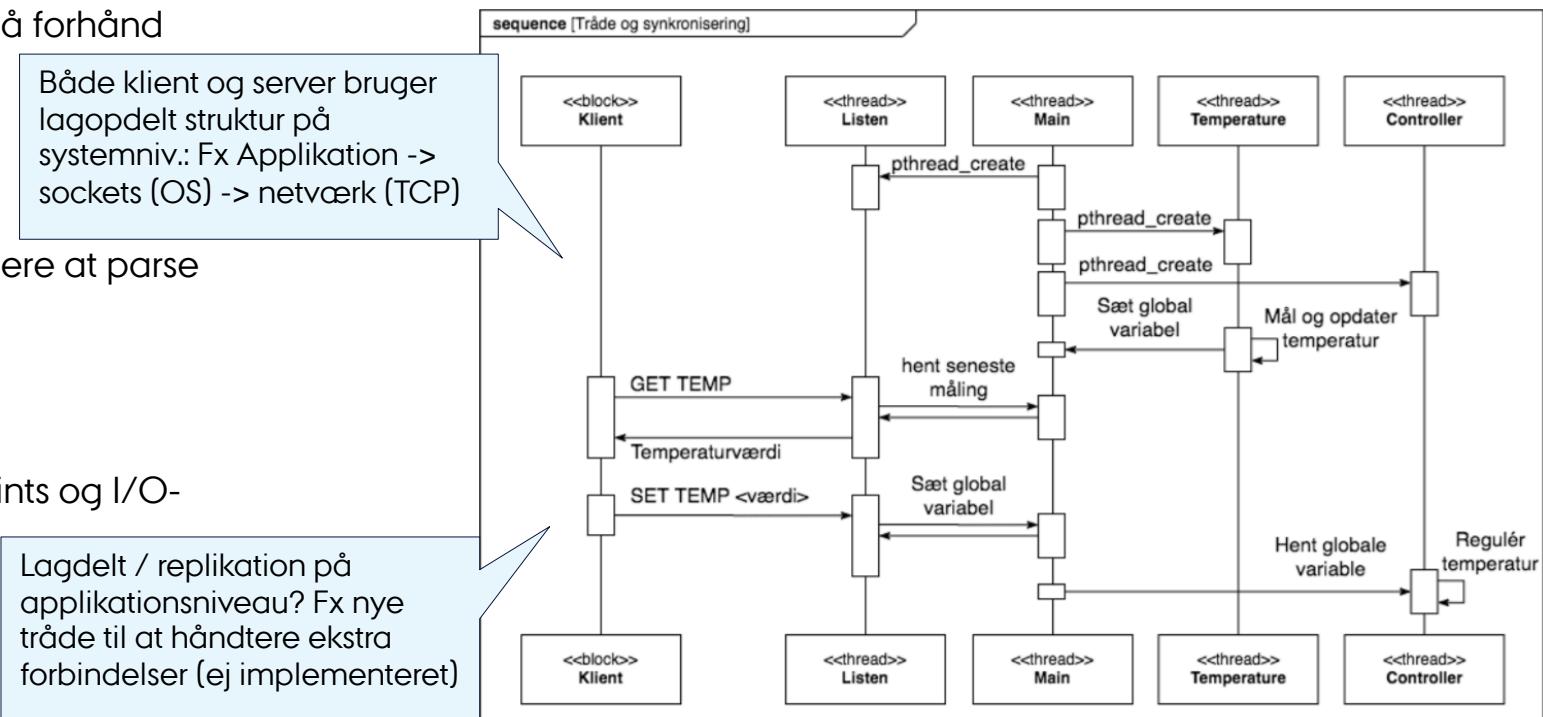


Formål med JSON-RPC: Struktureret kommunikation, lettere at parse syntax / grammatik for kommandoer og svar

Formål med brug af sockets og tråde i arkitekturen:

- Sockets: Kommunikere over netværk (TCP)
- Tråde: Opdele arbejde med forskellige tidsconstraints og I/O-typer.

Lagdelt / replikation på applikationsniveau? Fx nye tråde til at håndtere ekstra forbindelser (ej implementeret)



EKSEMPEL 2: FRA PRO4

WEBINTERFACE FOR FYSISKE PRØVESTANDE

Containeriseret/
Docker-drevet
system

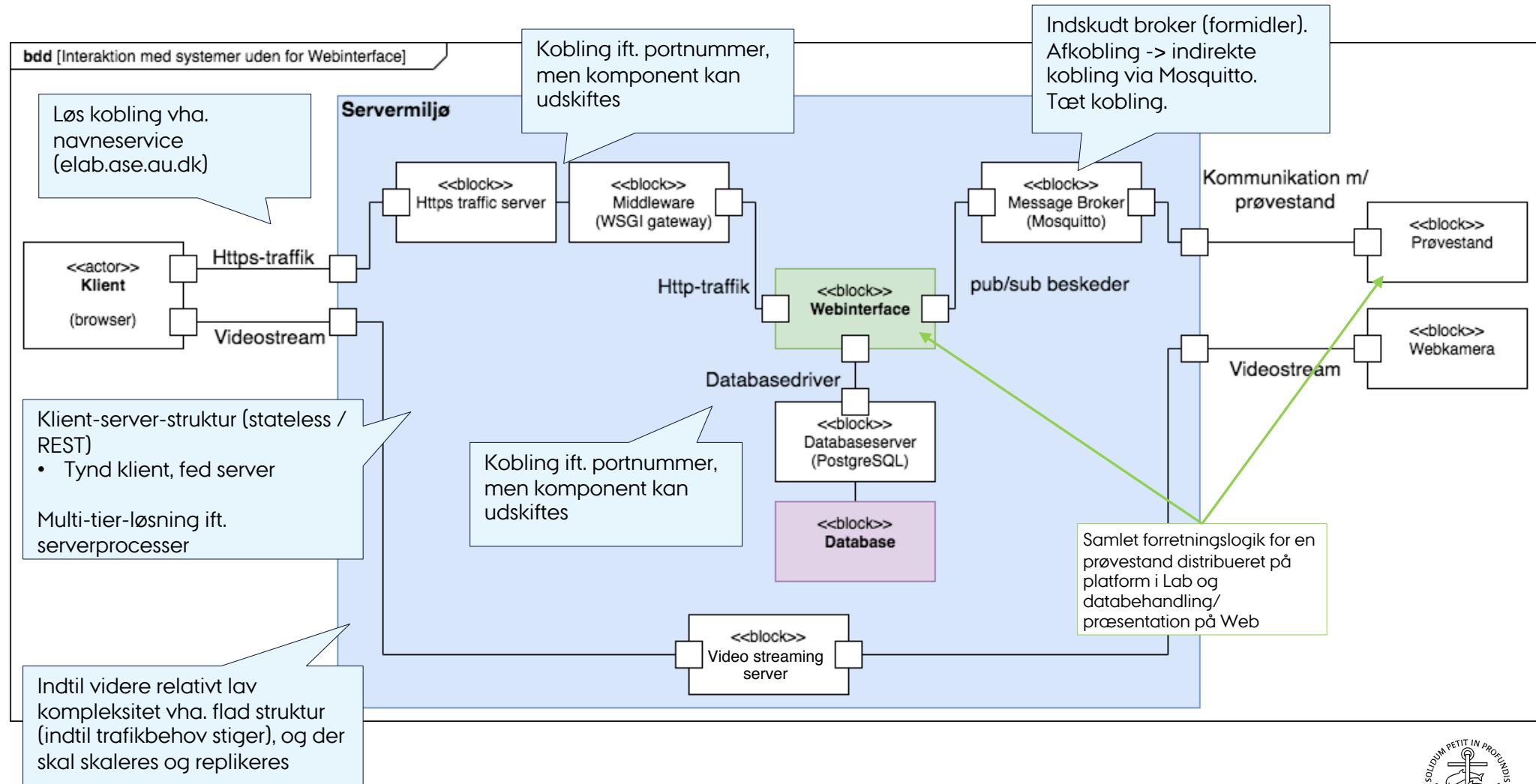
- Mikroservices

Systemet kunne
skaleres op,

Replikation af
services

Mest vanskelige at
skalere er
databasen

- Replikations-
metoder

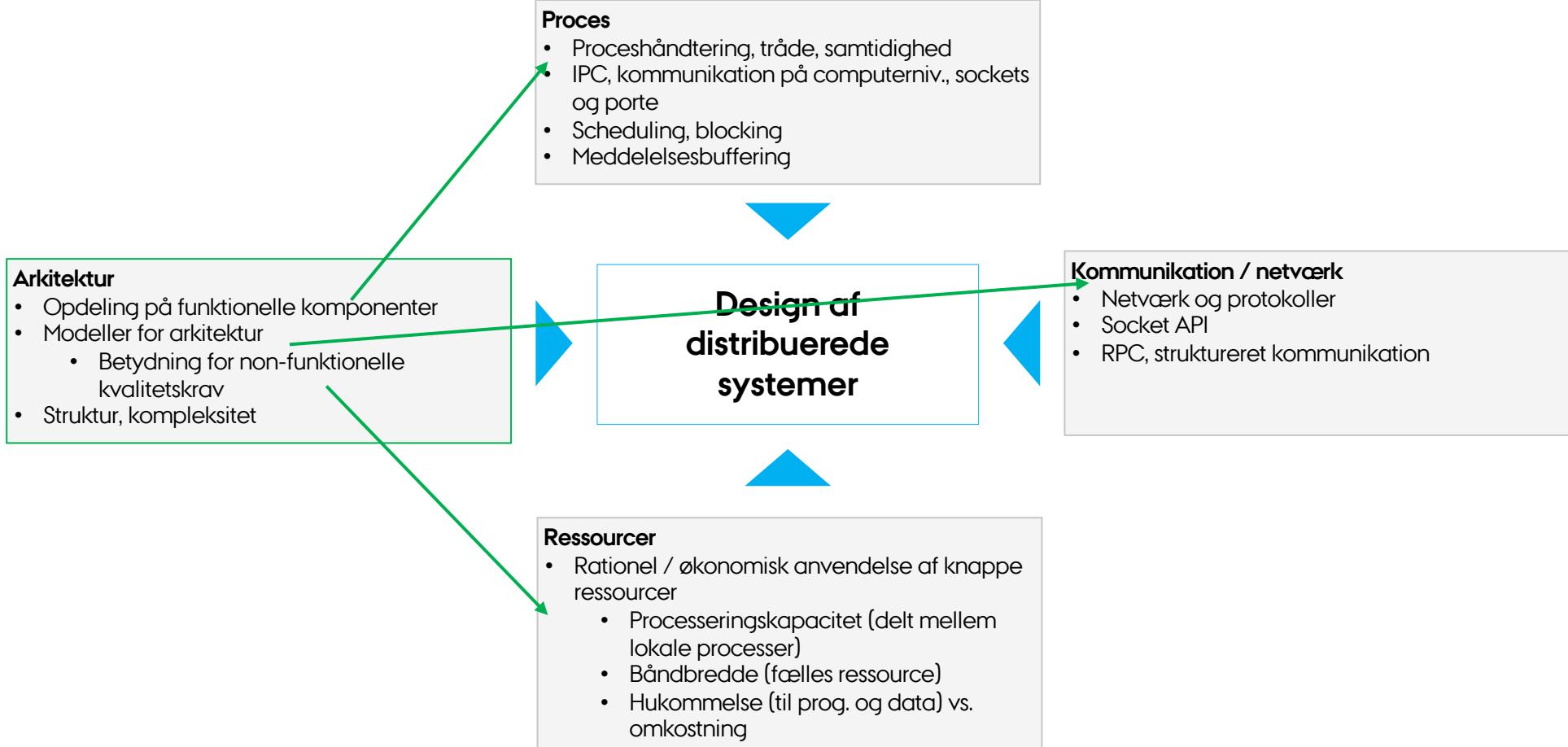


DE FIRE PERSPEKTIVER (VIEWS) PERSPEKTIVERING

Forskellige perspektiver på hvordan *krav* og *designmål* kan opnås

Forskellige indgange til fornuftigt anvendelse af *teknologi* i design af distr. systemer.

Gennemgående: Teknikker til at opnå gennemsigtighed/gennemsuelighed (transparency).



OPSUMMERING OG KONKLUSION



DEFINITIONER

Logisk arkitektur: komponent <-> komponent

Fysisk arkitektur: Binding til ressourcer, kommunikationslag

Distribueret system:

- Den samlede funktionalitet (programlogik) er distribueret ud på flere netværksforbundne enheder
- Ressourcer benyttet af systemet er delt ud over flere computere, der kommunikerer over netværk

Distribueret applikation:

- Programlogik er delt ud over to eller flere softwarekomponenter, som evt. eksekverer på forskellige computere

Netværksapplikation:

- Benytter netværk til kommunikation

HW1: CAN-BUS

1. Overblik
2. Bus-signaler og interfaces
3. Afsendelse og indhold i en dataframe
4. CAN og OSI-modellen + SocketCAN
5. Eksempel: Mini CAN-bus system
6. Konklusion

1: CAN bus

Redegør for CAN bus kommunikation, med fokus på to nederste lag i OSI modellen.

Du kan f.eks. vælge at fokusere på nogle af følgende emner :

- 1) Opbygning af CAN telegrammet
- 2) Arbitrering og ID prioriteter
- 3) Bit stuffing
- 4) Domiant/Recessive bus-states, Acknowledge
- 5) CAN controlleren på pocketBeagle's OSD3358 processor med fokus på CAN-utils, og Socket CAN

CAN OVERBLIK

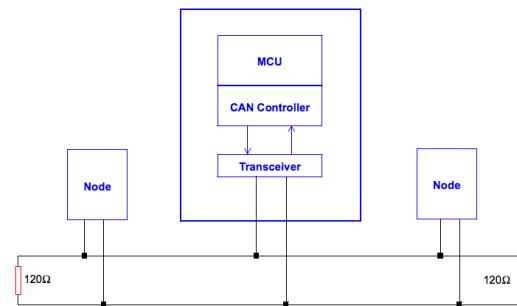
Punkt

Hvad er det?

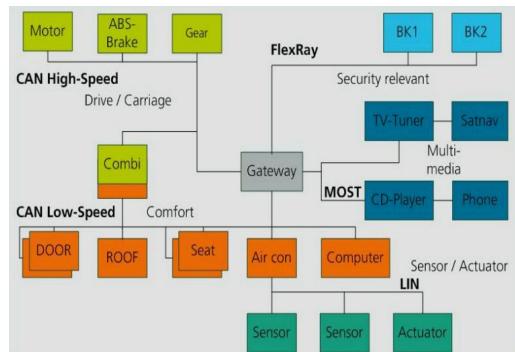
Detaljer

- CAN er en ISO-standard for arkitektur og protokol til robust, semi-hurtig, prioriteret og multicast/besked-baseret kommunikation mellem flere noder på en multi-master bus.
 - Robust = spec.'et til at fungere i miljø med støj:
 - To-ledning-bus med differentieret signal giver højere støjimmunitet end single-ended.
 - Indbygget error detection vha. CRC.
 - Semi-hurtig:
 - Hastighed varier med transmissionsafstand, op til 1 Mbit/s under 25 m [1, s. 14].
 - Prioriteret:
 - Anvendelse af arbitrering baseret på node ID implementerer, at laveste ID altid får forrang i kommunikation. Men en besked allerede under afsendelse kan ikke pre-emptes [1, s. 20-21].
 - Garanterer maksimal latency (hård realtid - men nok kun for laveste ID ☺) [1, s. 7].
 - Multicast/besked-baseret kommunikation:
 - Alle forbundne enheder lytter med.
 - Data sendes ud på bussen i frames, hver frame indeholder op til 8 bytes data.
 - Flere noder (se fig. 1):
 - Microcontrollers eller andre enheder (sensorer, aktuatorer) tilkobles bussen.
 - Multi-master bus (se fig. 1):
 - Bus uden en host, så flere noder kan tage ansvar for at være "masters", dvs. initiere beskeder ud på bussen.
- Oprindeligt til automotive, hvor mange sub-systemer er distribuerede dele af et samlet system i en begrænset afstand til hinanden (se fig. 2). En enkelt bus minimerer lange, komplicerede ledningstræk i bilen, giver single-point adgang til alle tilkoblede enheder, og med en bus af twisted-pair gives endnu bedre støjimmunitet.
- Anvendeligt i mange lignende situationer, hvor mechatroniske / embeddede, distribuerede enheder inden for en kort afstand skal kommunikere, evt. i et støjfyldt miljø (se fig. 3)
- Andre områder er fx robotsystemer med mange microcontrollers, sensorer og aktuatorer til hver robot-del. Military og aviation: Oplagt til robust kommunikation i fx fly, våbensystemer eller lign.

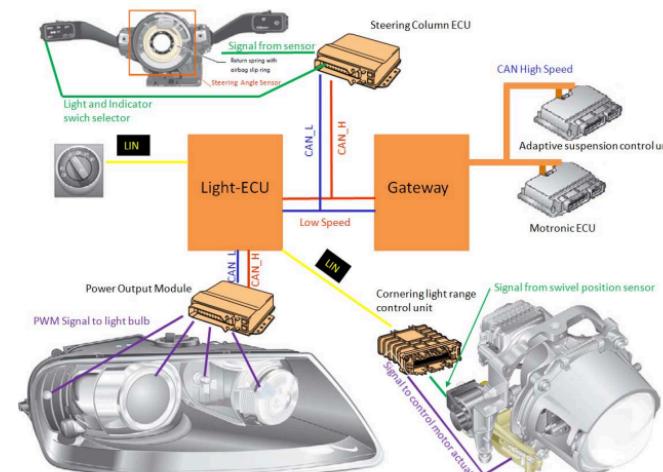
Anvendelser af
CAN i komplekse
distribuerede
systemer



Figur 1. Topologi til et simpelt system med CAN-bus [2]



Figur 2. Topologi til et CAN-bus-system i en bil [3]



Figur 3. Eksempel på anvendelse. Styrevinkel på rattet og lysindstillinger fra instrumentbræt er via CAN-bus forbundet til en ECU, som igen via CAN forbinder til sub-systemet til styring af forlygternes lysstyrke og vinkel [3]

[1]: Marco Di Natale. Understanding and using the Controller Area Network. 2008.

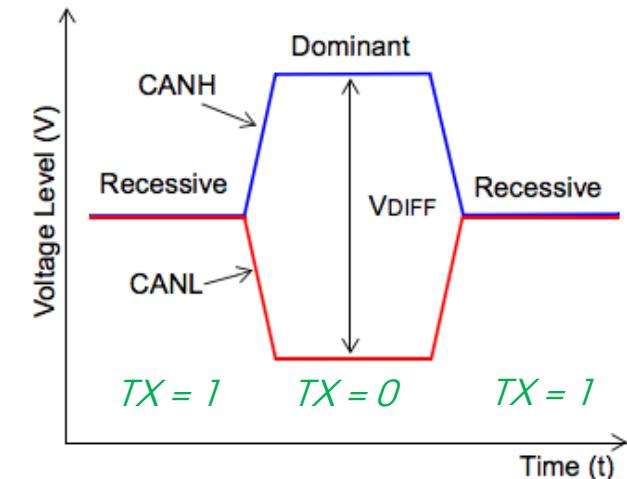
[2]: Microchip. A CAN Physical Layer Discussion. 2002. URL: <http://ww1.microchip.com/downloads/en/appnotes/00228a.pdf>

[3]: R Steinhilper, S. Freiberger & A. R. Nagel. Understanding the communication between automotive mechatronics and electronics for remanufacturing purposes. 2012.

BUS-SIGNALER OG INTERFACES

Differentielt CAN-bus signal

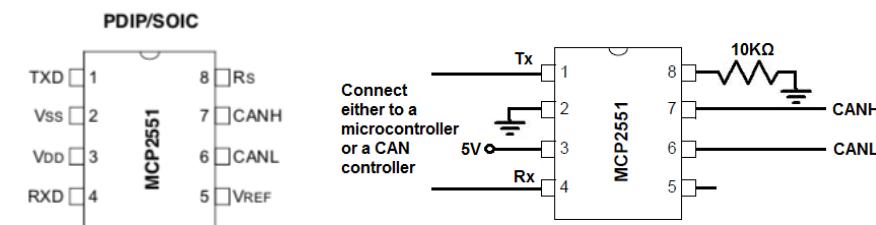
- TX = 1 (logisk høj) implementeres med "recessive" bit-tilstand på bussen (se fig. 1).
- TX = 0 (logisk lav) implementeres med "dominant" bit-tilstand.
 - -> Dominante bits på bussen vil altid "vinde" over recessive.
 - -> Arbitrering: laveste ID får altid forrang.



Figur 1. Bit-tilstande på bussen [2]

Interfaces fra controller til bus

- Transceiver er bindeleddet mellem controller og bus (se fig. 2), og driver spændingsniveauerne på bussen.
- Single-ended TX/RX fra/til controller interfacere med transceiver.
- Transceiver interfacere med bus vha. CANH og CANL.



Figur 2. Interfacing via transceiver (MCP2551) [4] [5]

[2]: Microchip. A CAN Physical Layer Discussion. 2002. URL: <http://ww1.microchip.com/downloads/en/appnotes/00228a.pdf>

[4]: Microchip. High-Speed CAN Transceiver. 2007. URL: <http://ww1.microchip.com/downloads/en/devicedoc/21667e.pdf>

[5]: Learning about electronics. How to Build a CAN Transceiver Circuit with an MCP2551 Chip. URL: <http://www.learningaboutelectronics.com/Articles/MCP2551-CAN-transceiver-circuit.php>

AFSENDELSE AF BESKEDER OG INDHOLD I EN DATAFRAME

CAN er asynkront, så hver node skal holde styr på timing / frekvens (op til 1 Mbit/s).

Hver node afsender beskeder uafhængigt af andre noders tilstand (besked-baseret protokol):

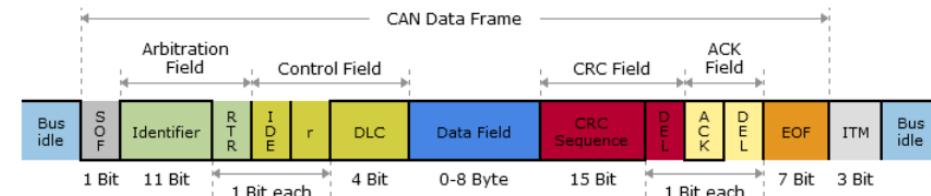
- Flere noder sender samtidig -> arbitrering vha. ID (laveste ID er dominant), men beskeder under afsendelse kan ikke pre-emptes.
- Beskedtyper kan være [1, s. 17]:
 - Data Frames -> Afsende data.
 - Remote Frames (RTR) -> Forespørge data.
 - Error Frames og Overload Frames (flag) -> Rapporterer detekterede fejl/overloading.

En CAN dataframe indeholder (se fig. 1):

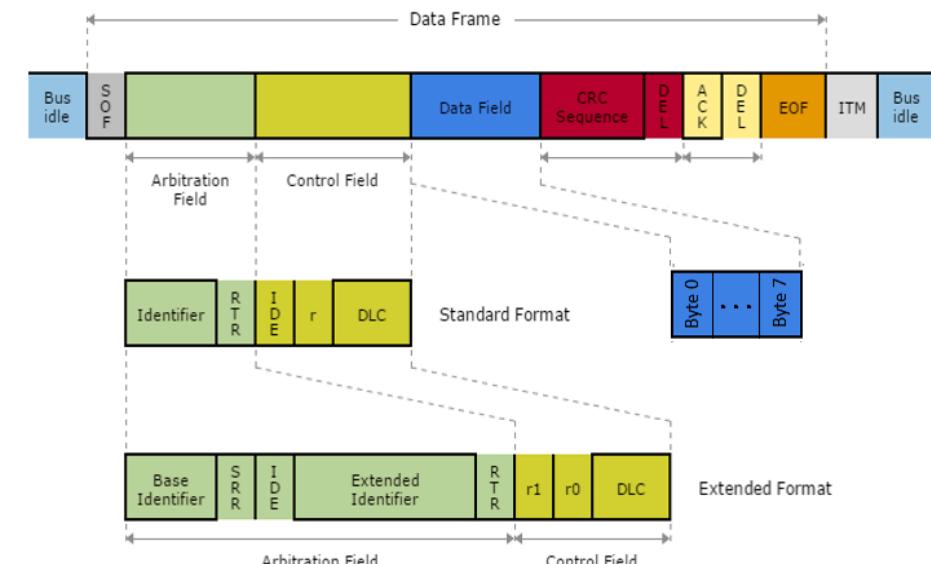
- **SOF:** Start of frame, 1 dominant bit.
- **Identifier:** Unikt node ID på CAN-bussen.
- **RTR:** Kun til Remote Transmission Request (Remote Frames).
- **IDE, r:** Recessiv markerer ext. ID (29-bits)ift. std. 11-bits ID. "r" er reserveret til kombination med IDE.
 - Se fig. 1b for sammenligning af formater med std. ID og extended ID.
- **DLC:** Længde på dataindhold (0-8).
- **Data:** Op til 8 bytes data.
- **CRC, DEL:** Cyclic redundancy check til fejldetektion + delimiter.
- **ACK, DEL:** Til acknowledgement af modtaget besked (modt. på bussen sender dominant) + delim.
- **EOF:** End of frame: 7 recessive bits.

Øvrigt:

- Bit-stuffing: Sekvens af 5 bits med samme polaritet -> stuffing 1 bit med modsat polaritet. Dataframes får variabel længde (antal bits, der "stuffles" ind). Bruges til timing.
- En dataframe efterfølges af Interframe spacing (ITM) på bussen: 3 recessive bits.
- En dataframe kan fx implementeres som i RTOS journal (se fig. 2).



Figur 1a. CAN datatransmission og CAN dataframe [3]



Figur 1b. Standard og extended (CAN 2.0A)

```
25 /* The CAN datagram type */
26 typedef struct {
27     unsigned int ID;           // holds 11/29 bit ID
28     unsigned char ID_ext;      // set to 1 if ID is extended -29 bit, else 0
29     unsigned char DLC;         // how many bytes of payload
30     unsigned char data[8];     // the data payload
31 } can_tlg_t;
```

Figur 2. CAN datastruktur (fra egen RTOS-journal)

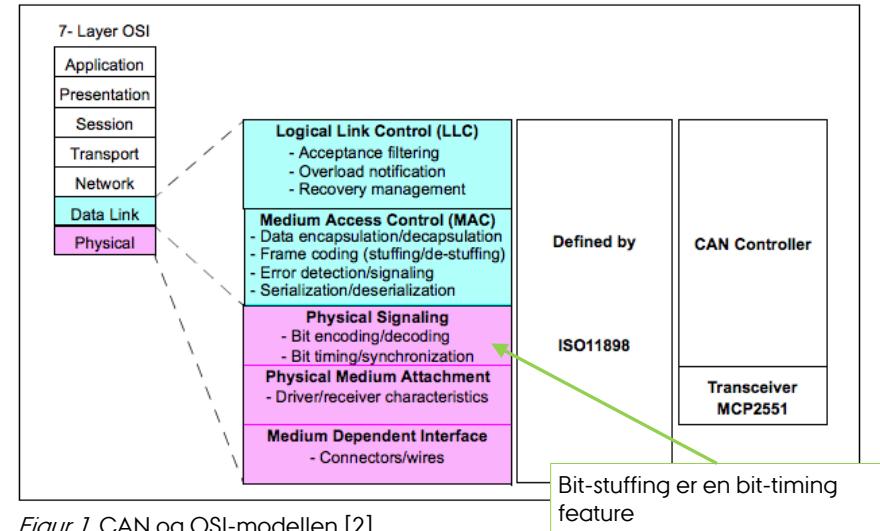
CAN OG OSI-MODELLEN RÅ CAN OG SOCKETCAN

CAN implementerer de to nederste lag i OSI-modellen (se fig. 1 til højre)

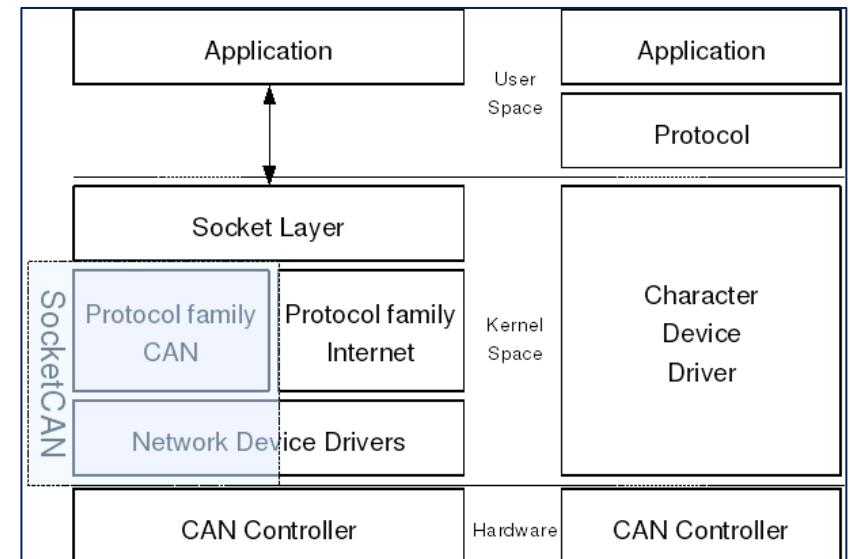
- CAN-controller (på PocketBeagle) implementerer det meste af data link layer.
- CAN-transceiver (PHY'en, MCP2551) implementerer forbindelse til det fysiske medium. Oversætter single-ended TX/RX til differentiel CAN_H, CAN_L.
- Bus-ledninger og terminering skal implementeres separat som defineret i standarden.

SocketCAN (se fig. 2) implementerer yderligere transport og netværkslag:

- CAN kan så benyttes som et hvert andet netværksinterface i Linux/Unix.
- Et CAN-device bliver en byte-strøm, som flere programmer kan læse fra / skrive til samtidig.



Figur 1. CAN og OSI-modellen [2]



Figur 2. SocketCAN-modellen [6]

[2]: Microchip. A CAN Physical Layer Discussion. 2002. Tilgængelig online: <http://www.microchip.com/downloads/en/appnotes/00228a.pdf>

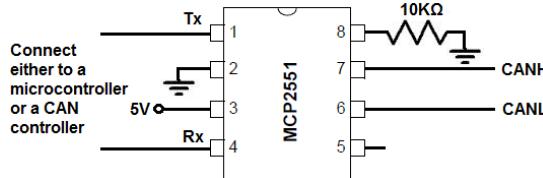
[6]: Wikipedia.org. SocketCAN. 2020. Tilgængelig online: <https://en.wikipedia.org/wiki/SocketCAN>

CAN-BUS-SYSTEM MED TO NODES VIA POCKETBEAGLE CONTROLLERS

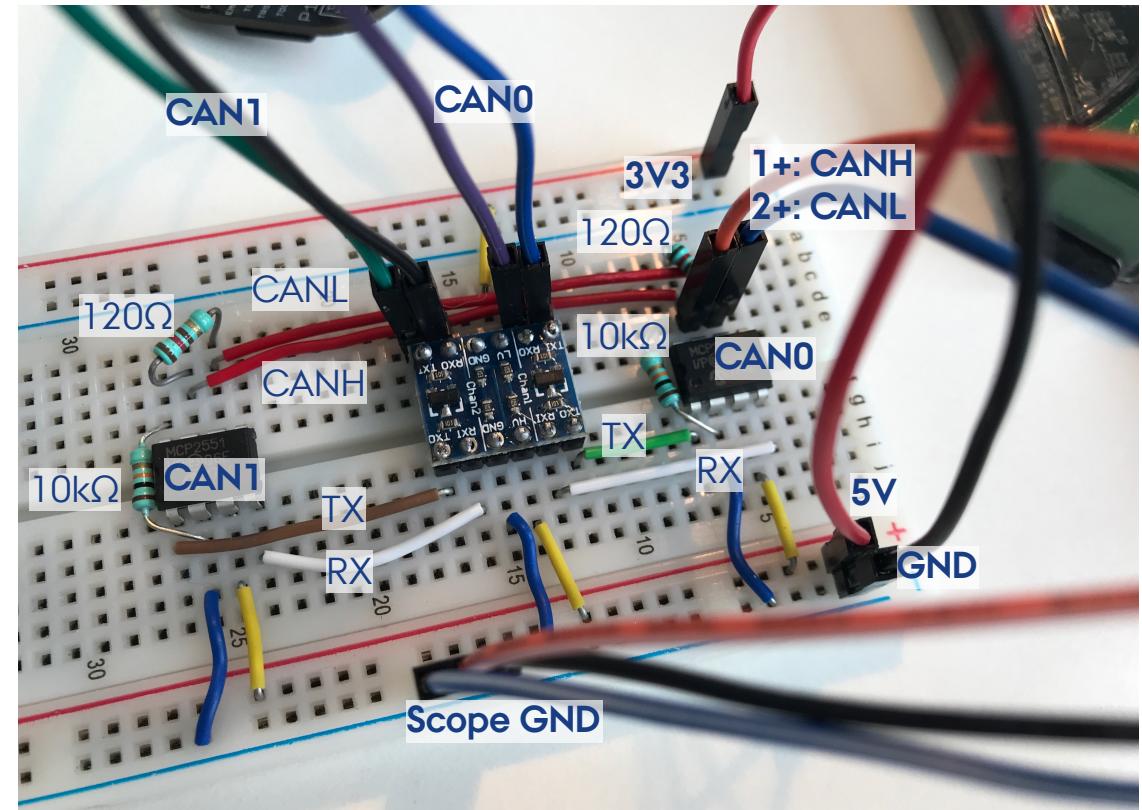
Systemet er sat op som på fig. 1

- Konvertering mellem PocketBeagles 3,3V logik TX/RX og transceiverens 5V-logik sker med en logic level converter (se midten af fig. 1).
- Bussen er termineret i hver ende med $120\ \Omega$.
- Slew Rate (SR) er sat med $10\ k\Omega$, hvilket giver ca. $24\ V/\mu s$, hvilket er relativt hurtige transitioner [4, s. 4].
- Se appendiks for billede inkl. PocketBeagle.

Forbindelser til MCP2551 er som vist i fig. 2



Figur 2. Interfacing via transceiver (MCP2551) [5]



Figur 7. Implementering af simpelt CAN-baseret system

PINMUX OG SOCKETCAN

CAN0 controller:

- TX: P1.26
- RX: P1.28



Figur 1. Pins til hhv. CAN0 og CAN1

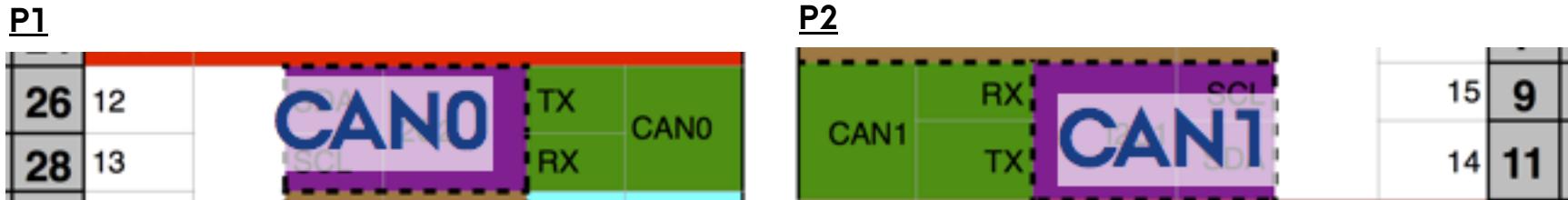
CAN1 controller:

- TX: P2.11
- RX: P2.9

Pinmux: config-pin (fig. 2)

Netværk vha. SocketCAN (can-utils)
(fig. 3)

Konklusion: Setup virker (fig. 4)



Figur 1. Pins til hhv. CAN0 og CAN1

```
setup-can.sh
1  # Config CAN0
  1 #RX
  2 config-pin -a P1.28 can
  3 #TX
  4 config-pin -a P1.26 can
  5
  6 # Config CAN1
  7 #RX
  8 config-pin -a P2.9 can
  9 #TX
 10 config-pin -a P2.11 can
```

Figur 2. Script til pin-config af CAN0 og CAN1

```
start-can.sh
1  # Run this script with sudo
  1 # Starts CAN0 and CAN1 with 50kbit/s
  2 ip link set can0 type can bitrate 50000
  3 ip link set can1 type can bitrate 50000
  4 ip link set can0 up
  5 ip link set can1 up
```

Figur 3. Script til start af CAN med 50kbit/s (skal køres med sudo)

Afsend:

```
janus@beaglebone:~/projects/E4ISD2/CAN$ cansend can0 014#11.22.33.
44.55.66.77.77
janus@beaglebone:~/projects/E4ISD2/CAN$ cansend can0 014#11.22.33.
44.55.66.77.77
```

Figur 4. Test cansend og candump

Modtag:

```
janus@beaglebone:~/projects/E4ISD2/CAN$ candump can1
can1 014 [8] 11 22 33 44 55 66 77 77
can1 014 [8] 11 22 33 44 55 66 77 77
can1 014 [8] 11 22 33 44 55 66 77 77
```

TEST

Opsætning:

- Sender på CAN0
- Lytter på CAN1
- 50 kbit/s -> **1 bit = 20 us**

Testbesked til afkodning:

- ID: 0x014
 - 0b00000010100 (11-bits)
 - Bemærk ID resulterer i bit-stuffing
- Payload: 0x01

Oscilloskop:

- VDIFF=CANH-CANL
- VDIFF høj \Leftrightarrow dominant
- VDIFF lav \Leftrightarrow Recessiv

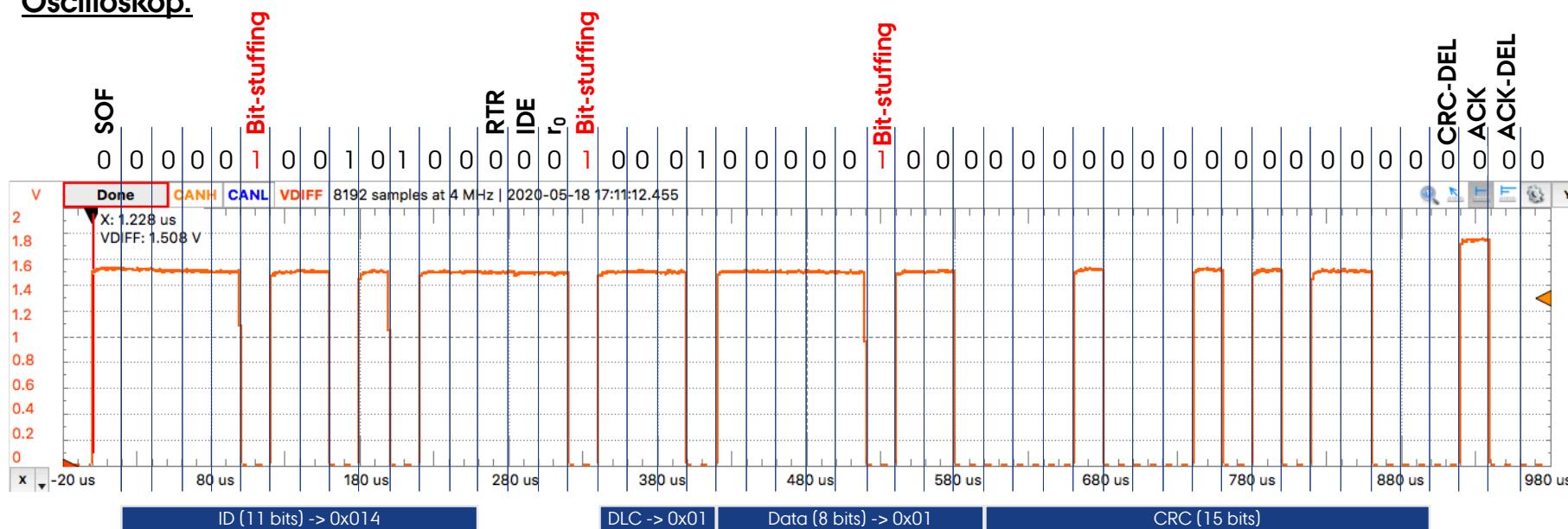
Afsend:

```
janus@beaglebone:~/projects/E4ISD2/CAN\$ cansend can0 014#01
```

Modtag:

```
can1 014 [1] 01
```

Oscilloskop:



Konklusion: Beskeden kan afkodes korrekt!

KONKLUSION (1)

CAN har mange fordele til de tiltænkte anvendelser: Robusthed/støjimmunitet, fornuftig hastighed, prioriteret kommunikation, alt sammen i en relativt simpel/enkel protokol.

- Godt til distribuerede, embeddede systemer, med wired enheder inden for kort afstand.

Ulemperne er dog bl.a.:

- Begrænsede dataframes:
 - Maks. 8 bytes per frame og maks. 1 Mbit/s over bussen kan være *for lidt* til avancerede systemer. Fx til benyttelse af videofeed til AI/ML-baseret navigation/objektgenkendelse (tænk selvkørende biler, droner, robotter og lignende).
- Multicast og faste ID'er (fixed priority) – mangler "scheduling" på bussen:
 - Højere ID'er bliver altid "crowded out" af lavere ID'er,
 - Ingen congestion avoidance,
 - => I principippet kan der ske noget tilsvarende "priority inversion" i realtids-systemer:
 - Lav-ID node venter på data fra en højere ID-node, der bliver crowded out.
 - Ingen mulighed for "priority inheritance".
 - Ingen indbygget "connection eller session" mellem devices (skal implementeres i et højere OSI-lag)
 - => Anvendelse af bussen er ikke efficient til point-to-point komm. (bruger meget tid på arbitration, ID'er, ACK, osv.).

KONKLUSION (2)

I denne øvelse er:

- CANs karakteristika og anvendelser kort opsummeret.
- Et simpelt system med CAN-bus og to noder implementeret vha. to MCP2551 transceivers og de to CAN-controllers på PocketBeagle.
- Vist hvorledes can-utils i Linux kan benyttes til at styre CAN-controllers og foretage kommunikation ud på CAN-bussen.
- Demonstreret hvordan CAN-kommunikation kan debugges vha. oscilloskop.

Muligt fremtidigt arbejde:

- Benytte socketCAN i et meningsfyldt C/C++ program til kommunikation mellem distribuerede enheder på en CAN-bus.

HW2: FREERTOS

1. Overblik over FreeRTOS
2. Eksempel på benyttelse af FreeRTOS til at kommunikere mellem to tasks
 1. Oprettelse af tasks
 2. Brug af kø og synkronisering
3. Konklusion

2: FreeRTOS

Redegør for hvad FreeRTOS operativsystemet består af, og hvordan du ville anvende et RTOS.

Du kan f.eks. vælge at fokusere på nogle af følgende emner:

- 1) Hvad er en scheduler, multitasking og blocking functions
- 2) Hvilke tilstande kan et task have, og hvordan skiftes fra én tilstand til en anden
- 3) Tasks: prioriteter og starvation, priority inversion
- 4) Queue, Mutex, Semaphore
- 5) Interrupts, deferred interrupt processing

FREERTOS OVERBLIK

Punkt

Hvad er det?

Detaljer

- Letvægts-realtidsoperativsystem til embeddede devices, giver mulighed for at designe til både hård realtid¹ og til et multi-tasking-paradigme² (se fig. 1).
- FreeRTOS består af en mikrokerne:
 - Kernen giver basale abstraktioner som 'tasks', 'states' (se fig. 2) og 'priorities'.
 - Kernen laver 'scheduling' af tasks, som er klar til at køre (se fig. 3). Dette gøres ud fra en scheduling policy og task priorities. Som standard benyttes pre-emptive scheduling³.
 - Kernen stiller funktioner/mekanismer til afkobling af / kommunikation mellem / synkronisering af processer; navnlig 'queues' og 'semaphores', samt 'mutexes' til beskyttelse af delte ressourcer.
 - Kernen implementerer forskellige metoder til dynamisk memoryhåndtering.
 - Kernen indeholder ikke netværks-stack, drivers eller andet high-level.

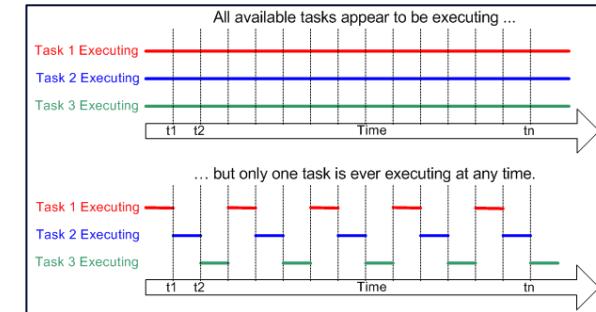
Anvendelser af et
realtidssystem

Oplagt at designe til hård realtid¹, hvor:

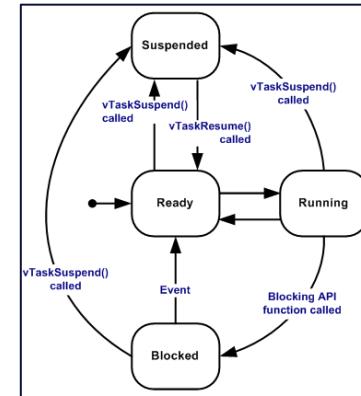
- Sikkerhedsmæssige, regulatoriske eller UX-hensyn stiller krav til hurtig reaktion på events, inden for fastlagte tidsgrænser.
 - Automotive, fx ABS, airbags, GPS,
 - Medicinsk udstyr, fx pacemakers, kardiovaskulær overvågning, guidet defibrillator,
 - Kontrolsystemer i industrien, fx fysiske/kemiske/biologiske procesanlæg,
 - Våbensystemer, fx electronic warfare, osv.

Oplagt at bruge multi-tasking-paradigmet² til systemer med:

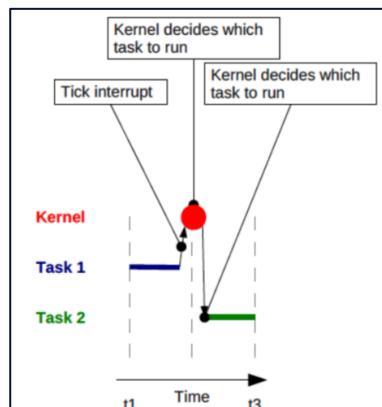
- Behov for flere processer / opdeling af programarkitektur i flere samtidige funktioner.
- Behov for høj grad af responsiveness, *samtidig* med der udføres forskellige andre opgaver, såsom:
 - sensoraflestning og beregning, seriel-kommunikation, opdatering af displays, osv.
- Beregningstunge men ikke tids-kritiske opgaver, som med fordel kan flyttes i baggrunden.



Figur 1. Multitasking giver oplevelsen af at alle tasks kører samtidig, mens dog kun én afvikles ad gangen.



Figur 2. En task kan have følgende states og transitioner.



Figur 3. Ved hvert tick vælger scheduler en task med "Ready" og sætte denne til at afvikle.

1. Garanterede deadlines, dvs. garanteret reaktionstid eller frekvens, fx i ms eller antal ticks. Kan være 'strict timing', 'flexible timing' eller 'deadline-only timing'.

2. Flere samtidige processer med hver sin stack frem for procedural/sekventiel afvikling af en proces først efter en anden er kørt til slut.

3. Typen *pre-emptive* scheduling betyder at scheduler fordeler processortid mellem tasks på baggrund af bl.a. de definerede task-prioriteter. Hvis time-slicing er slået til, skiftes også mellem tasks med lige prioritet. Scheduler swapper tasks 'in' og 'out', og sørger for at kontekst til en task gemmes/genoprettes efter state er skiftet (se fig 2.). *Pre-emptive* er i modsætning til *cooperative*, som kræver at en task giver processoren tilbage (=yield'er) og til *time-sharing*, hvor tiden fordeles ligeligt mellem alle tasks.

QUEUES OG TASK-SYNKRONISERING

ØVELSE 2 UGE 7

Formål:

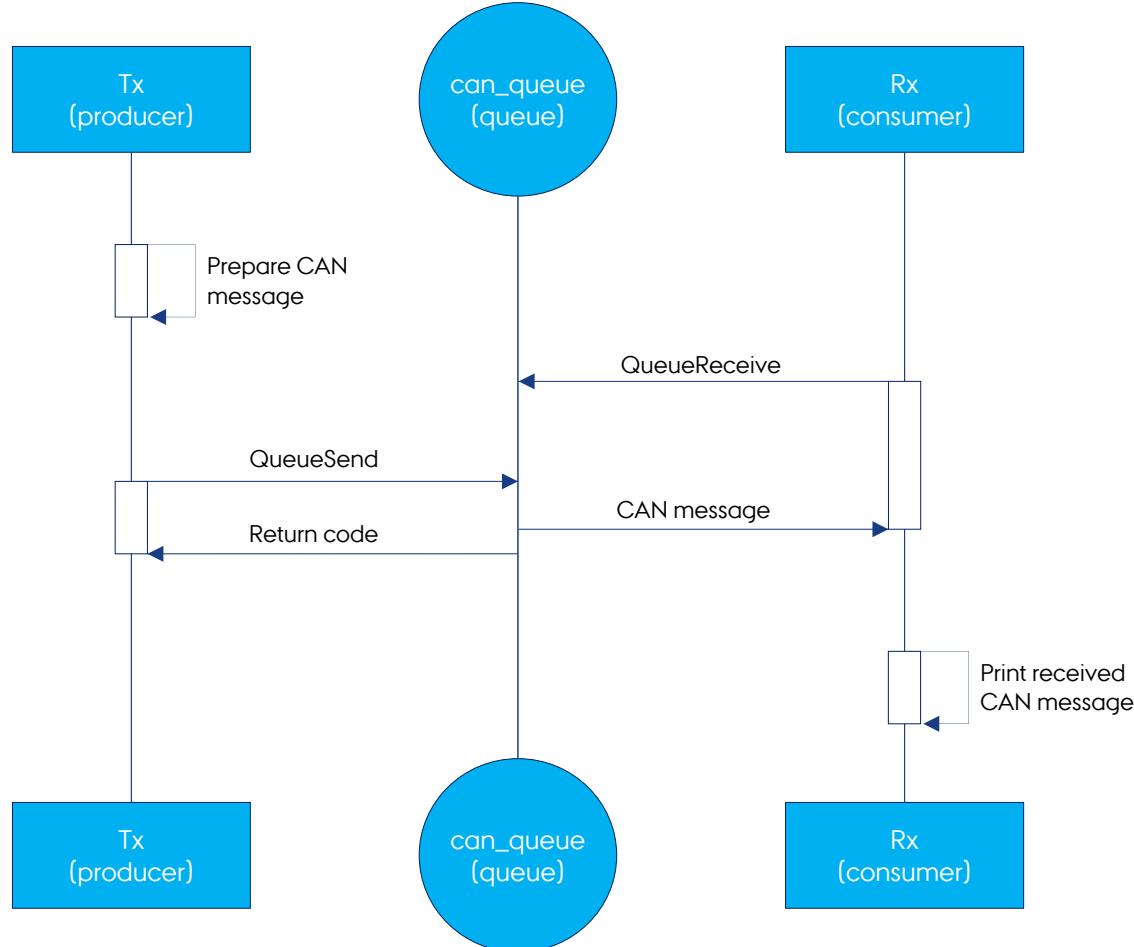
- Implementering af en kø mellem to tasks:
 - Udveksle CAN-beskeder
 - Task-synkronisering ved udveksling af beskeder

Opsætning:

- Ny datatype (CAN datagram)
- Kø med ny datatype
- Tx og Rx tasks i producer-consumer pattern

Metode (se figur):

- De to tasks kører asyntont af hinanden, men synkroniseres gennem queue
- Bemærk i illustration, at Rx kan forsøge at hente fra køen før der er data til rådighed
 - I så fald blokerer kaldet indtil der er data (eller indtil timeout er gået, hvad end sker først).
- Tx ville kunne blokere (el. gå timeout), hvis køen var fuld.



CAN DATATYPE OG HANDLES

I denne kodelump erklæres grundelementerne til programmet

```
2④ /**
3 * @file E4ISD2_wk07_FreeRTOS_Queue_rx_tx.c
4 * @author Janus Bo Andersen (JA67494)
5 * @date March 2020 (E4ISD2 spring 2020)
6 * @brief Main function and tasks to implement tx-rx system
7 * that sends CAN datagrams via a queue in FreeRTOS.
8 * @note Inspiration from https://www.youtube.com/watch?v=yHfD0\_jiIFw
9 */
10
11 #include <stdio.h>
12 #include <stdlib.h>
13 #include "board.h"
14 #include "peripherals.h"
15 #include "pin_mux.h"
16 #include "clock_config.h"
17 #include "MKL25Z4.h"
18 #include "fsl_debug_console.h"
19 #include "FreeRTOS.h"
20 #include "task.h"
21 #include "queue.h"
22
23 #define QUEUE_LEN 5 // Queue depth
24
25 /* The CAN datagram type */
26④ typedef struct {
27     unsigned int ID; // holds 11/29 bit ID
28     unsigned char ID_ext; // set to 1 if ID is extended -29 bit, else 0
29     unsigned char DLC; // how many bytes of payload
30     unsigned char data[8]; // the data payload
31 } can_tlg_t;
32
33 /* Create handle for the two tasks */
34 TaskHandle_t tx_hdl = NULL;
35 TaskHandle_t rx_hdl = NULL;
36
37 /* Create handle for the queue */
38 QueueHandle_t can_queue = NULL;
```

Headerfilerne FreeRTOS.h og task.h er nødvendige for at lave et RTOS-projekt.

Headerfilen queue.h er nødvendig for at bruge kø-datastrukturen og tilhørende metoder.

Definerer, at der skal kunne holdes 5 elementer i køen. Det tal er helt arbitraet i dette eksempel.

Definerer datastrukturen til CAN-beskeder.

Handles til producer og consumer tasks.

Handle til køen erklæres som global variabel, så begge tasks kan få fat i den.

PRODUCER TASK: TX

I denne kodelstump defineres **producer task** – dvs. funktionen, som opretter og afsender CAN-beskeder gennem køen.

```
41/* @brief This task transmits random CAN datagrams
42 * @param None
43 */
44void tx(void * p) {
45
46    can_tlg_t message; // datagram
47
48    BaseType_t rc; // gets return code from xQueueSend
49
50    uint8_t r = 0; // gets the random values
51    srand(0); // seed to always start same place
52
53    while(1) {
54
55        /* fill the control fields */
56        message.ID = 14;
57        message.ID_ext = 0;
58        message.DLC = 7; // 7 bytes of data to be sent
59
60        /* fill the data fields */
61        for (int i = 0; i < message.DLC; i++) {
62            r = rand() % UINT8_MAX; // draw random num
63            message.data[i] = r; // put in buffer
64        }
65
66        /* copy CAN telegram to the queue */
67        /* wait for 500 ticks before timeout */
68        printf("\nSent telegram with ID %d to receiver_task. ", message.ID );
69        rc = xQueueSend(can_queue, &message, 500);
70        puts( rc ? "Sent.\n" : "Failed.\n" );
71
72        vTaskDelay(200);
73    }
74
75    // Task must not return!
76    vTaskDelete(NULL);
77 }
```

Udfylder CAN-beskedens kontrolfelter, som spec'et i opgaven

Udfylder CAN-beskedens datafelter med værdier 0-255.

Sætter (sender) CAN-beskeden ind i køen (bagenden).

Tjekker om det gik godt (ingen fejl eller timeout).

Vent 200 ticks (0.2 sek @ 1kHz tick rate) før næste CAN-besked begyndes

CONSUMER TASK: RX

I denne kodelump defineres **consumer task** – dvs. funktionen, som opretter og modtager CAN-beskeder gennem køen og udskriver dem til konsollen.

```
79/* @brief This task receives CAN datagrams and prints them
80 * @param None
81 */
82void rx(void * p) {
83    can_tlg_t message;      // incoming message
84
85    while(1) {
86        /* Receive from the queue, wait max 500 ticks */
87        if ( xQueueReceive(can_queue, &message, 500) ) {
88            // received a message
89            printf("Received CAN telegram with %d bytes of data:\n", message.DLC);
90            printf("ID: %d\n", message.ID);
91
92            for (int i = 0; i < message.DLC; i++) {
93                printf("Data [%d]: %d\n", i+1, message.data[i]);
94            }
95
96        } else {
97            // failed to receive, or timed out
98        }
99    }
100
101    // Task must not return!
102    vTaskDelete(NULL);
103}
```

Henter data fra køen (eller blokerer), og tjekker samtidig returværdien.

Udskriver kontrolfelter som spec'et i opgaven.

Udskriver datafelter, afhængigt af datalængden i bytes (DLC).

Ingen handling defineret her, men det ville give mening at definere betydningen af, at ingen data er modtaget inden timeout...

- Er det en fejl, der skal rapporteres?
- Er det et tegn på at gå i dvale?
- Skal der bare forsøges igen?
- Osv...

MAIN: OPRETTELSE AF KØ, OPRETTELSE AF TASKS OG START AF SCHEDULER

I denne koden oprettes kø-objektet (der allokeres hukommelse).

De to tasks oprettes.

Scheduler startes.

```
119      PRINTF("E4ISD2 FreeRTOS exercise 2/wk7. Transmission via queues.\n");
120
121  /* Create the queue to hold 5 CAN datagrams */
122  can_queue = xQueueCreate(QUEUE_LEN, sizeof(can_tlg_t));
123
124  if (!can_queue) {
125      printf("Error. Queue could not be created.\n");
126      return -1; /* No point continuing */
127  }
128
129
130
131  /* Create the tx and rx tasks */
132  xTaskCreate(tx,
133              "TX",
134              configMINIMAL_STACK_SIZE*2,
135              NULL,
136              tskIDLE_PRIORITY+1,
137              &tx_hdl); // function pointer to tx task
138
139  xTaskCreate(rx, "RX", configMINIMAL_STACK_SIZE*2,
140              NULL, tskIDLE_PRIORITY+1, &tx_hdl);
141
142  /* Run the scheduler */
143  vTaskStartScheduler();
144
```

Køen oprettes (allokering). Der allokeres X antal elementer, der hver har størrelse som CAN-beskeds typen.

Bekræft, at allokering er OK.

Opretter de to tasks, hhv. Tx og Rx. Bemærk især:

- Stack size er $2 * \text{minimum}$ ($2 * 90$ words). Det er et sjus, og sikkert mere end rigeligt.
- Prioritet for begge tasks er ens, og er højere end idle-task (0).

Start scheduler

OUTPUT FRA DE TO TASKS

Output fra de to tasks er som spec'et i opgaven.

Hastigheden er således, at der udskrives en ny CAN-besked ca. hvert sekund. Dette er grundet høj latency ved skrivning til stdout/terminal via debug-interfacet.

```
Sent telegram with ID 14 to receiver task. Sent.  
Received CAN telegram with 7 bytes of data:  
ID: 14  
Data [1]: 62  
Data [2]: 8  
Data [3]: 44  
Data [4]: 132  
Data [5]: 116  
Data [6]: 26  
Data [7]: 86  
  
Sent telegram with ID 14 to receiver task. Sent.  
Received CAN telegram with 7 bytes of data:  
ID: 14  
Data [1]: 230  
Data [2]: 13  
Data [3]: 173  
Data [4]: 157  
Data [5]: 231  
Data [6]: 212  
Data [7]: 210  
  
Sent telegram with ID 14 to receiver task. Sent.  
Received CAN telegram with 7 bytes of data:  
ID: 14  
Data [1]: 47  
Data [2]: 59  
Data [3]: 181  
Data [4]: 66  
Data [5]: 239  
Data [6]: 125
```

Producer-task (Tx) beretter om afsendt CAN-besked.

Den ventende consumer-task (Rx), beretter om en modtaget CAN-besked, og udskriver datafelterne.

Pga. ens prioriterer og configUSE_TIME_SLICING=0, får producer-task lov til at køre færdig før consumer-task swappes ind (ellers var denne linje brudt).

CAN-beskedens dataindhold ændres hver gang, men kontroldata forbliver uændret.

KONKLUSION

FreeRTOS kan nemt benyttes til at oprette flere tasks, afvikle dem som multi-tasking og lade dem kommunikere/synkronisere:

- Tasks er afkoblede og asynkrone, men synkroniserer eller kommunikerer vha. køer.
- Den indbyggede kø-abstraktion hænger tæt sammen med state-systemet, så en ventende task blokerer – indtil indsættelse af data i køen får den ventende task til at rykke til "Ready" state (eller indtil den går timeout).
- Tilsvarende, hvis køen er fuld, så vil den producerende task vente på ledig plads (eller gå timeout).
- Et timeout skal håndteres alt efter "semantik" og kontekst: Det skal vides, hvad timeoutet egentlig betyder.

Det anvendte designmønster (producer-consumer) er meget anvendt, og er ud over i FreeRTOS anvendeligt til:

- Afkobling af processer i flertrådet programmering.
- Afkobling af devices, fx med indskudt serielinterface, fx til co-design-projektet ☺

HW3: CO-DESIGN OG SYSTEMARKITEKTUR

1. Overblik
2. Design-modellen
3. Processer og aktiviteter i co-design
4. Design trade-offs i HW/SW mix
5. Illustration af co-design ved projekt fra forår 2020
 1. Vi må se, hvor meget vi når
6. Konklusion

3: CoDesign og system arkitektur

Redegør for hvordan man arbejder med flere platforme, der skal interagere med hinanden, tag afsæt i din CoDesign øvelse, med KL25 og Artix 7 FPGA'en. Du kan vælge at fokusere på :

- 1) Forklar begrebet CoDesign
- 2) Hvad er de væsentlige aktiviteter i CoDesign
- 3) Forklar hvordan du kunne forbedre kl25/FPGA systemets performance, f.eks vha. DMA og interrupts på input delen
- 4) Giv et eksempel på en kommunikationsprotokol, ved at forklare hvordan du sender data imellem KL25 og FPGA
- 5) Sammelign kort de tre platforme, FPGA (Artix7), Cortex M0+ CPU (KL25), og Pocket Beagle. Hvad karakteriserer de enkelte platforme, og hvilke opgaver vil de hver især egne sig til at løse?

CO-DESIGN OVERBLIK

"80 percent of a product's cost is determined during the first 20 percent of its development cycle."

www.garysmitheda.com

Punkt

Hvad er det?

Detaljer

- **Formål:** Reducere risici og omkostninger i udvikling, opnå hurtigere time-to-market
 - Opnå "optimalt" SW/HW-mix ift. krav og behov
- De fleste komplekse elektroniske produkter indeholder både HW og SW, der skal sam-udvikles
- Co-design er en metode og model for parallelt design af denne hardware og software
 - Systemtænkning og integreret design
 - Kan også benyttes til IC- og SoC-design, osv.

Problemet

- Specifikation af system bestående af HW og SW:
 - Opdele systemet på HW- og SW-komponenter
 - Mappe systemadfærd til systemstruktur.

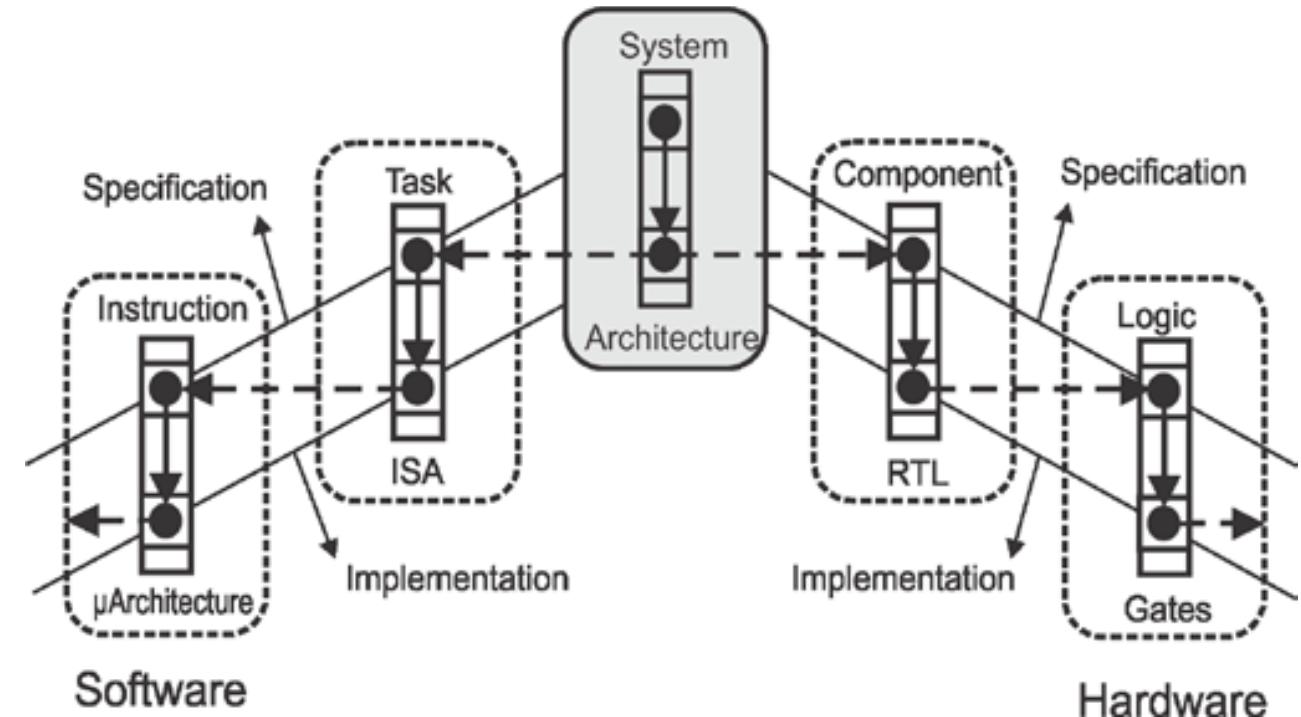
Nøglebegreber

- **Co-:** Concurrency, Coordination; Complexity, Correctness
- Co-design proces = syntese og mapping: Allocation, Binding, Scheduling
- EUDP: Architecture, Modelling og Partitioning

DESIGN-MODEL

Double-roof-modellen

- Systemet er øverste abstraktionsniv.
 - "Double roof": Fra Specifikation (behavioural) til Implementation (strukturel)
- Opdeler systemet i SW- og HW-grene:
- Udviklingskæder til syntese af løsning.
 - Bevæger sig trinvist fra højeste til lavere abstraktionsniveau
 - Implementeringsdetaljer fastlægges / forfines (vertikale pile), dvs. en forfining fra behavioural-beskrivelse til structural-design.
- Fokuseret på løsning af de tre problemer:
allokering, binding og scheduling.



Kilde: Teich (2012)

PROCES OG AKTIVITETER

Synthesis

Aktiviteter

- Oversætte fra *behavioural* specifikation til *strukturel* specifikation.
- For hvert abstraktionsniv.

Allocation

Aktiviteter

- Udvælge el. designe systemressourcer
 - Processorer, IP-blokke (interfaces), memory, osv.
 - Forbindelser; netværk, osv.
- Komponere systemarkitektur
- Allokere ressourcer.

Gentag og forfin iterativt
(test, evaluering, feedback loops, co-simulering, osv)

Binding

Aktiviteter

- Binde *behavioural/funktionelle* krav til strukturelle/arkitektur-objekter.
- Opgaver, processer og funktioner bindes til processeringsressourcer
- Variabler og datastrukturer bindes til hukommelser
- Kommunikation bindes til ruter mellem ressourcer.

Scheduling

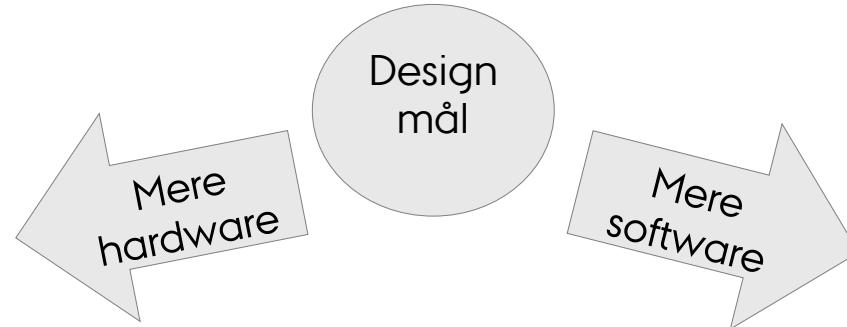
Aktiviteter

- Planlægge hvornår forskellig funktionalitet afvikles på de forskellige ressourcer.
 - Direkte spec. af rækkefølger, eller
 - Specifikation af scheduler for hver CPU
 - Task prioriteter.

Øvrigt:

- Som partitioning i EUDP:
- Afgøre for hvert sub-system om funktionaliteten mest hensigtsmæssigt implementeres i HW eller SW.
- Opnå opdeling, som vil give krævet ydeevne i system
- Omkostninger, størrelse, effekt/energi, hastighed, osv

DESIGN TRADE-OFFS I HW/SW-MIX



Implementering i hardware giver relativt...

- Bedre ydeevne
 - Typisk hurtigere
 - Parallel processesering, forskellige hardware-acceleratorer
 - Garantier for realtid
 - Typisk højere energi-effektivitet.

Implementering i software giver relativt...

- Lavere udviklingsomkostninger
- Kortere udviklingstid
- Lavere designkompleksitet
- Bedre supportmuligheder: Softwaren kan opdateres senere.

FPGA'en er et eksempel hvor der "nemt" foretages et trade-off:

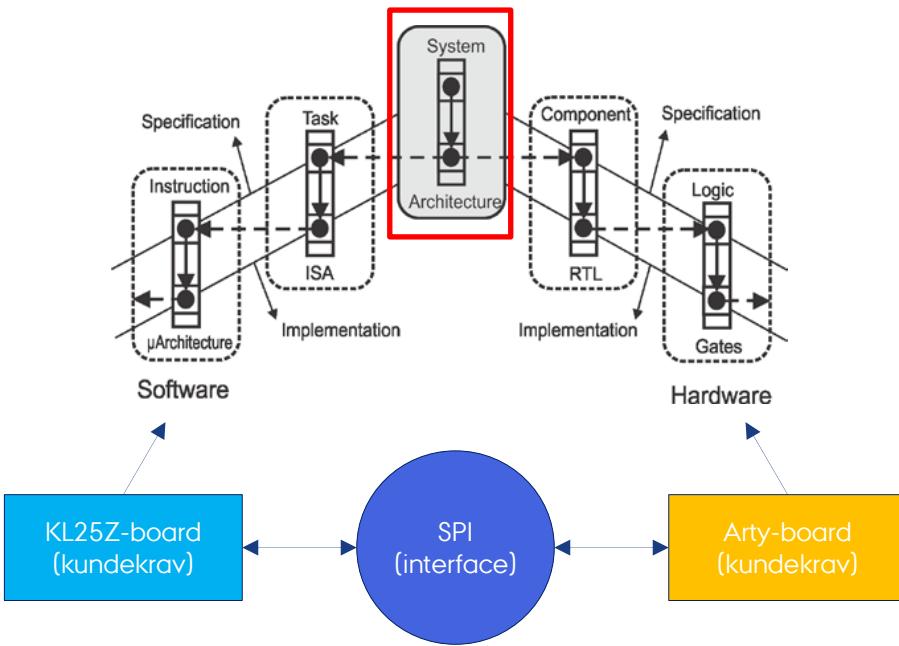
- SoC med syntetiserbar soft-processor.
 - Udvikling af HW, evt. med IP-kerner.
 - Udvikling af SW til afvikling på soft-processor

Tilsvarende skal foretages i fx DSP-applikationer eller udvikling af andre ASIC'er.

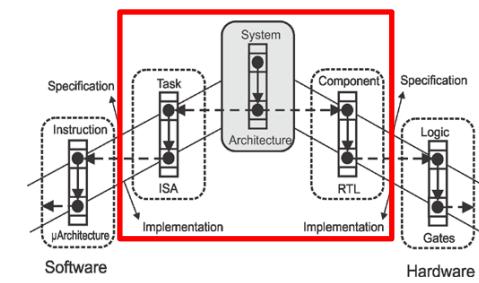
CO-DESIGN PROJEKT FORÅR 2020

Project brief, overordnede "behavioural" krav:

- Lysstyrke på 4 LED'er på Arty-board skal kunne styres via PWM.
- Brugerinterface: Bruger indtaster valg/værdier i konsol til KL25Z-board.
- KL25Z-board sender kommandoer til Arty-board via SPI.
- Arty-board (HW) håndterer stimulering af LED'er via PWM ift. ønsket lysstyrke.

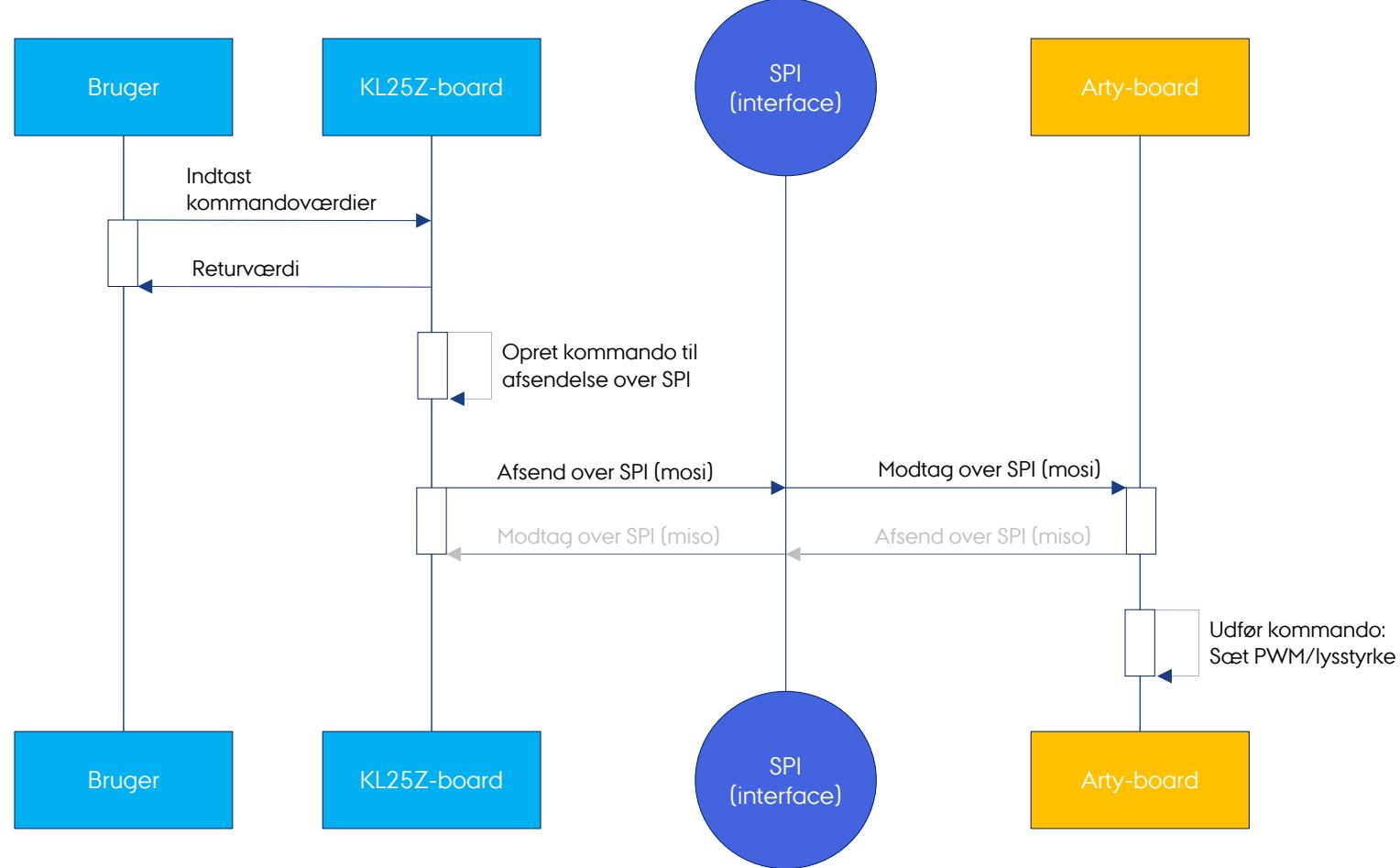


SEKVENSDIAGRAM OVERORDNET SYSTEM: SW+HW

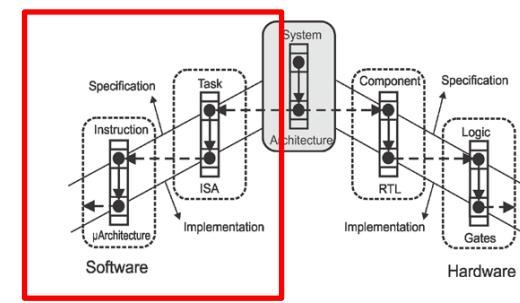


Sekvensdiagrammet viser den overordnede:

- **Allocation**
 - Til platforme / ressourcer
- **Binding**
 - Hvilken ressource udfører hvilken funktionalitet
- **Scheduling**
 - Rækkefølge i afvikling
 - Rækkefølge i kommunikation



SEKVENSDIAGRAM SOFTWARE-SIDE



Sekvensdiagrammet viser den overordnede:

• Allocation

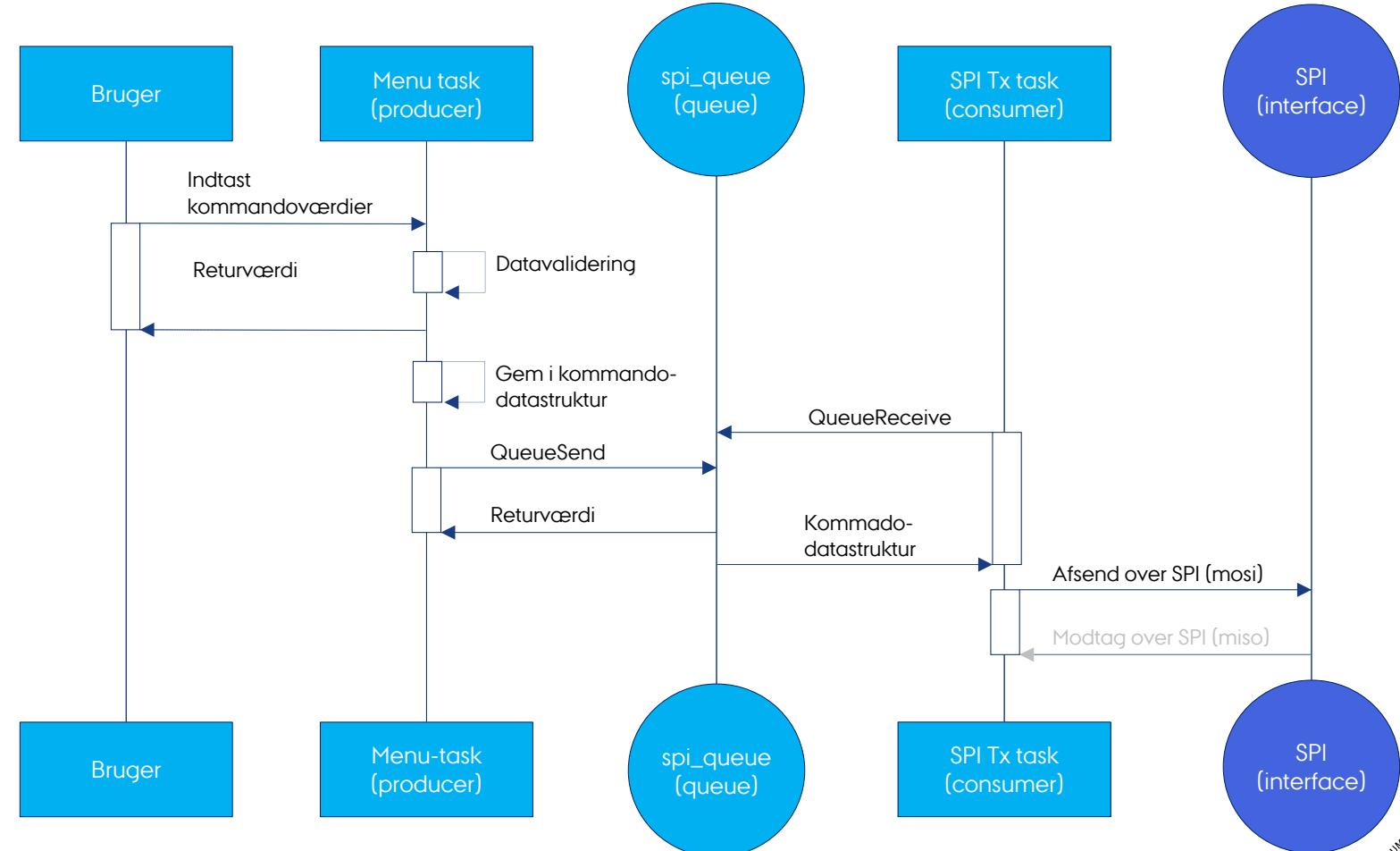
- To RTOS tasks allokeret til at udføre funktionalitet
- En queue til kommunikation mellem tasks

• Binding

- Ansvar i hver task

• Scheduling

- Synkronisering vha. RTOS queues



KOMMUNIKATION: PROTOKOL

En kommando sætter **intensitet** (0-255) for en farve (R, G, B) på en af de 4 RGB-LED'er på Arty.

En kommando består af **2 bytes**

- Byte 0: Adresse på LED-farve-kombination
- Byte 1: Intensitet (0-255)

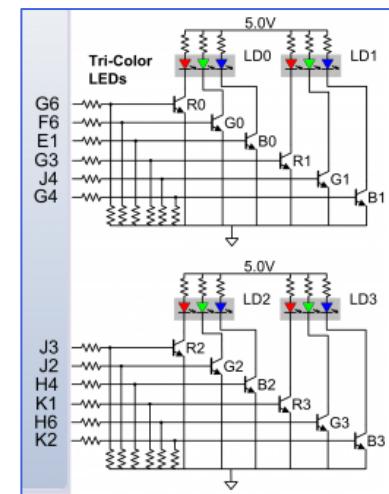
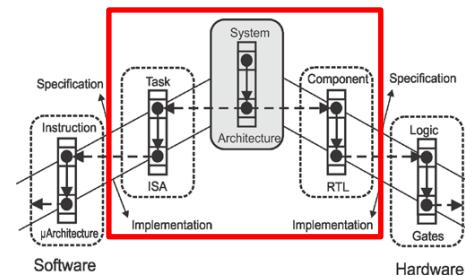
Adressetabel (se th.) i FPGA oversætter fra adresse til enable-signal vha. dekoder.

- Intensiteter (0-255) -> register på adresse

Datastruktur benyttes på KL25Z til at holde en kommando:

```
21 /* Data structure for data transfer */
22 typedef struct {
23     uint8_t addr;    // 1 byte address field
24     uint8_t data;    // 1 byte value field (0-255)
25 } fpga_cmd_t;
```

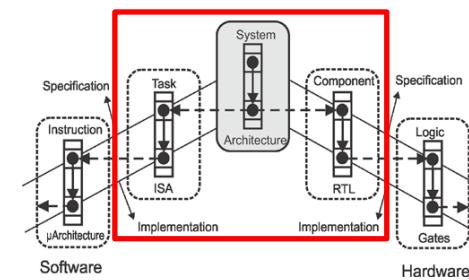
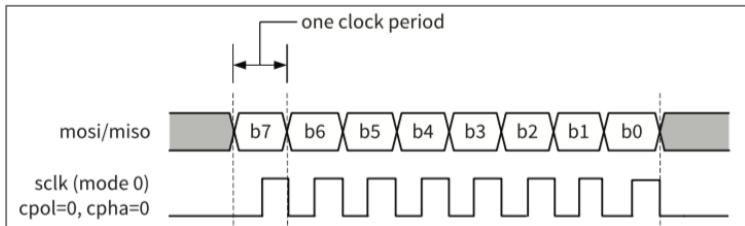
Adresse	LED	Farve	Arty Pin
0	LD0	R0	G6
1	LD0	G0	F6
2	LD0	B0	E1
3	LD1	R1	G3
4	LD1	G1	J4
5	LD1	B1	G4
6	LD2	R2	J3
7	LD2	G2	J2
8	LD2	B2	H4
9	LD3	R3	K1
10	LD3	G3	H6
11	LD3	B3	K2



INTERFACE OG PROTOKOL

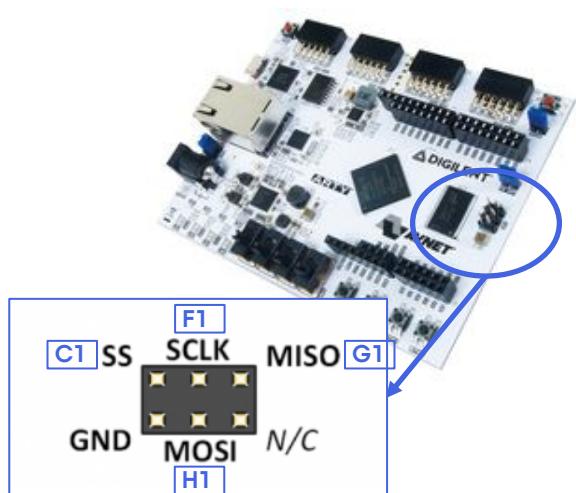
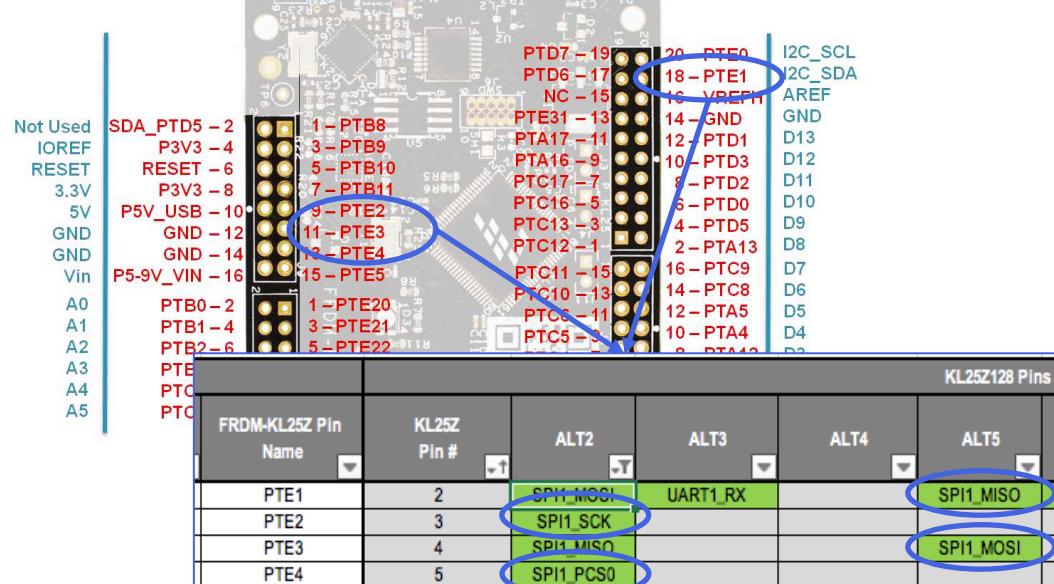
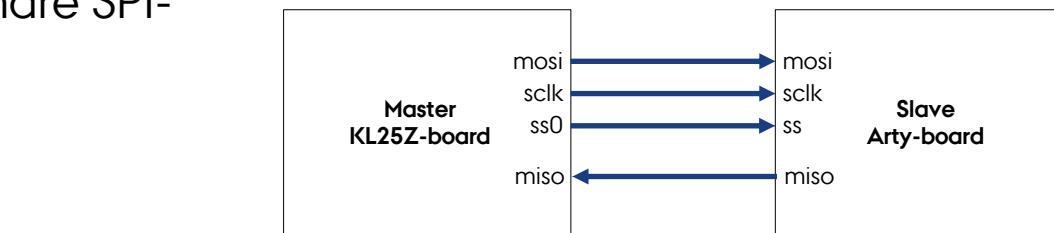
SPI-kommunikation:

- Som standard: 1 byte ad gangen -> 2 datapakker for at overføre en kommando
- Benytter standard (frem for at ændre SPI-implementeringen)

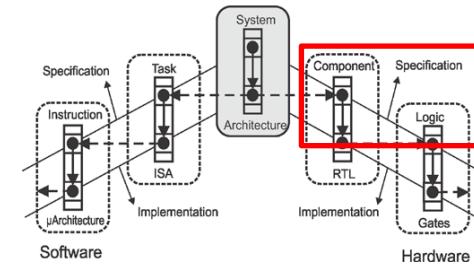


Interface setup:

- KL25Z-board -> *Master: SPI1*
- Arty-board -> *Slave: Custom SPI*
- 3,3 V interfaces
- SPI-indstillinger (se fig.):
 - SPI Mode 0
 - MSB først
 - SPI-clockfrekvens: ca. 46 kHz



HARDWARE-DESIGN



Principper og koncept

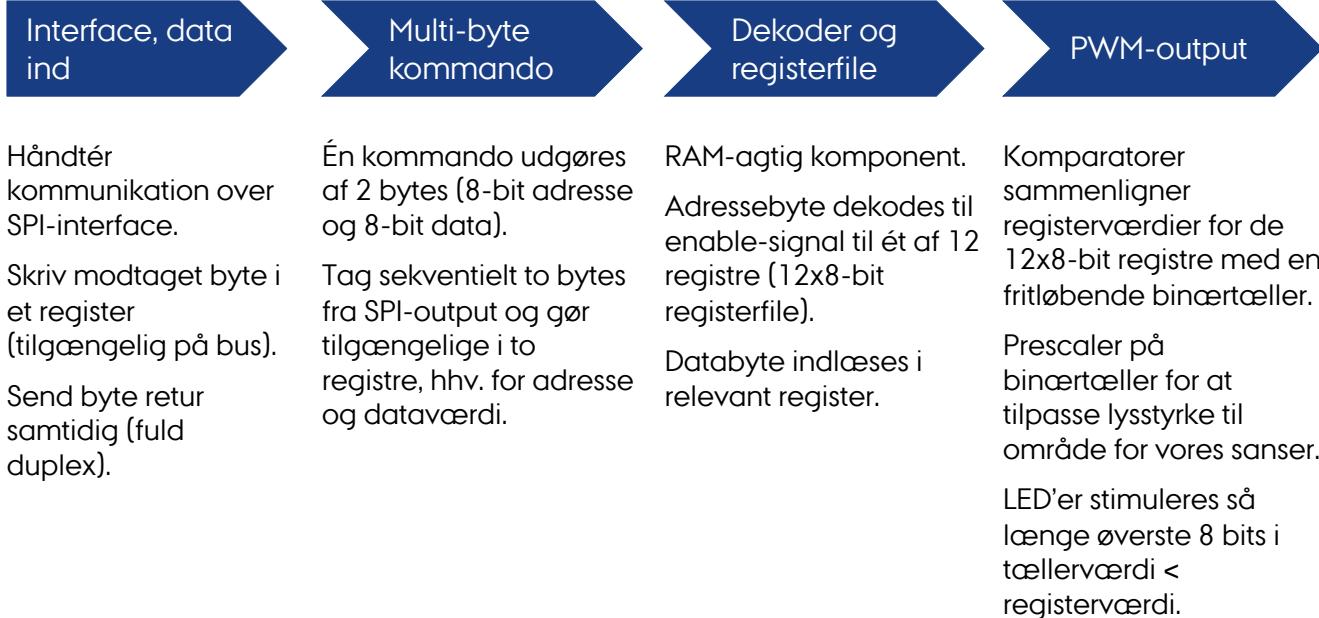
Overordnede design-principper:

- Anvender synkront design og FSM / FSMD'er.
- Modulære komponenter: Hvert HW-kredsløb er en genbrugelig, instantierbar komponent.
- Top-filen instantierer de relevante komponenter.

Overordnet design-koncept:

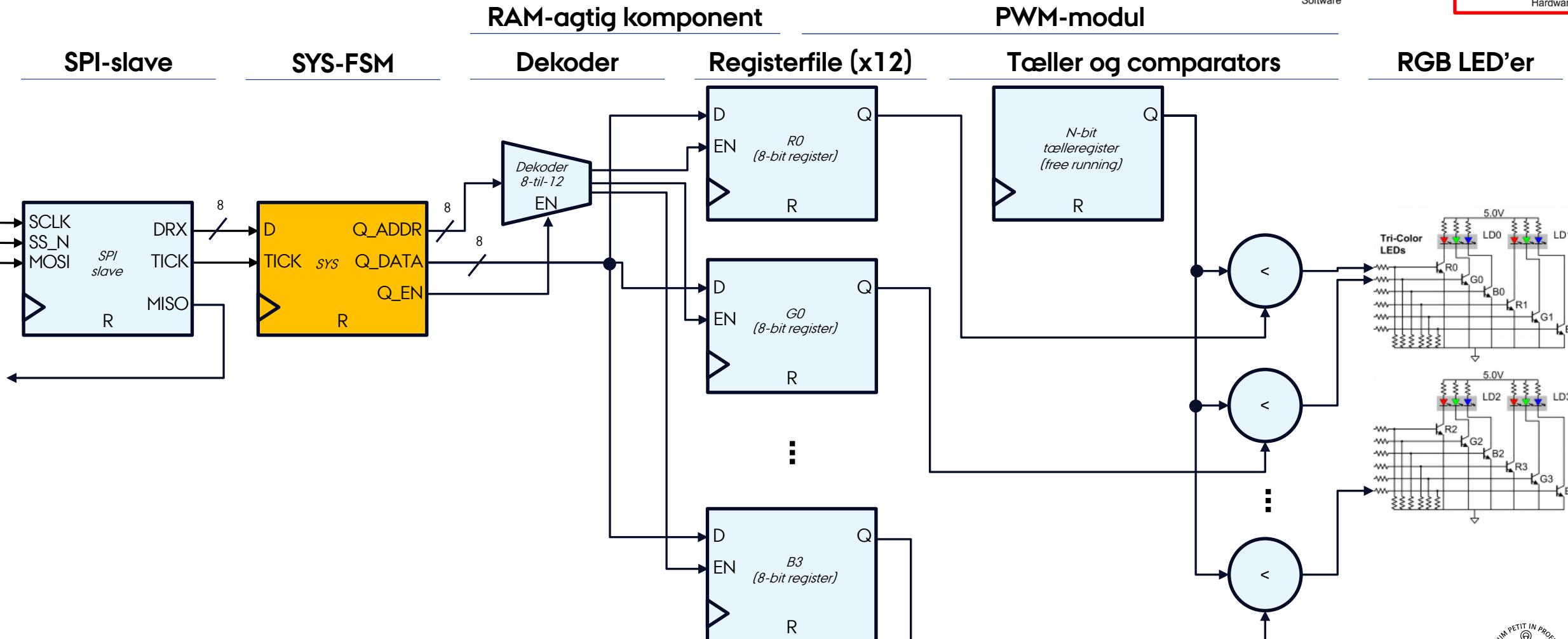
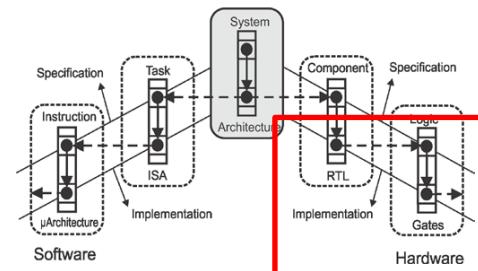
- **System struktureret som en "pipeline":**
 - Data ind -> behandling -> LED output
 - Hver komponent giver 'tick' når data er klar.
 - Application-specific!
 - Frem for et mere generisk system med fælles bus og central processesering.
 - Kunne have valgt at instantiere MicroBlaze MCS core, AXI bus, generisk SPI, osv., og så skrive drivers / kode.

"Pipeline"-struktur (behavioural)

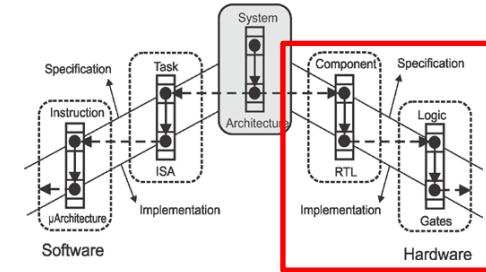


HW: BLOKDIAGRAM

RTL-STRUKTUR FOR SYSTEM



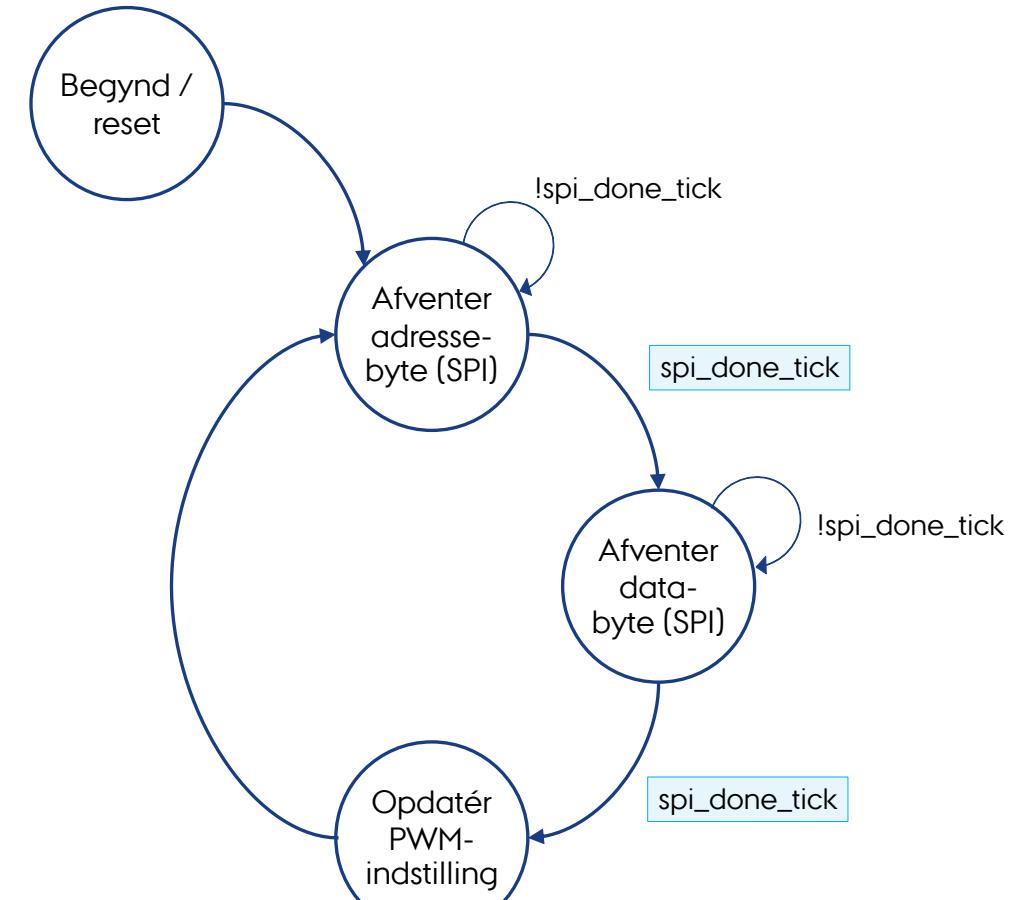
HARDWARE SYS-FSM ER KERNEN I SYSTEMET



Overordnet koncept:

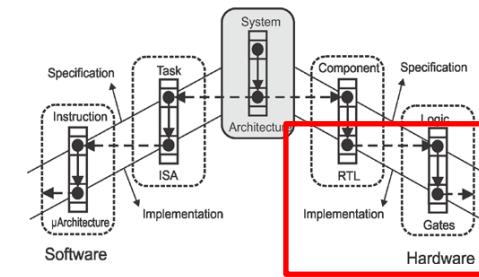
- SYS-FSM er synkront **bindeled** mellem indkommende SPI-transmissioner og PWM-stimulering af LED
- Modelleres som FSM (se t.h.)

State machine til system-tilstand (SYS-FSM)



SYSTEM-FSM (SYS_FSM)

SYNKRONT SYSTEM



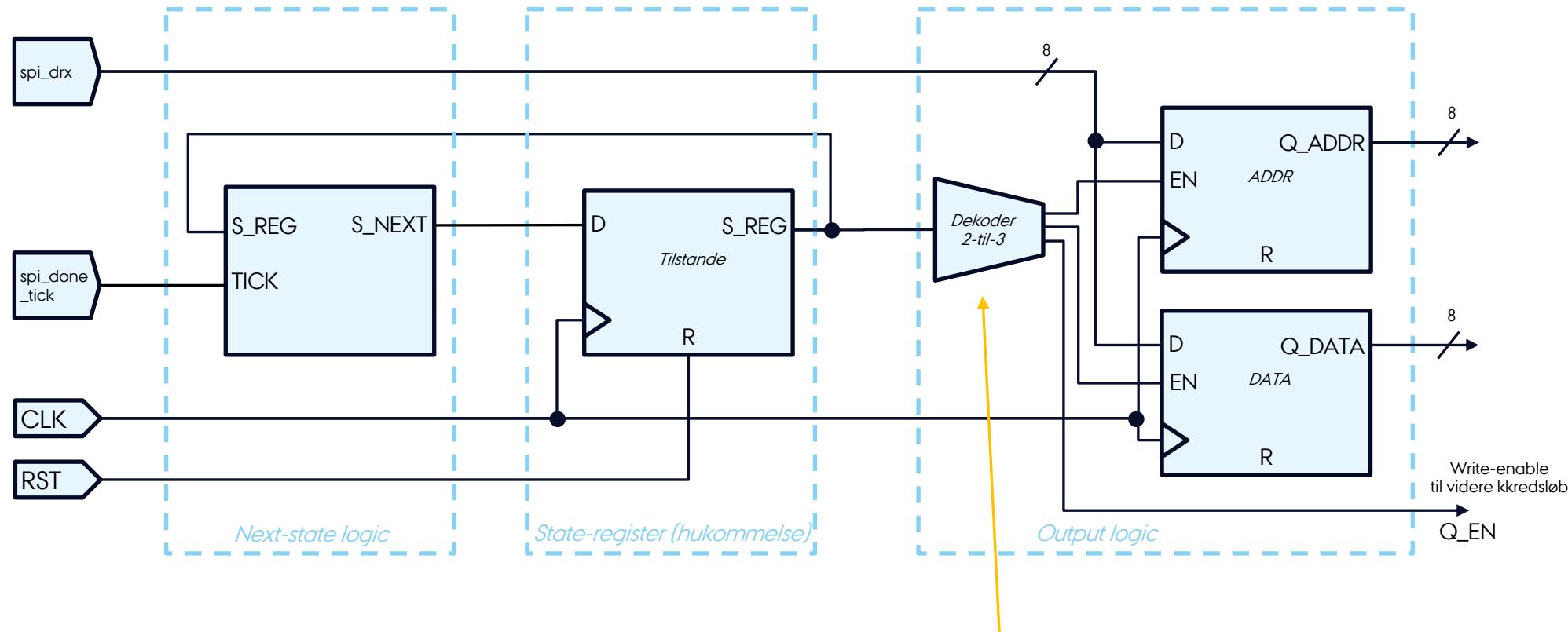
RTL-struktur for state machine til system-tilstand (SYS-FSM)

Mealy-type FSM

- Indeholder data path
- Tilstande avancerer ved SPI-done tick
- Output-logik afhængigt af input

Opdaterer registre i rækkefølge

- ADDR-register får første modtagne byte
- DATA-register får anden modtagne byte
- Endelig giver dekoder et højt write-enable signal til PWM registerfile



Idé:
 $S_REG == "00" \rightarrow Y(0:2) = "100"$,
 $S_REG == "01" \rightarrow Y(0:2) = "010"$,
 $S_REG == "10" \rightarrow Y(0:2) = "001"$.
 Men syntetiseret til 3 Mux'er.

SYS-FSM-MODUL

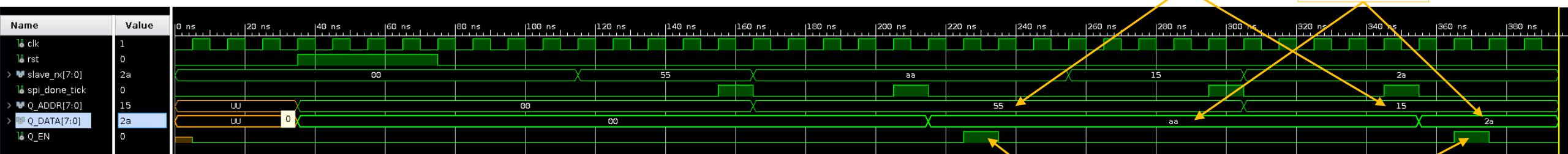
IMPLEMENTERING OG TEST

Implementeret som FSMD.

- Bytes fra SPI (simuleret) -> **SYS-FSM** -> Q_ADDR og Q_DATA (aflæses)

Testbench OK:

- Reset-knappen virker
- Der kan modtages flere kommandoer:
 - To bytes kommer ind fra SPI per hver kommando.
 - Kommando 1 := [55, AA]. Kommando 2 := [15, 2A].
 - Efter hvert *spi_done_tick*='1' ses at,
 - Q_ADDR* opdateres først, dernæst opdateres *Q_DATA*.
 - Efter adresse-bus og data-bus er opdateret, sendes et enable-tick til det videre kredsløb.



HARDWARE

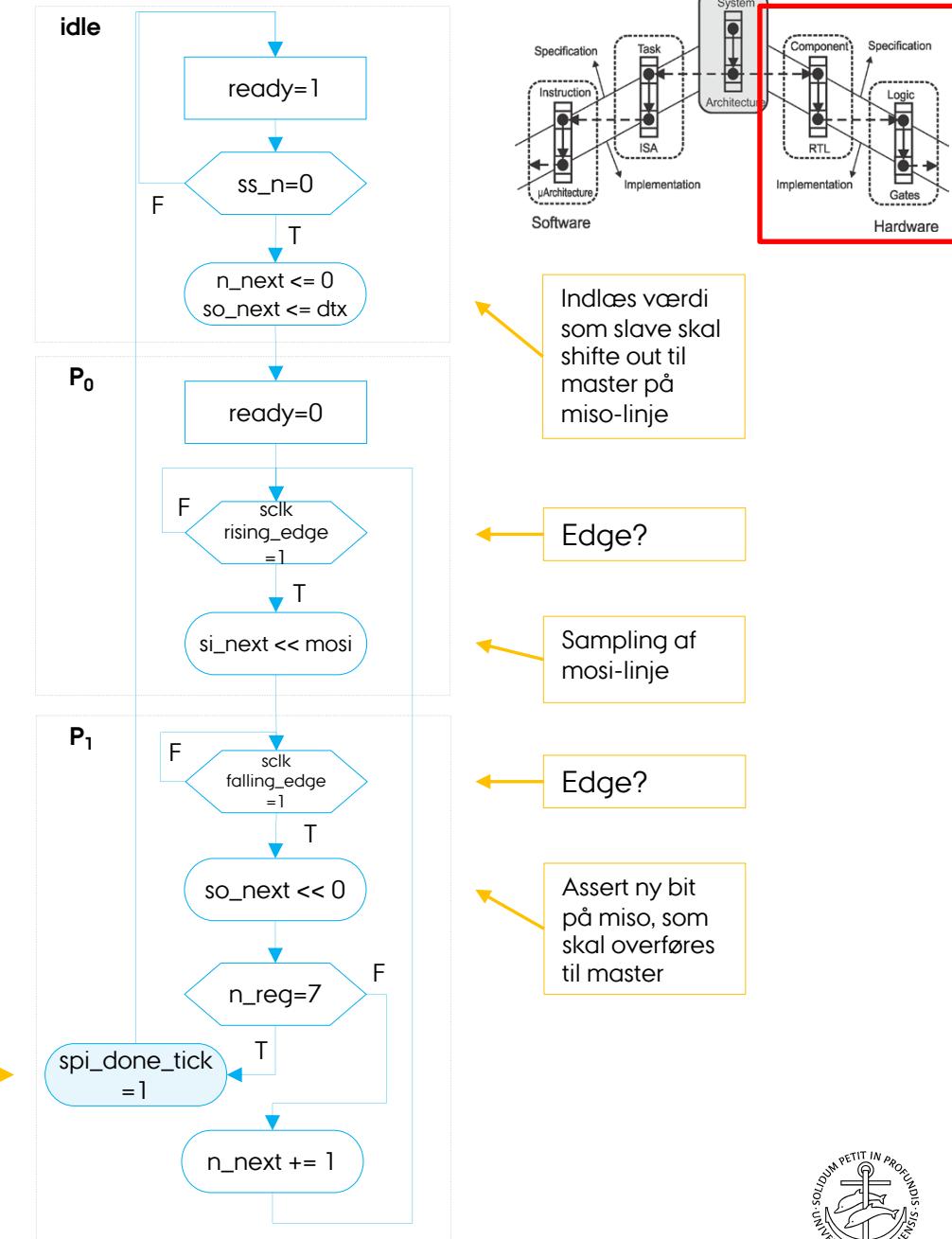
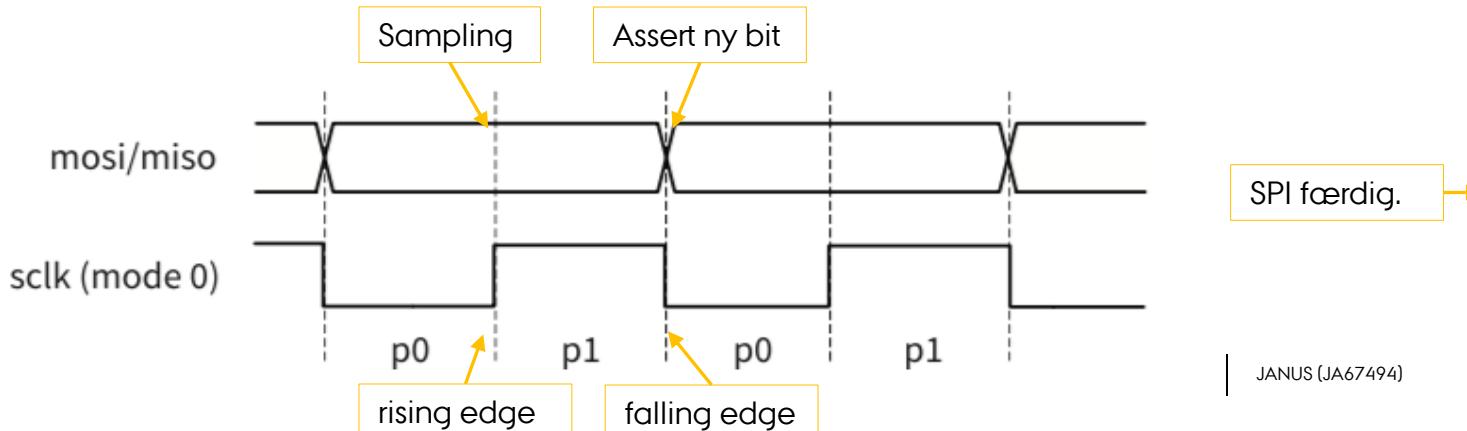
SPI SLAVE

Komponenten skal håndtere modtagelse (og afsendelse) af 8 bits over SPI-interface

- Interface er "Mode 0"
- Skal præsentere modtaget byte i et output-register (på bus)

Koncept:

- Modelleret som FSM (se t.h.):
 - Skifter tilstand i takt med SS og SCLK
 - SS og SCLK er oversamplet med sys_clk (100 MHz)
 - Gør systemet synkront
 - Synkron edge detector på SCLK.
 - Giver output tick hver gang 8 bits er overført og klar i register.



SPI SLAVE-MODUL

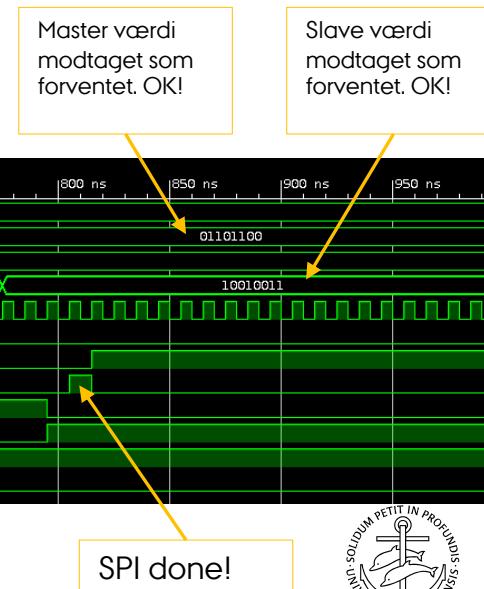
IMPLEMENTERING OG TEST

Implementeret som FSM og med fuld-duplex (shifter ind og ud samtidig med to skifteredistre).

- Simulerede værdier over SPI-interface (SS, SCLK, MOSI, MISO) i tråd med SPI-specifikation (mode 0) -> **SPI-modul** -> registrerværdier (aflæses).

Testbench OK:

- Reset-knappen virker
- Master sender *master_tx* <= 0b10010011, slave sender *slave_tx* <= 0b01101100
- Efter *spi_done_tick*='1' ses at,
 - *slave_rx* indeholder nu masters besked
 - *master_rx* indeholder ni slaves besked

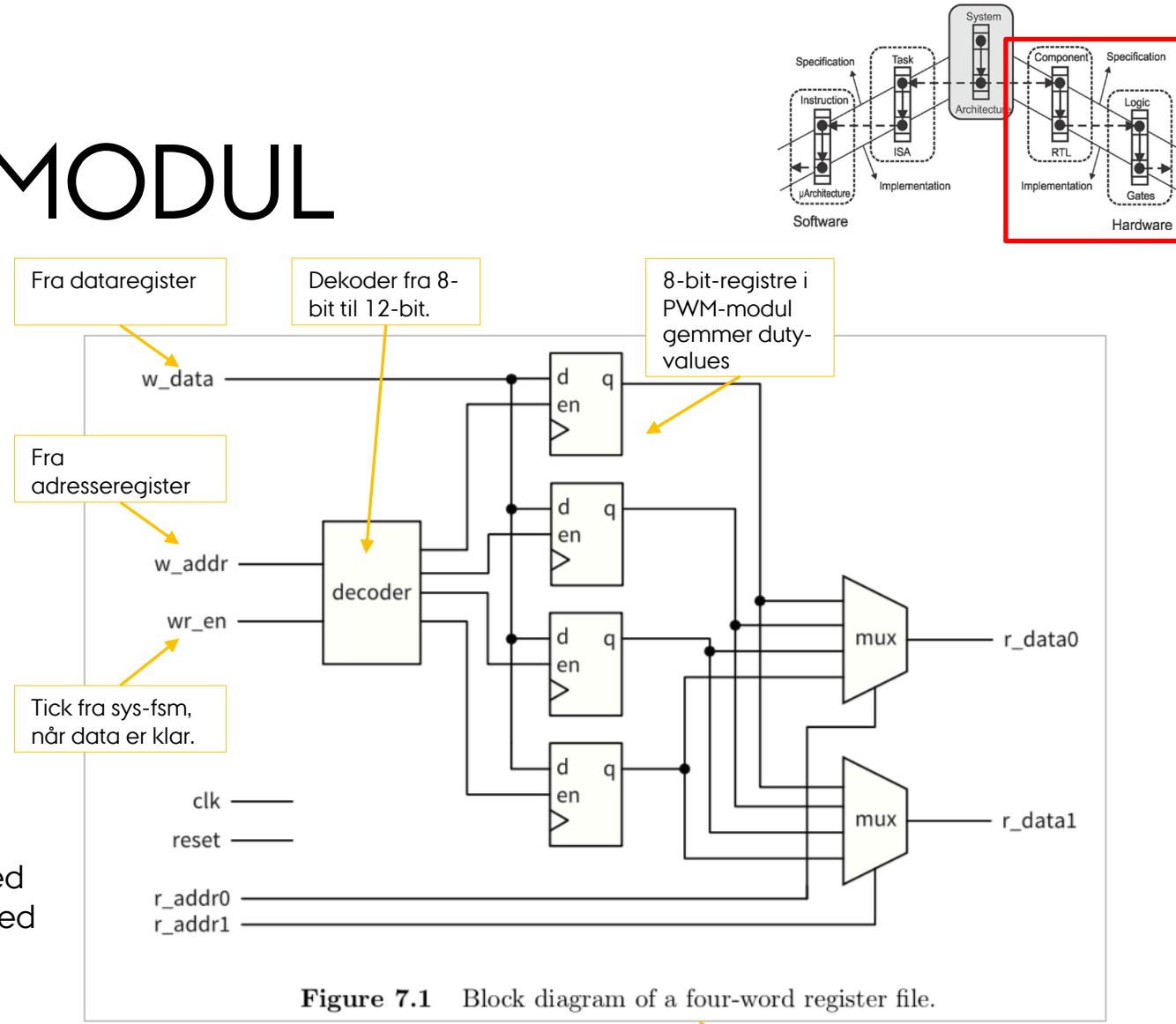


Name	Value
> master_tx[7:0]	10010011
> master_rx[7:0]	01101100
> slave_tx[7:0]	01101100
> slave_rx[7:0]	10010011
clk	0
rst	0
ready	1
spi_done_tick	0
sclk	0
ss_n	1
mosi	1
miso	0

HARDWARE DEKODER OG PWM-MODUL

Design-koncept:

- Dekoder og registre i PWM-modul fungerer sammen som en RAM-agtig komponent.
 - Inspiration: Chu s. 147 ff. (se fig. 7.1 t.h.)
- **Dekoder** skal oversætte en 8-bit (`uint8_t`) adresseværdi til enable-signaler til de 12 PWM-registre (registerfile).
- **PWM-modul** skal stimulere LED'er med defineret duty cycle.
 - Gemmer duty-value i 12x8-bit registre
 - Én fritløbende binærtæller avancerer synkront på clk.
 - Comparator sammenligner øverste 8 bits i binærtæller med registrerværdier, og stimulerer MOSFET til tilhørende LED med 3,3V eller 0V.
 - Baseret på eksempel med 1 RGB-diode fra MOJ / SMM.



I dette tilfælde bygger vi et 12 word register. Et word har en bredde på 8 bits.

DEKODER-MODUL

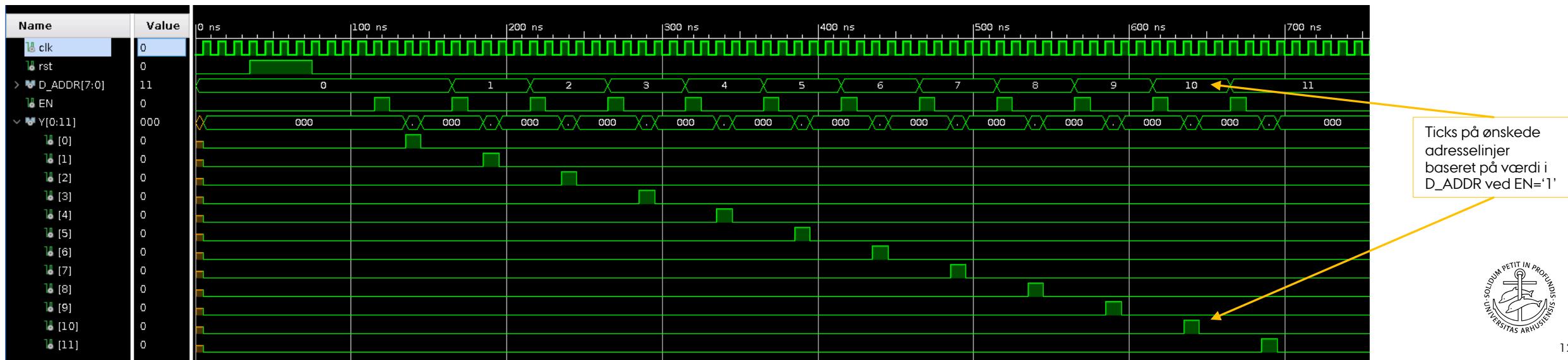
IMPLEMENTERING OG TEST

Implementeret som synkront system med synkron sampling af D_ADDR og EN, hvilket giver 2 clock cycles delay.

- SYS-FSM adresseregister (simuleret) -> **Dekoder** -> Enable-signaler til PWM-registre (aflæst)
- Skal give et enable-tick til PWM-registerfiles, så de indlæser ny data.
- Kunne evt. forbedres med ren kombinatorisk dekoder

Testbench OK:

- Giver tick på ønsket adresselinje, baseret adresse-værdi modtaget fra SPI



PWM-MODUL

IMPLEMENTERING OG TEST

Denne test køres med en 4-bit tæller og 2-bit PWM-værdi (generic mapping), da simuleringen ellers vil tage alt for mange clock cycles at gennemføre.

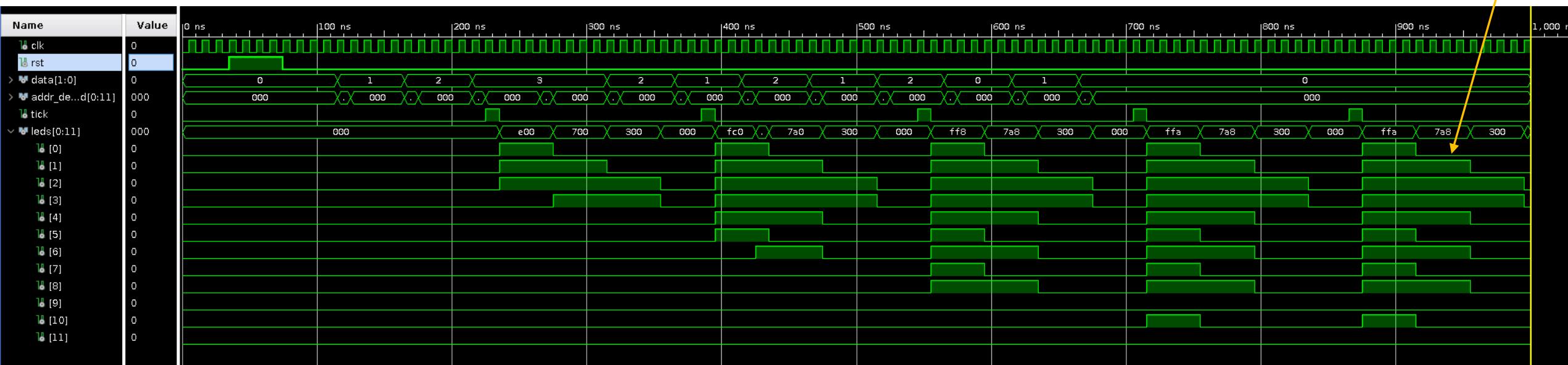
Implementeret 12-LED (4 x RGB) udvidelse til MOJ's skabelon via én binærtæller, registerfile med 12 registre og 12 comparators.

- Enable fra dekoder (simuleret) og data fra SYS-FSM (simuleret) -> **PWM-modul** -> duty-cycles til LED'er (afslæst)

Testbench OK:

- Binærtæller generisk mappet til $N=4$ bits -> overflow med tick hver 16 clock-cycles.
- Dataværdi generisk mappet til $DV=2$ bits bredde -> 4 mulige indstillinger for PWM duty cycle (0%, 25%, 50%, 75%).
 - I HW-implementering laves binærtæller $N=16$ og $DV=8$, så der er 255 mulige indstillinger, de laveste 8 bits er prescaling.
- PWM-registre kan loades med forskellige værdier, *addr[0:11]* er enable signaler, *data[0:DV-1]* indeholder PWM værdi.
- Testeksempel: Indsat forskellige duty cycles på de 12 LED'er, *leds[0:11]*

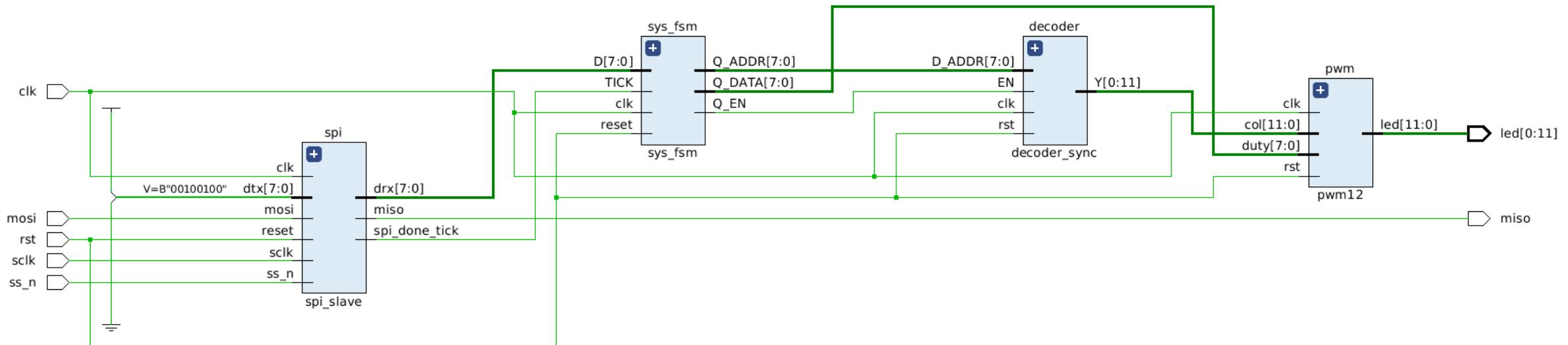
Ønskede PWM
duty cycles for
de 12 LED'er.
OK!



SAMMENSÆTNING AF HW-SYSTEM

Design og implementering:

- Én top-fil:
 - Instantierer og forbinder udviklede komponenter.
- Constraints:
 - Forbinder til I/O: SPI-interface pins, on-board clock (100 MHz), reset-knap og de 4 RGB-LED'er.



INTEGRATIONSTEST

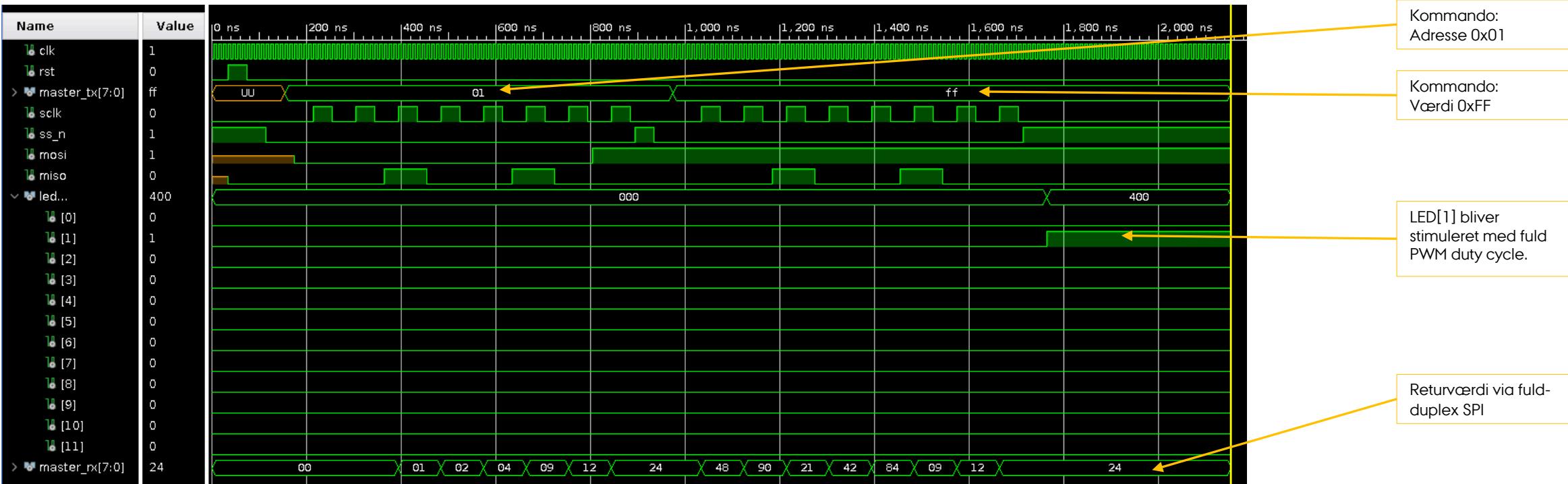
TEST AF 4 MODULER SAMMEN

Sammensætning af hele HW-systemet.

- Tester top-filen, dvs. integreret HW-system.
- Dataflow: SPI-interface (simuleret) -> **SPI-modul** -> **SYS-FSM** -> **Dekoder** -> **PWM-modul** -> LED'er (aflæst)

Testbench OK!:

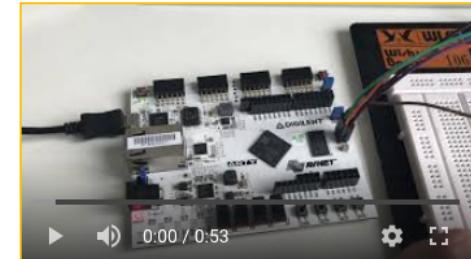
- Sender kommando for at sætte LED[1] (LD0, grøn) til højeste duty cycle => (byte1, byte2) = (0x01, 0xFF)
- Resultat, LED[1] bliver stimuleret som ønsket.



HARDWARE-TEST

Test OK!:

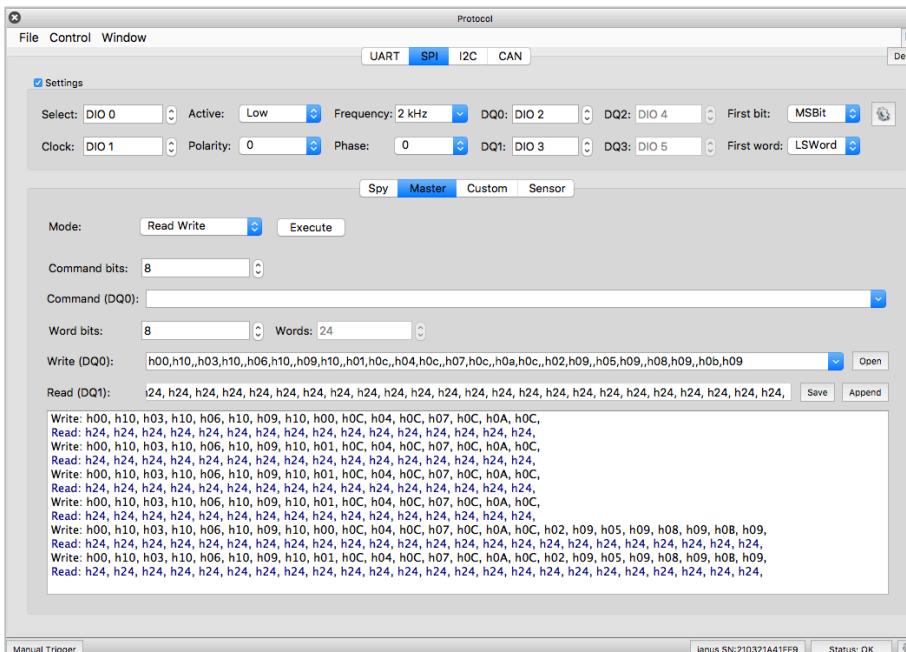
- Kommandoer til at fire RGB'er overføres og virker som forventet.
- SPI Slave returnerer den hardcodede returværdi (0x24)
- Reset virker.



https://youtu.be/jqSOHixNE_w

Overfør kommandoer via SPI for alle fire RGB'er

Analog Discover er SPI Master, og overfører til Arty SPI Slave



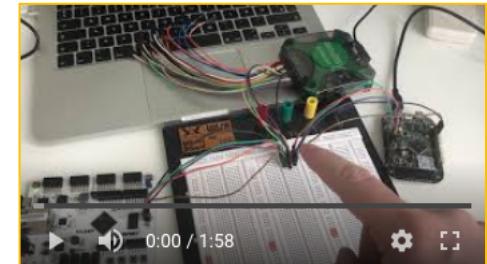
SAMLET SYSTEM



SAMLET SYSTEMTEST

Systemtest OK!:

- Kommandoer overføres fra KL25Z-board til Arty-board via SPI-interface.
- Stimulering på LED'er modsvarer brugerens ønsker / indtastning.
 - Værdi 0 slukker dioden, værdi 255 giver maksimal lysstyrke.
- Alle 4 RGB LED'er (12 forskellige dioder/adresser) kan stimuleres.



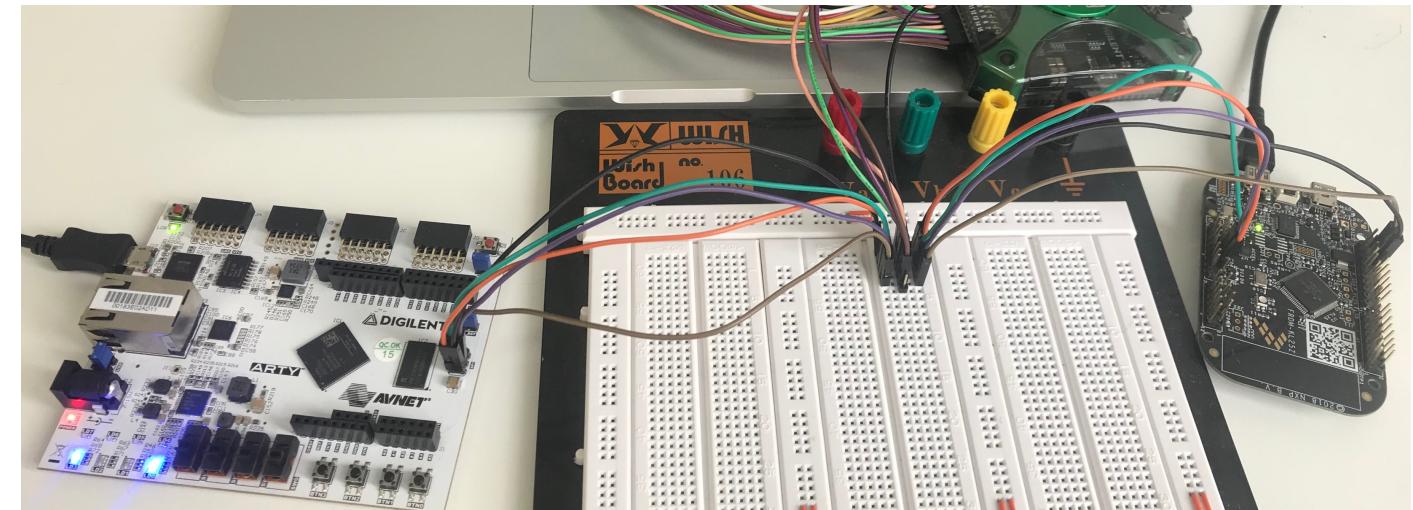
<https://youtu.be/TKfKw5NFccQ>

Bruger indtaster kommandoer i konsol

```
E4ISD2_co-design PE Debug [GDB PEMicro Interface Debugging] Semihosting Console
P&E Semihosting Console

Hello. Co-design Spring 2020.
Enter LED number (address) 0-11:
0
Entered value: 0
Enter PWM value 0-255:
255
Entered value: 255
Sending.
Sent over SPI.
Enter LED number (address) 0-11:
2
Entered value: 2
Enter PWM value 0-255:
255
Entered value: 255
Sending.
Sent over SPI.
```

Opstilling – kommandoer overføres over SPI-interface



KONKLUSION OG EVALUERING CO-DESIGN-ØVELSE FORÅR 2020

Resultat:

- Vellykket udvikling af HW+SW-system, der implementerer kundens minimumskrav.
- Forsøgt at følge double-roof-modellen i udviklingen.

Læringer:

- Tidlige analyser og design-beslutninger meget vigtige:
 - Interfaces og funktionelle bindinger "varer ved" og er svære at ændre undervejs. Do it right the first time!
- STOR betydning af krav/allokering til platforme:
 - MEGET kortere vej til målet, hvis al funktionalitet var allokeret, bundet og implementeret i software, eller allokeret til to forskellige software-programmérbare platforme (fx KL25Z og Arduino).
- Udvikling og test af HW tager faktor 10 (eller mere) ift. udvikling af software/C-kode.

Konklusion:

- Sjov øvelse! Meget tidkrävende.
- Banker betydning og behov for co-design (og en god udviklingsproces) fast.

APPENDIKS: DMA



AARHUS
UNIVERSITET