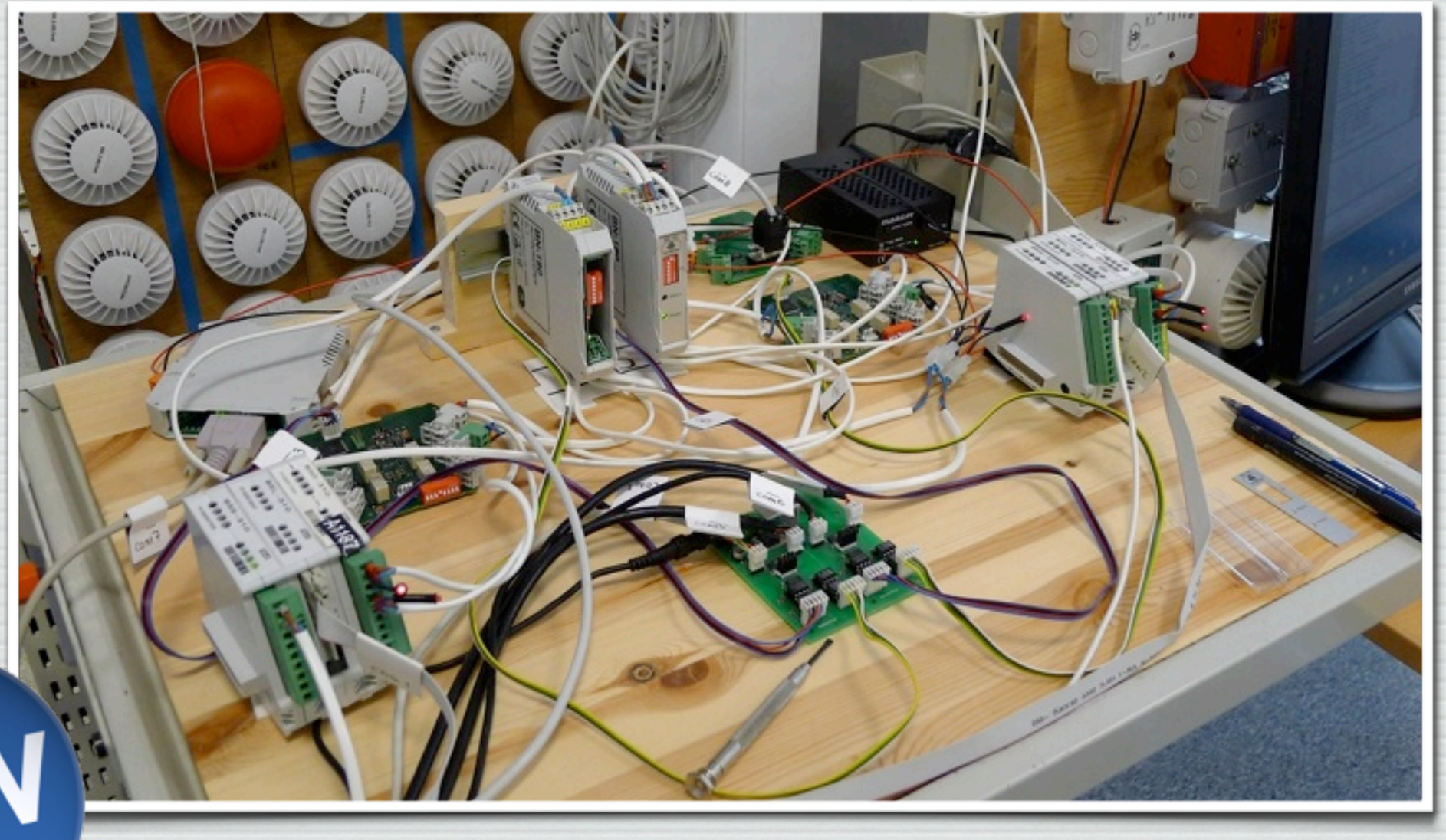
Fra harde µ-sekunder til forte år: sann tid i industrien



Øyvind Teig
senior utviklingsingeniør, Autronica
Gjesteforelesning, 25. april 2013
NTNU, fag TTK 4145 Sanntidsprogrammering (Real-Time Programming)
Dette foredraget: http://www.teigfam.net/oyvind/pub/NTNU_2013/foredrag.pdf

Autronica Fire and Security (AFS)

- ✓ Ca. 370 ansatte med ca. 25 på utvikling
- ✓ 825 mill NOK omsetning (2012)
- ✓ På Lade, her i Trondheim
- ✓ Samme sted, og for meg - seks logoer
- ✓ Begynte med "gründerne" i 1957
- ✓ Børs og forskjellige eiere

Autronica Fire and Security (AFS)
Siden 2005

- ✓ "A UTC Fire & Security Company"
  United Technologies Corporation
- ✓ Ca. 200.000 ansatte
- ✓ Otis, Pratt & Whitney, Carrier, Sikorsky, UTC Aerospace Systems (Hamilton Sundstrand, Goodrich), UTC Power (Fuel Cell Technologies), Clipper Windpower, Autronica, etc..

"Disclaimer"

Punktene i denne forelesningen står for egen regning!

- Datafaglige erfaring og meninger,
erfart, sett og lest

- Ingen Autronica-sensitiv informasjon

- Autronicas produkter godkjennes etter aktuelle normer

CV

- ✓ NTH, 1975
- ✓ Autronica, 1976-2013 (+studentjobbing der fra 1972)
- ✓ HW og SW (mest SW)
- ✓ Har jobbet med embeddedsystemer hele tida
- ✓ Publisert en del - relativt uvanlig i industrien
- ✓ Hjemmesiden min: http://www.teigfam.net/oyvind

«Pappesken» (SW)

- ✓ HW/SW for 30+ år i den pappesken
- ✓ Diverse assembler (78-80)
- ✓ PL/M (80-90)
- ✓ Modula-2 (88-90)
- ✓ MPP-Pascal (82-88)
- ✓ occam (90-01)
- ✓ C (02-nå)
- ✓ Java (97-00)
- ✓ Perl (02)

«Pappesken» (SW)

Øverst til høyre (fra diplomen 1975)

Den papirtapen inneholder fremdeles programmet!

«Pappesken» (SW)

Sann tid går alt for fort!

Øverst til høyre (fra diplomen 1975)

Den papirtapen inneholder fremdeles programmet!

- ✓Sanntidsprogrammering er gøy!
- ✓Det kommer til å bli mye mer av det!
- ✓Inkludert "concurrency"

«Pappesken» (sanntidsmetodikk & produkt)

- ✓ 1978: Ikke noe kjøresystem, ren assembler
  Dieselaggregat start/stopp nødstrøm
- ✓ 1979: MPP Pascal med prosessbegrep
  Protokollkonvertering, nivåmåling og brannvarsling
- ✓ 1980: PL/M med NTH-utviklet kjøresystem
  Maskinromsovervåking
- ✓ 1982: Assembler med jobbsnekret kjøresystem
  Brannvarsling
- ✓ 1988: PL/M med jobbsnekret kjøresystem
  Brannvarsling

«Pappesken» (sanntidsmetodikk & produkt)

- ✓ 1988: Modula-2 med kjøpt kjøresystem
  Brannvarsling
- ✓ 1990: occam med prosessbegrep
  Motormålinger og effektberegninger
- ✓ 1995: C med VxWorks opsys/kjøresystem
  Brannvarsling
- ✓ 2003-8: C med CSP kanaler oppå SDL kjøresystem (AutroLooper)
  Brannvarsling
- ✓ 2009: "ChanSched" stand-alone kjøresystem, med kollega..
  Brannvarsling
- ✓ 2010: ..brukt i patentert enhet (AutroKeeper) i cruiseskip med det nye internasjonale kravet "Safe Return to Port"
  Brannvarsling

Lange år

- OO er fra sekstitallet
  - I bruk nå, men lite i embedded (blir mer og mer!)
- Sanntidsmekanismene er fra 60-70+ tallet
  - Semafor etc. i bruk siden da
- CSP etc. fra 70-tallet. Lite i bruk (ennå..(?))
- UML 2.0 er komplekst, og har flere aksjonsspråk under paraplyen «Executable UML»
- Java -"synchronized" - det første allment brukte språket som har concurrency-tankegang innebygd..
  - ..og som er "livsfarlig" å bruke!
  - ..men som det går an å bygge CSP oppå!

Lange år

- ✓ OO er fra sekstitallet
  - ✤ I bruk nå, men lite i embedded (blir mer og mer!)
- ✓ Sanntidsmekanismene er fra 60-70+ tallet
  - ✤ Semafor etc. i bruk siden da
- ✓ CSP etc. fra 70-tallet. Lite i bruk (ennå..(?))
- ✓ UML 2.0 er komplekst, og har flere aksjonsspråk under paraplyen «Executable UML»
- ✓ Java -"synchronized" - det første allment brukte språket som har concurrency-tankegang innebygd..
  - ✤ ..og somler "livsfarlig" å bruke!
  - ✤ ..men som det går an å bygge CSP oppå!

Dette går da ikke fort!

Lange år

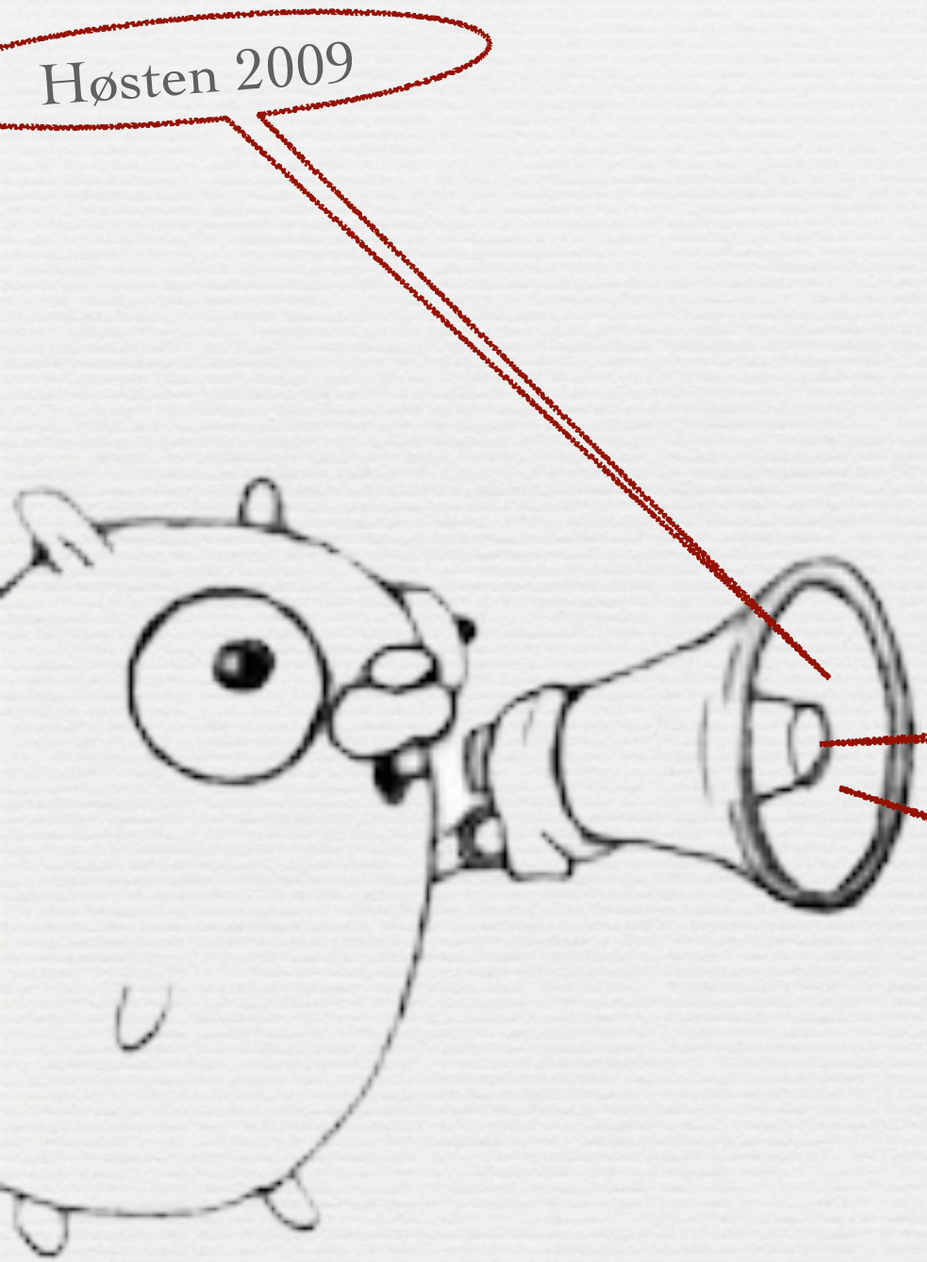
- ✓ OO er fra sekstitallet
  - ✤ I bruk nå, men lite i embedded (blir mer og mer!)
- ✓ Sanntidsmekanismene er fra 60-70+ tallet
  - ✤ Semafor etc. i bruk siden da
- ✓ CSP etc. fra 70-tallet. Lite i bruk (ennå..(?))
- ✓ UML 2.0 er komplekst, og har flere aksjonsspråk under paraplyen «Executable UML»
- ✓ Java -"synchronized" - det første allment brukte språket som har concurrency-tankegang innebygd..
  - ✤ ..og som er "livsfarlig" å bruke!
  - ✤ ..men som det går an å bygge CSP oppå!

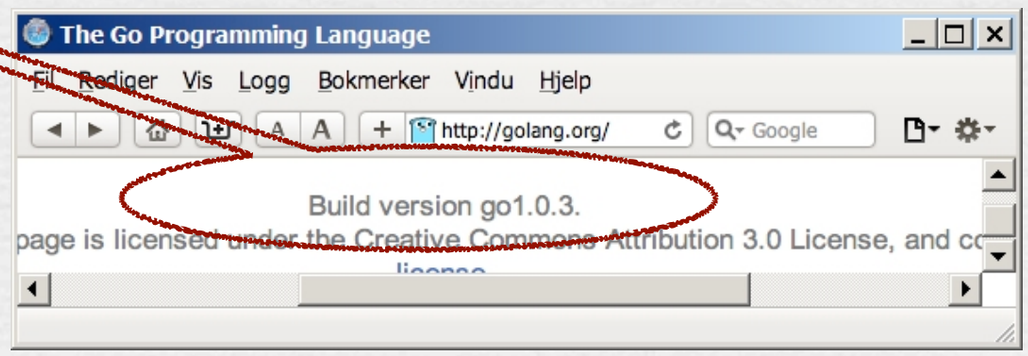
men...

21 år etter occam (i 2009)

- ✓ **Go** fra Google plukket med seg litt occam!
- ✓ **chan**, **go**, **select**, eksplisitt typekonvertering, array-oppdeling, ikke pekerartitmetikk
- ✓ Ikke "sikkert" (no «parallel usage rules»)

25 år etter occam (i 2013)





Etter forelesningen ønsker jeg

- ✓ At dere skal huske at å jobbe med embedded-systemer, er gøy
  - ➡ ...
- ✓ "En sivilingeniør har til oppgave å løse praktiske problemer, ofte i team og i samarbeid med kunden". (Wikipedia 2007)
- ✓ At dere skal huske å fortelle om det dere har lært i dette faget når dere havner i slike team!

Små embeddedsystemer i industrien

- ✓ Vil nok forholde seg til C en god stund til!
- ✓ Vil nok ha prosjekt- ledere og -deltakere som (bare) kan asynkrone system en god stund til!
- ✓ Ikke overta deres vertøykasse uten å legge synkrone kanaler og tette prosesser oppi:
- ✓ *Sikre* bufrede kanaler (asynkron til full)

Kjøresystem I

For små (AVR/XMega) system i ANSI C

1. Begynte vi med et asynkront **SDL** kjøresystem (brukes mye!)
2. La så et synkront kanal-lag oppå **CHAN_CSP** (brukes, men ikke i nye)
3. Har nå lagd et nakent synkront kjøresystem **ChanSched** (brukes)
4. Framtid? **BuffXChanSched** == en drøm!-)

Kjøresystem II

For større 32-bits system (AVR32 og ARM)

- Linux
  - Buildroot-basert
  - Kjerne 2.6
  - Egenmodifisert uClibc
  - Egne drivere (RAM-test, USB, HSR Ethernet (redundant ring), etc., alt med →GPL← lisens)

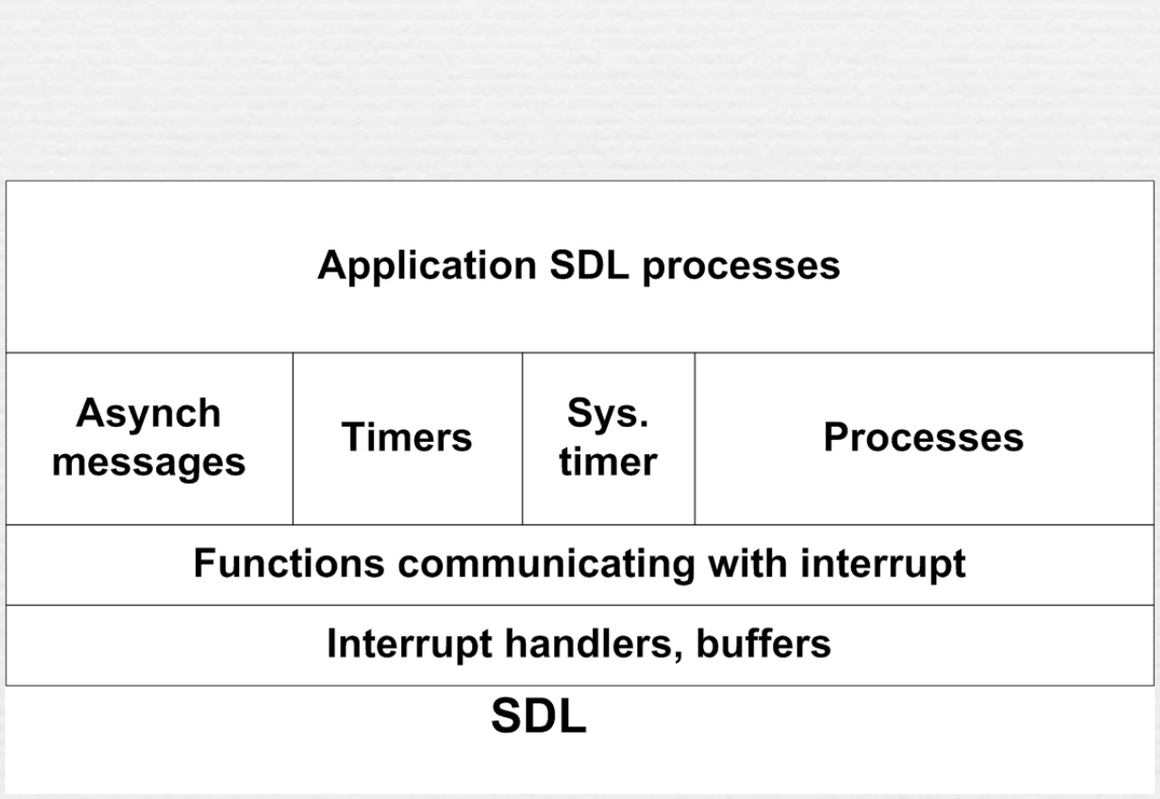
Kjøresystem II

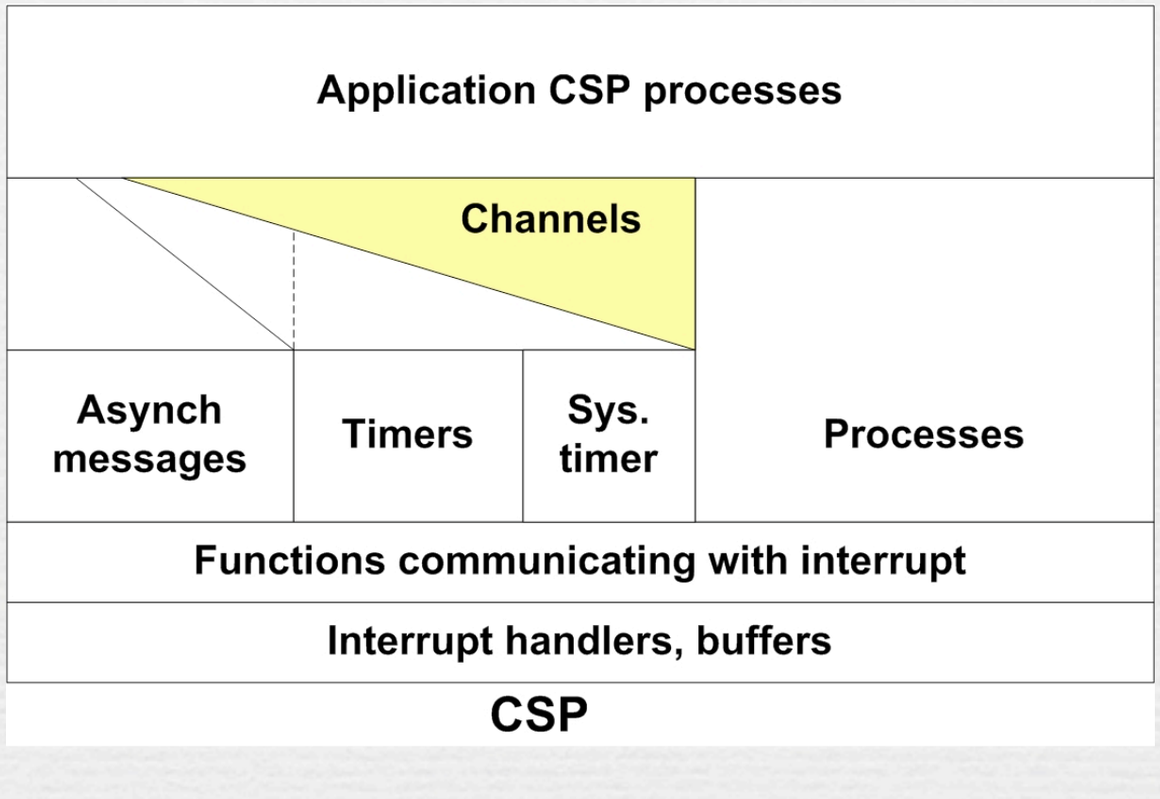
For større 32-bits system (AVR32 og ARM)

- Linux
  - Buildroot-basert
    - Kjerne 2.6
    - Egenmodifisert uClibc
    - Egne drivere (RAM-test, USB, HSR Ethernet (redundant ring), etc., alt med →GPL← lisens)

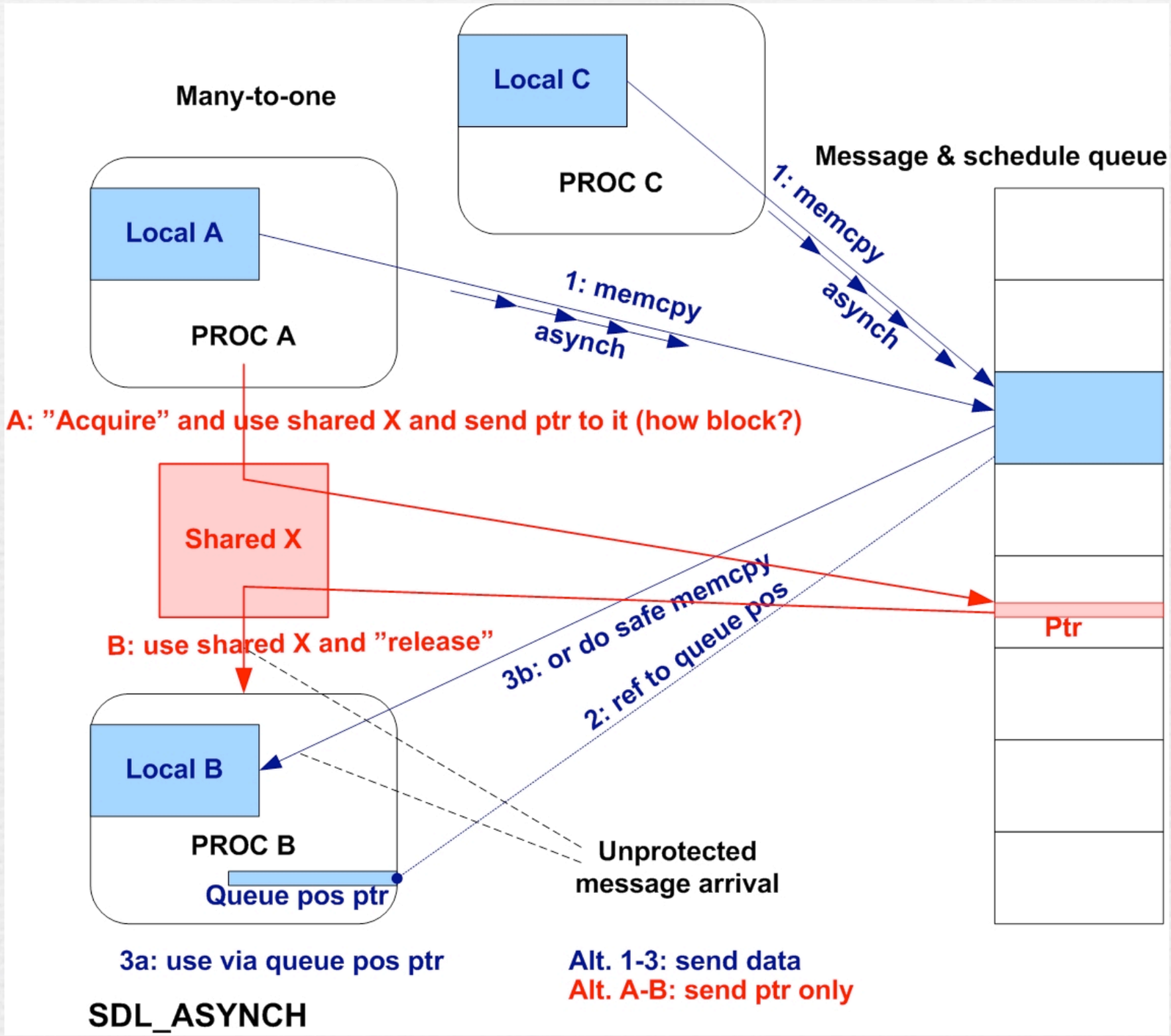
Ikke vår kjernekompetanse (som er brannvarsling!)

1: CHAN_CSP

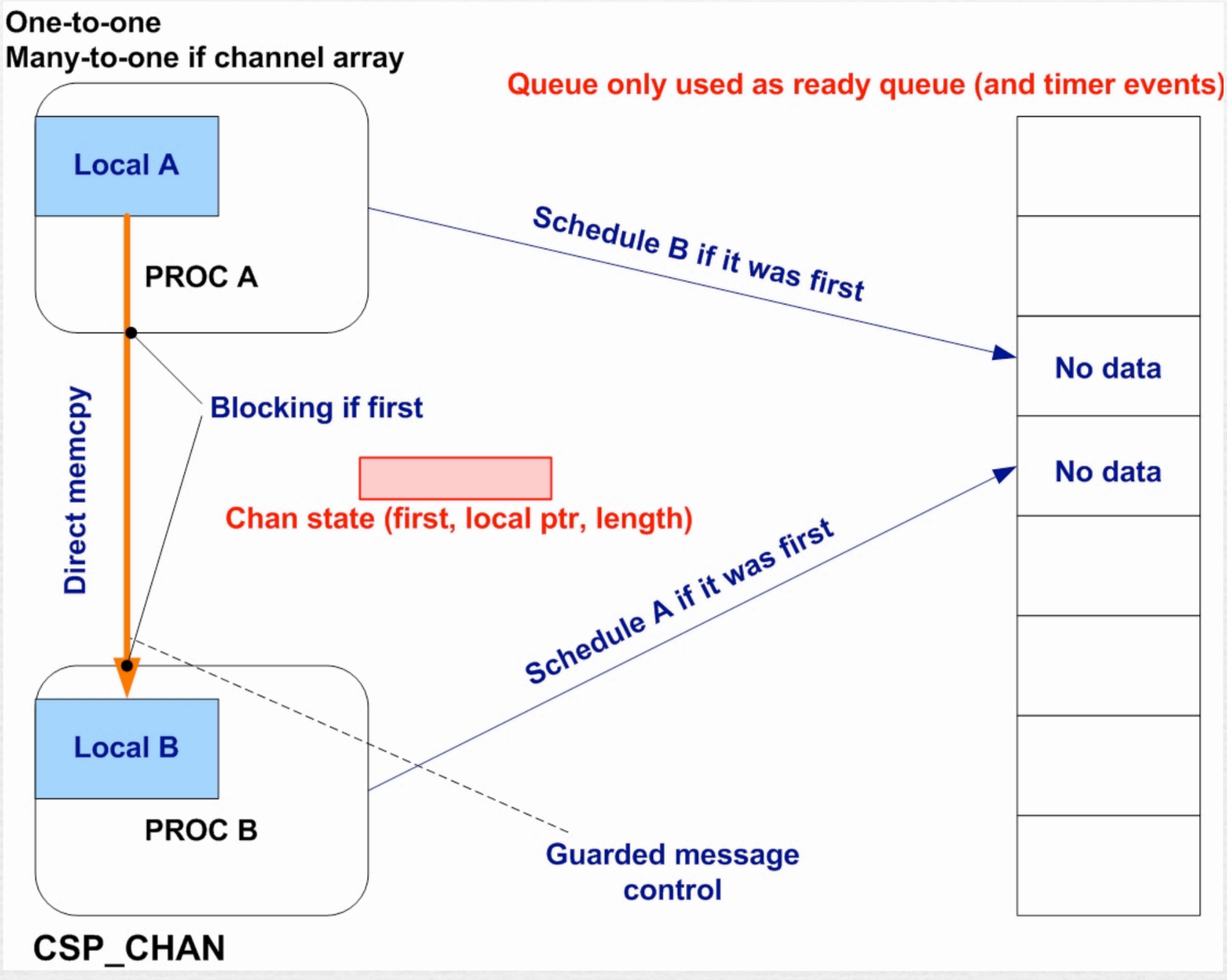

Application SDL processes
Asynch messages
Timers
Sys. timer
Processes
Functions communicating with interrupt
Interrupt handlers, buffers
SDL


Application CSP processes
Channels
Asynch messages
Timers
Sys. timer
Processes
Functions communicating with interrupt
Interrupt handlers, buffers
CSP

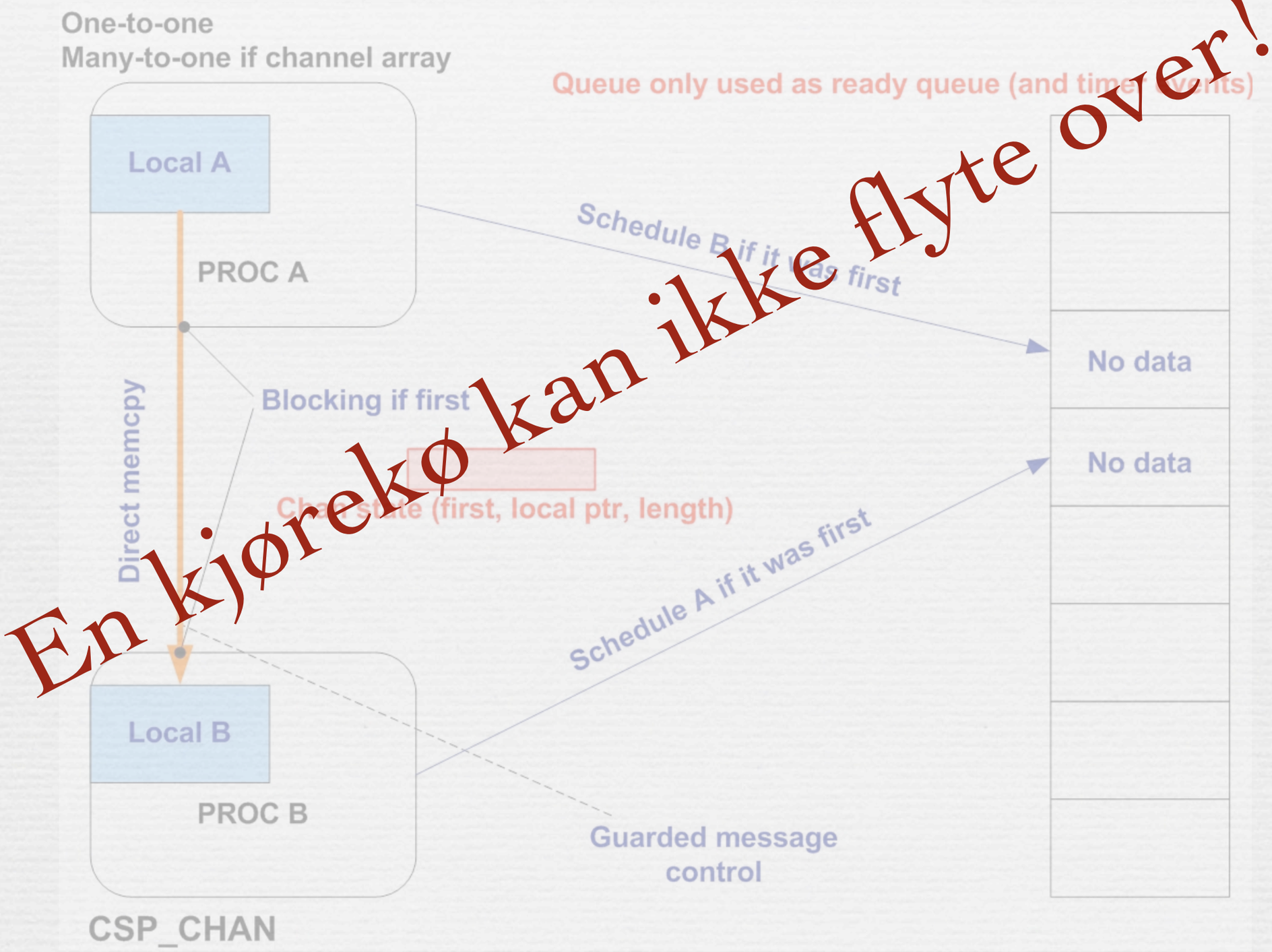
Fra meldingskø...



...til kjørekø


One-to-one
Many-to-one if channel array
Queue only used as ready queue (and timer events)
Local A
PROC A
Schedule B if it was first
No data
No data
Direct memcpy
Blocking if first
Chan state (first, local ptr, length)
Schedule A if it was first
Local B
PROC B
Guarded message control
CSP_CHAN

...til kjørekø



En kjørekø kan ikke flyte over!

Meldingskøoverflyt

- ✓ Overflyt er naturlig
- ✓ Rask produsent treg konsument
- ✓ Det skal håndteres på applikasjonsnivå

Meldingsko

overflyt

- ✓ Overflyt er naturlig
  Men ikke mellom programmer
- ✓ Det skal håndteres på applikasjonsnivå
  Og ikke av operativsystemet
- ✓ Meldinger skal kastes bevisst
  Men kanskje ikke alle
- ✓ Meldingsflyt skal håndteres bevisst
  Så å holde igjen «ved halvgjort» er flott!

"Blokkerende" prosesser

- ✓ Et system som består av blokkerende kommunikasjon får ikke gjort mer enn et system som ikke blokkerer
- ✓ Heller ikke mindre (som nesten alle tror)
- ✓ Jo flere prosesser og dermed "parallell slakkhet" - desto mer greier et synkront system
- ✓ De fleste funksjoner og prosesser er aktive kort tid om gangen ("run to completion")
- ✓ OBS! Synkrone kanaler er ikke implementert i noen busy-poll mekanisme, og polles heller ikke av kjøresystemet. Derfor har de null overhead. De vurderes bare når partene ankommer

2: ChanSched

- ✓ Et system som består av blokkerende kommunikasjon får ikke gjort mer enn et system som ikke blokkerer
- ✓ Heller ikke mindre (som nesten alle tror)
- ✓ Jo flere prosesser og dermed "parallell slakkhet" - desto mer greier et synkront system
- ✓ De fleste funksjoner og prosesser er aktive kort tid om gangen ("run to completion")

ChanSched

- ✓ OBS! Synkrone kanaler er ikke implementert i noen busy-poll mekanisme, og polles heller ikke av kjøresystemet. Derfor har de null overhead. De vurderes bare når partene ankommer

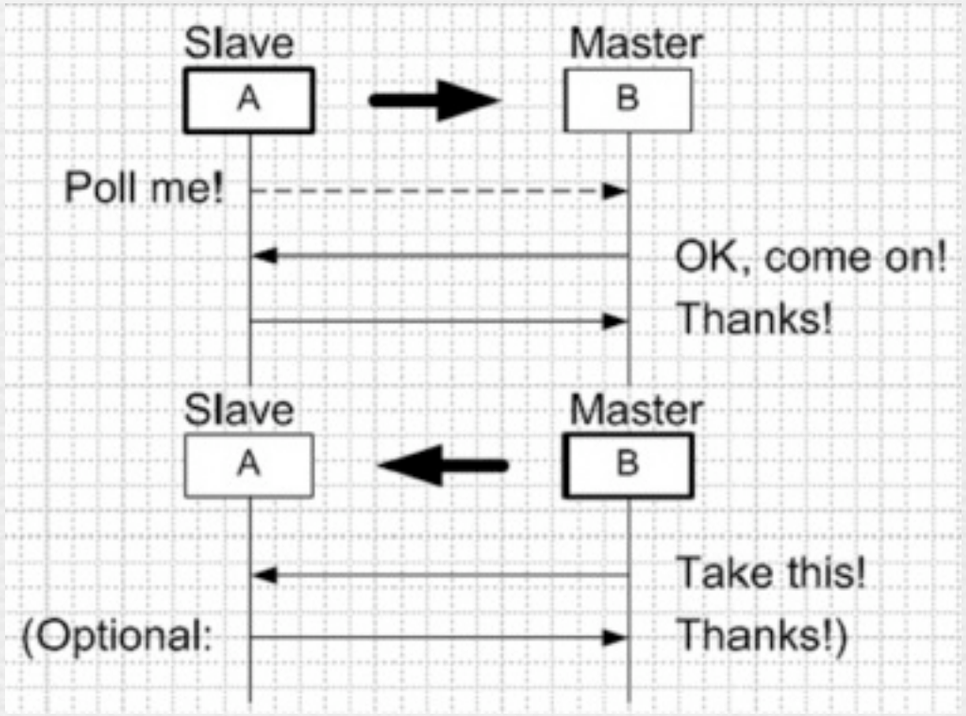
Noen bloggnotater fra 2013

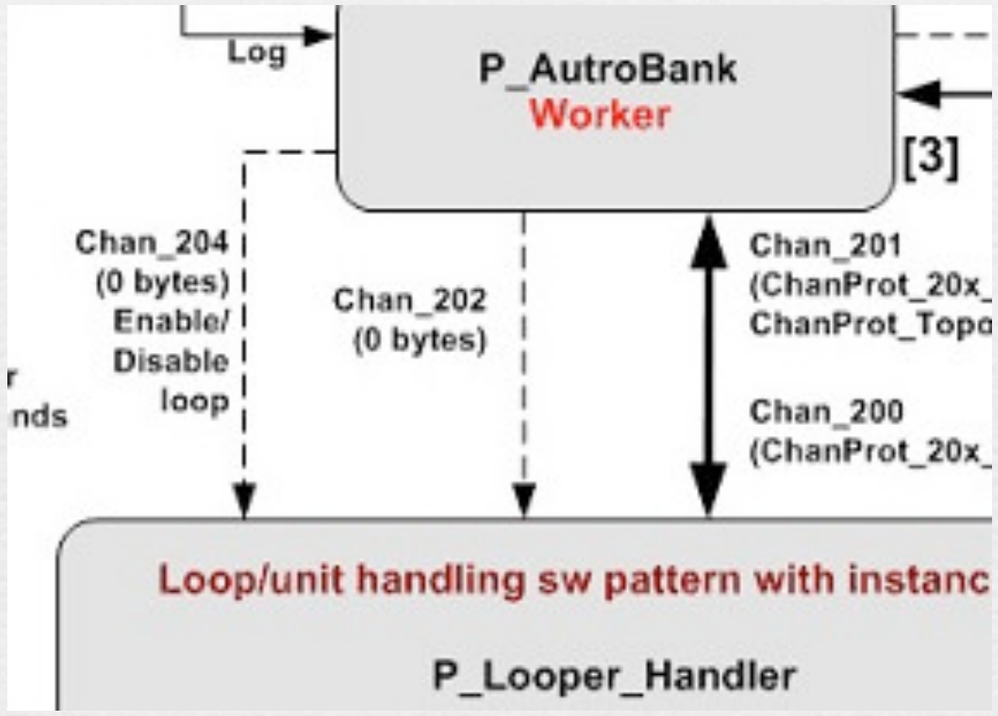
http://www.teigfam.net/oyvind/home/technology/056-some-questions-about-sdl/
«Some questions about SDL»

http://www.teigfam.net/oyvind/home/technology/062-waiting-faster/
“Waiting faster”

Data-komm som alltid funker

- ✓ Mønster som garanterer mot vranglås

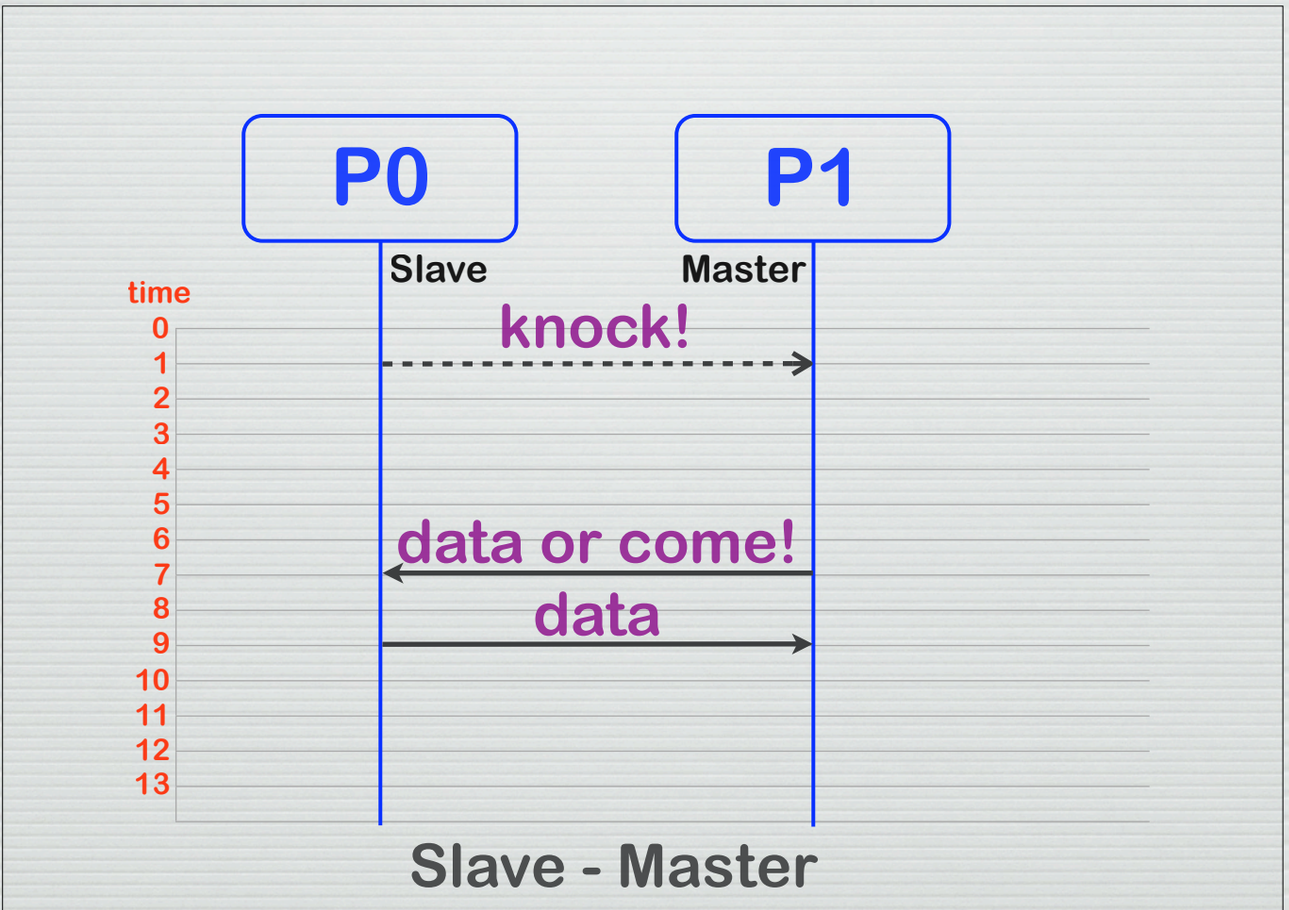


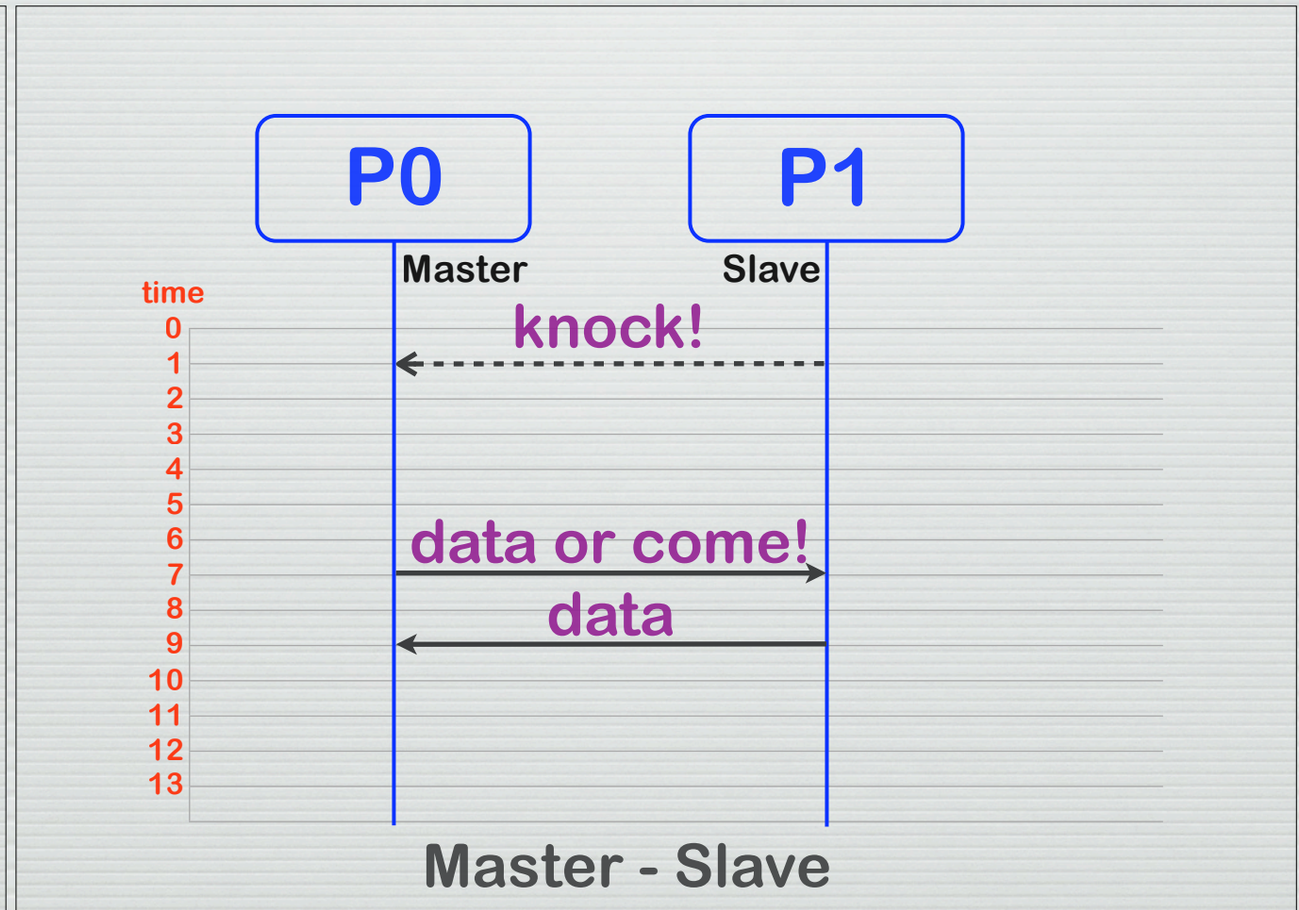


- ✓ «Knock-come» verifisert i Promela/Spin

http://oyvteig.blogspot.com/2009/03/009-knock-come-deadlock-free-pattern.html

Men hvem har hvilken rolle?





Master-Slave eller omvendt?

Seriell OO modellering

- ✓ Her benytter vi UML
- ✓ Vi benytter Rhapsody
  - ✤ Klassediagrammer
  - ✤ Tilstandsmaskiner
  - ✤ Kodegenerering av ramme og skruktur
- ✓ Håndskrevet:
  - ✤ Aksjonskoden
  - ✤ Lim mot resten av systemet
  - ✤ Prosesser via operativsystem så langt

«modellering»

- ✓ ..er nåtidas «ekspertsystem»-ord
- ✓ Men både assembler, C, Java, CSP, occam, Go - og UML er *abstraksjoner*
- ✓ Altså «modeller»
- ✓ Men i varierende grad er de ovenfor «formelle modeller» - dvs. kan være input til et verktøy som gjør verifisering for korrekthet (null til..?)
- ✓ Men disse er: CSP/FDR2, Promela/Spin, LTSA, SPARK, RavenSPARK etc. Rask utvikling, følg med!

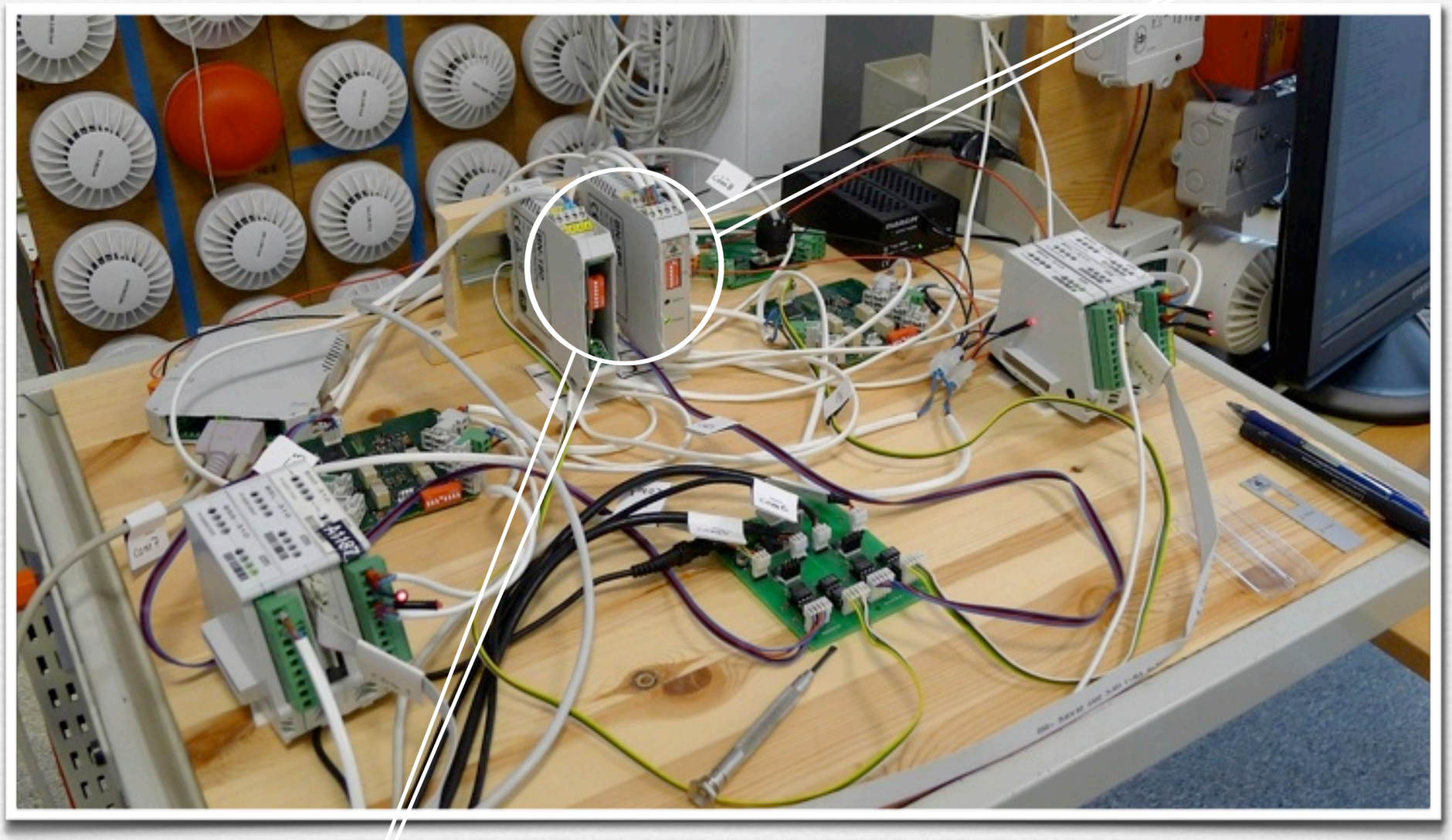
Resten av tida skal jeg vise dere noe jeg har lært i de siste årene

New ALT for Application Timers and Synchronisation Point Scheduling

(Two excerpts from a small channel based scheduler)

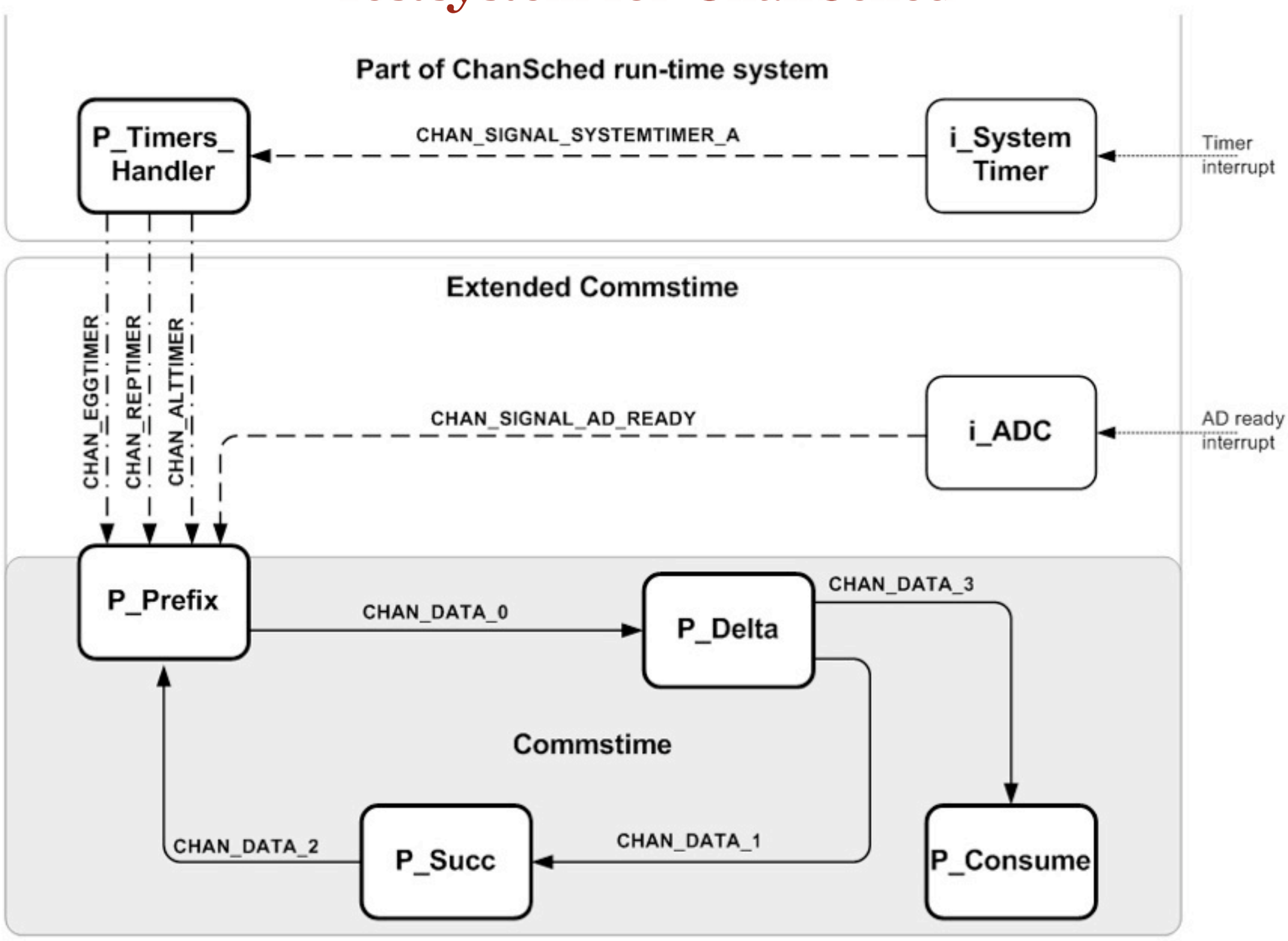
«ChanSched»

Øyvind TEIG and Per Johan VANNEBO
Autronica Fire and Security, Trondheim, Norway



At *Communicating Process Architectures 2009* (CPA-2009), 1-4 November, 2009, Eindhoven, the Netherlands
http://www.wotug.org/cpa2009/

Testsystem for ChanSched



Prosess/dataflyt for utvidet «commstime»

```
void P_Standard_CHAN_CSP (void)
{
  CP_a CP = (CP_a)g_ThisExtPtr; // Application
  switch (CP->State)            // and
                                // communication
                                // state
  {
    case ST_INIT: {/*Init*/ break;}
    case ST_IN:
    {
      CHAN_IN(G_CHAN_IN,CP->Chan_val1);
      CP->State = ST_APPL1;
      break;
    }
    case ST_APPL1:
    {
      // Process val1
      CP->State = ST_OUT;
      break;
    }
    case ST_OUT:
    {
      CHAN_OUT(G_CHAN_OUT,CP->Chan_val1);
      CP->State = ST_IN;
      break;
    }
  }
}

void P_libcsp2 (Channel *in, Channel *out)
{
  int val3;
  for(;;)
  {
    ChanInInt (in, &val3);
    // Process val3
    ChanOutInt (out, val3);
  }
}
```

```
void P_Extended_ChanSched (void)
{
  CP_a CP = (CP_a)g_ThisExtPtr; // Application
  // Init here                  // state only
  while (TRUE)
  {
    switch (CP->State)
    {
      case ST_MAIN:
      {
        CHAN_IN(G_CHAN_IN,CP->Chan_val2);

        // Process val2

        CHAN_OUT(G_CHAN_OUT,CP->Chan_val2);
        CP->State = ST_MAIN; // option1
        break;
      }
    }
  }
}

PROC P_occam (CHAN OF INT in, out)

  WHILE TRUE
  INT val4:
    SEQ
      in ? val4
      -- Process val4
      out ! val4

:
```

«En scheduler/ kjøresystem/ operativsystem/ er er ikke så usynlig som jeg trodde»

```
void P_Standard_CHAN_CSP (void)
{
  CP_a CP = (CP_a)g_ThisExtPtr; // Application
  switch (CP->State)            // and
                                // communication
                                // state
  {
    case ST_INIT: {/*Init*/ break;}
    case ST_IN:
    {
      CHAN_IN(G_CHAN_IN,CP->Chan_val1);
      CP->State = ST_APPL1;
      break;
    }
    case ST_APPL1:
    {
      // Process val1
      CP->State = ST_OUT;
      break;
    }
    case ST_OUT:
    {
      CHAN_OUT(G_CHAN_OUT,CP->Chan_val1);
      CP->State = ST_IN;
      break;
    }

  }
}

void P_libcsp2 (Channel *in, Channel *out)
{
  int val3;
  for(;;)
  {
    ChanInInt (in, &val3);
    // Process val3
    ChanOutInt (out, val3);
  }
}
```

Kjøreystem oppå annet kjøresystem

```
void P_Extended_ChanSched (void)
{
  CP_a CP = (CP_a)g_ThisExtPtr; // Application
  // Init here                  // state only
  while (TRUE)
  {
    switch (CP->State)
    {
      case ST_MAIN:
      {
        CHAN_IN(G_CHAN_IN,CP->Chan_val2);
        // Process val2
        CHAN_OUT(G_CHAN_OUT,CP->Chan_val2);
        CP->State = ST_MAIN; // option1
        break;
      }
    }
  }
}

PROC P_occam (CHAN OF INT in, out)

  WHILE TRUE
  INT val4:
    SEQ
      in ? val4
      -- Process val4
      out ! val4

:
```

Kjøreystem alene gir eget prosessbegrep

```
void P_Standard_CHAN_CSP (void)
{
  CP_a CP = (CP_a)g_ThisExtPtr; // Application
  switch (CP->State)            // and
                                // communication
                                // state
  {
    case ST_INIT: {/*Init*/ break;}
    case ST_IN:
    {
      CHAN_IN(G_CHAN_IN,CP->Chan_val1);
      CP->State = ST_APPL1;
      break;
    }
    case ST_APPL1:
    {
      // Process val1
      CP->State = ST_OUT;
      break;
    }
    case ST_OUT:
    {
      CHAN_OUT(G_CHAN_OUT,CP->Chan_val1);
      CP->State = ST_IN;
      break;
    }

  }
}

void P_libcsp2 (Channel *in, Channel *out)
{
  int val3;
  for(;;)
  {
    ChanInInt (in, &val3);
    // Process val3
    ChanOutInt (out, val3);
  }
}
```

```
void P_Extended_ChanSched (void)
{
  CP_a CP = (CP_a)g_ThisExtPtr; // Application
  // Init here                  // state only
  while (TRUE)
  {
    switch (CP->State)
    {
      case ST_MAIN:
      {
        CHAN_IN(G_CHAN_IN,CP->Chan_val2);
        // Process val2
        CHAN_OUT(G_CHAN_OUT,CP->Chan_val2);
        CP->State = ST_MAIN; // option1
        break;
      }
    }
  }
}

PROC P_occam (CHAN OF INT in, out)

  WHILE TRUE
  INT val4:
    SEQ
      in ? val4
      -- Process val4
      out ! val4

:
```

En typisk ChanSched prosess

```
1.  Void P_Prefix (void)                          // extended “Prefix”
2.  {
3.    Prefix_CP_a CP = (Prefix_CP_a)g_CP; // get process Context from Scheduler
4.    PROCTOR_PREFIX()                            // jump table (see Section 2)
5.    ...  some initialisation
6.    SET_EGGTIMER (CHAN_EGGTIMER, LED_Timeout_Tick);
7.    SET_REPTIMER (CHAN_REPTIMER, ADC_TIME_TICKS);
8.    CHAN_OUT (CHAN_DATA_0, Data_0);  // first output
9.    while (TRUE)
10.   {
11.     ALT();                                    // this is the needed ”PRI_ALT”
12.       ALT_EGGREPTIMER_IN (CHAN_EGGTIMER);
13.       ALT_EGGREPTIMER_IN (CHAN_REPTIMER);
14.       ALT_SIGNAL_CHAN_IN (CHAN_SIGNAL_AD_READY);
15.       ALT_CHAN_IN        (CHAN_DATA_2, Data_2);
16.       ALT_ALTTIMER_IN    (CHAN_ALTTIMER, TIME_TICKS_100_MSECS);
17.     ALT_END();
18.     switch (g_ThisChannelId)
19.     {
20.       ...  process the guard that has been taken, e.g. CHAN_DATA_2
21.       CHAN_OUT (CHAN_DATA_0, Data_0);
22.     };
23.   }
24. }
```

"Fjernet" siden i første utgave

```
1.  Void P_Prefix (void)                          // extended “Prefix”
2.  {
3.    Prefix_CP_a CP = (Prefix_CP_a)g_CP; // get process Context from Scheduler
4.    PROCTOR_PREFIX()                            // jump table (see Section 2)
5.    ...  some initialisation
6.    SET_EGGTIMER (CHAN_EGGTIMER, CP->LED_Timeout_Tick);
7.    SET_REPTIMER (CHAN_REPTIMER, ADC_TIME_TICKS);
8.    CHAN_OUT (CHAN_DATA_0, &CP->Data_0, sizeof(CP->Data_0));  // first output
9.    while (TRUE)
10.   {
11.     ALT();                                    // this is the needed ”PRI_ALT”
12.       ALT_EGGREPTIMER_IN (CHAN_EGGTIMER);
13.       ALT_EGGREPTIMER_IN (CHAN_REPTIMER);
14.       ALT_SIGNAL_CHAN_IN (CHAN_SIGNAL_AD_READY);
15.       ALT_CHAN_IN        (CHAN_DATA_2, &CP->Data_2, sizeof (CP->Data_2));
16.       ALT_ALTTIMER_IN    (CHAN_ALTTIMER, TIME_TICKS_100_MSECS);
17.     ALT_END();
18.     switch (g_ThisChannelId)
19.     {
20.       ...  process the guard that has been taken, e.g. CHAN_DATA_2
21.       CHAN_OUT (CHAN_DATA_0, &CP->Data_0, sizeof (CP->Data_0));
22.     };
23.   }
24. }
```

Nesten usynlig hopp tilbake til der man sist blokkerte

Se http://www.teigfam.net/oyvind/pub/pub_details.html#NewALT

```
64. #define SCHEDULE_AT goto

66. #define CAT(a,b,c,d,e) a##b##c##d##e // Concatenate to f.ex. "SYNCH_8_L"

68. #define SYNCH_LABEL(a,b,c,d,e) CAT(a,b,c,d,e) // Label for Proctor-table

70. #define PROC_DESCHEDULE_AND_LABEL() \
71.         CP->LineNo = __LINE__; \
72.         return; \
73.         SYNCH_LABEL(SYNCH,_,__LINE__,_,L):

75. #define CHAN_OUT(chan,dataptr,len) \
76.         if (ChanSched_ChanOut(chan,dataptr,len) == FALSE) \
77.         { \
78.             PROC_DESCHEDULE_AND_LABEL(); \
79.         } \
80.         g_ThisAltTaken = FALSE
```

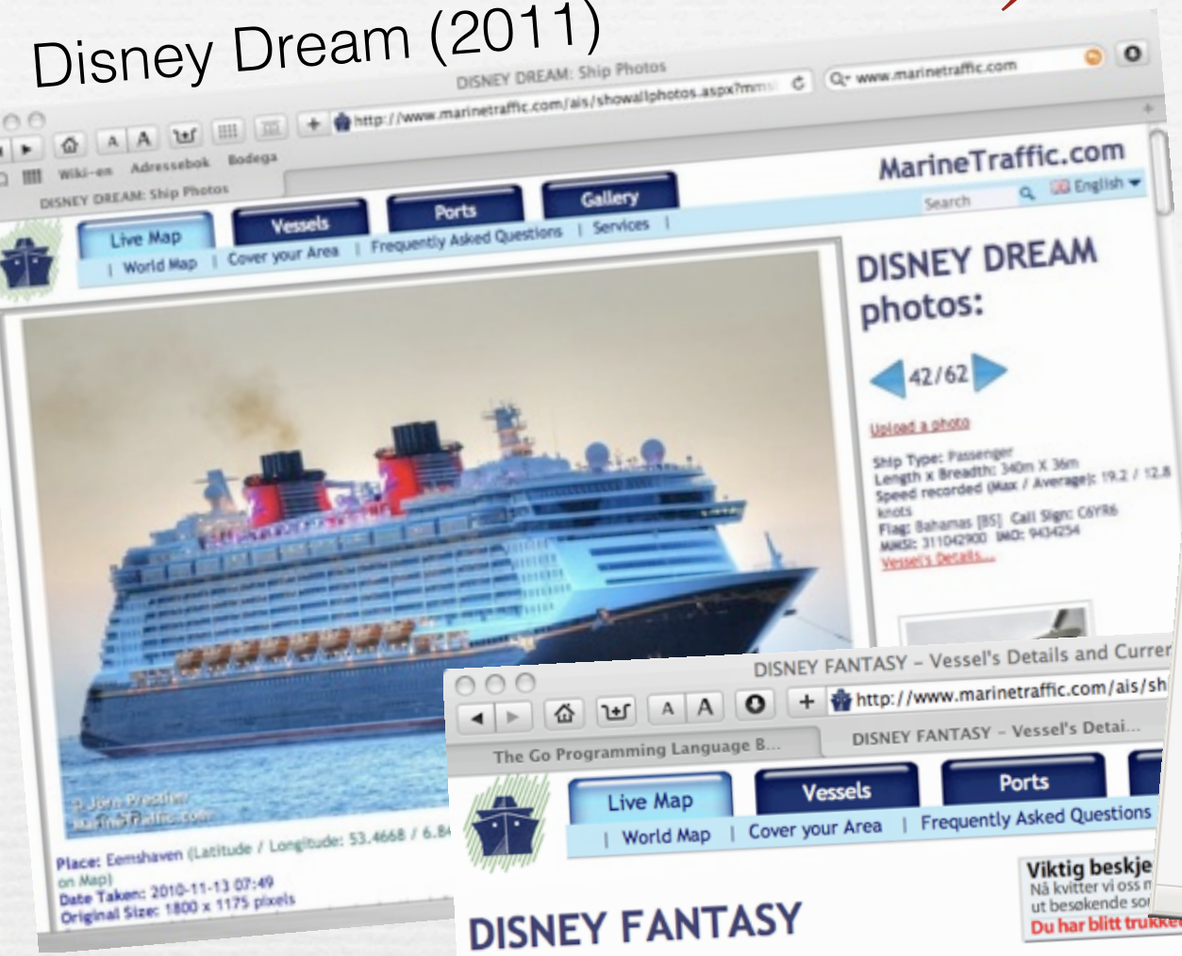
```
81. #define PROCTOR_PREFIX()\
82.         switch (CP->LineNo)\
83.         {\
84.             case 0: break;\
85.             case 8: SCHEDULE_AT SYNCH_8_L;\
86.             case 17: SCHEDULE_AT SYNCH_17_L;\
87.             case 21: SCHEDULE_AT SYNCH_21_L;\
88.             DEFAULT_EXIT\
89.         }
```

Bare denne er synlig, ref to sider siden

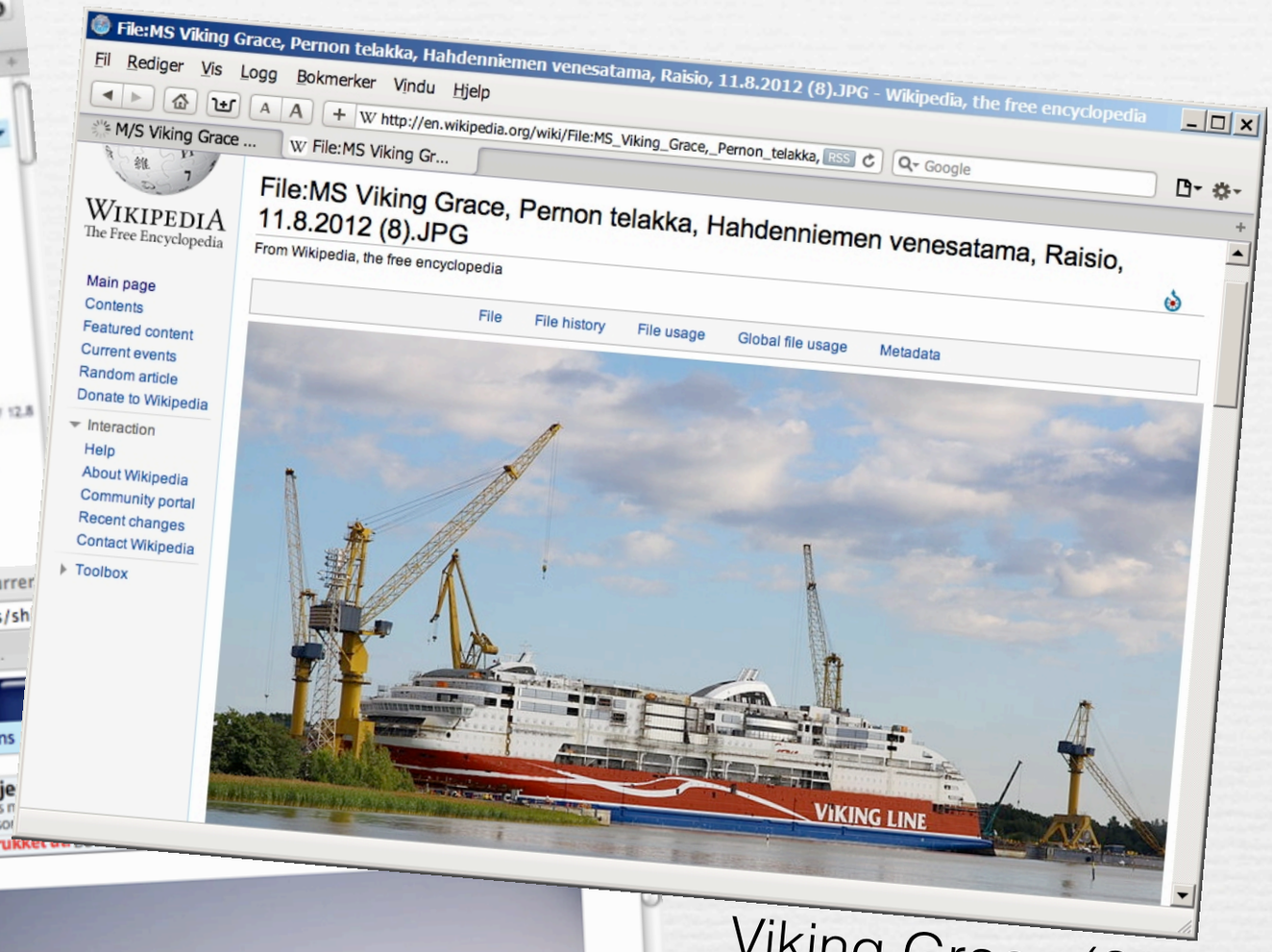
Genereres av verktøy og ligger i egen .h fil

Dette kjører ombord på

Disney Dream (2011)



Disney Fantasy (2012)

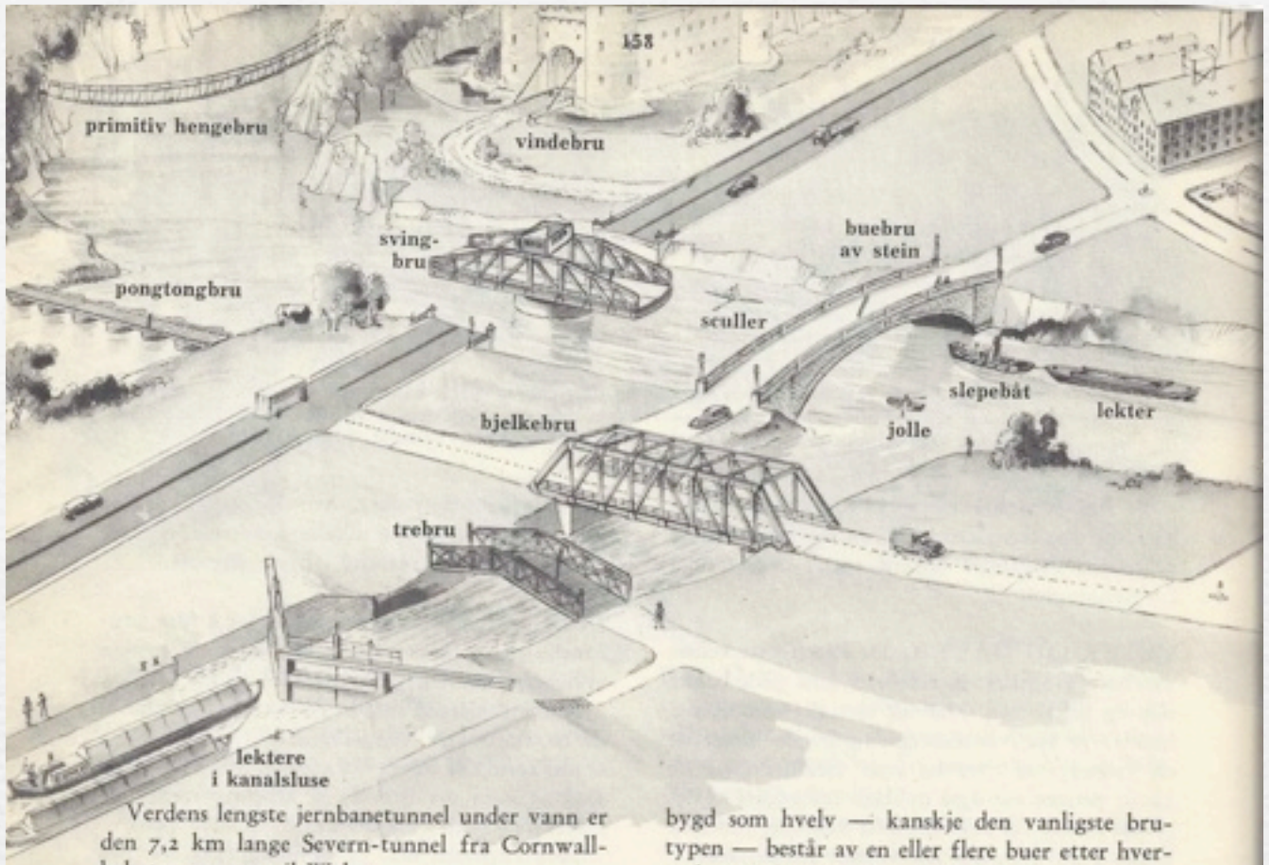


Viking Grace (2013)

++

AutroKeeper
(BNA-180)
Safe Return to Port
Dual Safety

AutroKeeper: patentert 329859 i Norge, PCT/NO2009/000319 utenlands (granted as #2353255)



Verdens lengste jernbanetunnel under vann er den 7,2 km lange Severn-tunnel fra Cornwall-halvøya og over til Wales.

Verdens lengste tunnel for biltrafikk — og den bredeste tunnel som noen gang er bygd — er Merseytunnelen, mellom Liverpool og Birkenhead. Den er 3,4 km lang, 13,5 meter i diameter og har fire kjørebaner, to i hver retning — en for langsom og en for rask trafikk.

BRUER. En bygger bruer når en vil føre en vei eller en jernbanelinje over en elv, et sund, en kanal, en dal eller over en annen vei eller jernbanelinje.

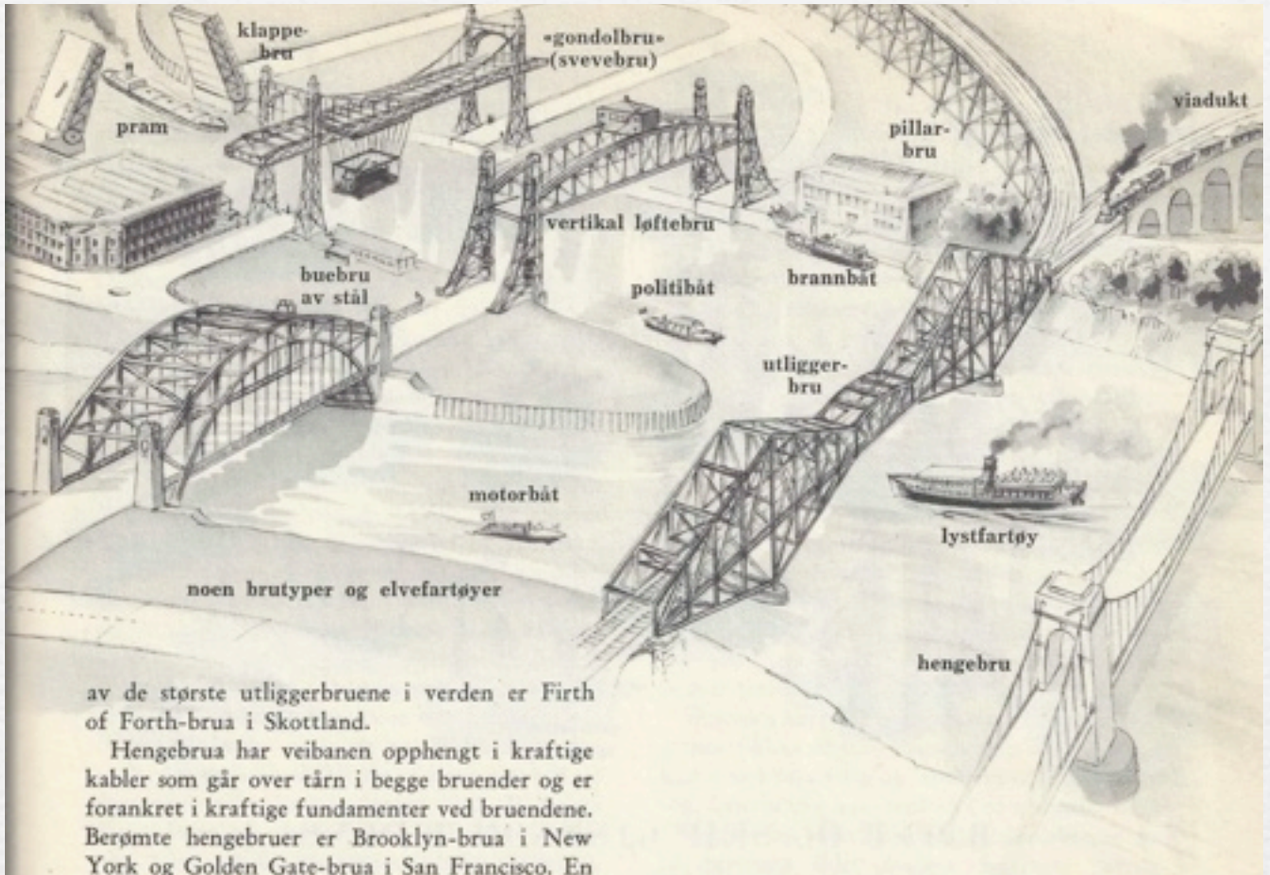
Hvilken brutype en velger, avhenger av hvor lang brua må være. Bjelkebrua brukes bare når det gjelder et nokså kort spenn. Bruer som er bygd som hvelv — kanskje den vanligste brutypen — består av en eller flere buer etter hverandre som bærer veibanen.. En bruker stein, murstein eller betong som byggemateriale. Mange av de store moderne buebruene er bygd av stål. Den største i verden er brua over havnebassenget i Sydney i Australia. En annen vakker buebru av stål går over Sambesifloden i Afrika, og så tett inntil Victoria-fallene at togene som kjører over, ofte blir oversprøytet av dusjen fra fossen.

Når det gjelder svære spenn, bruker en ofte den såkalte utliggerbru. Den hviler på en rekke pilarer, og de enkelte bruspenn rager dels ut over pilarene, dels kan de være opphengt mellom de utstikkende brudelene. Stabiliteten av slike bruer er basert på en viktig tilpassing i lengde og vekt mellom utliggerne og de innhengte spennene. En av de største utliggerbruene i verden er Firth of Forth-brua i Skottland.

Hengebrua har veibanen opphengt i kraftige kabler som går over tårn i begge bruender og er forankret i kraftige fundamenter ved bruendene. Berømte hengebruer er Brooklyn-brua i New York og Golden Gate-brua i San Francisco. En bruker gjerne hengebruer når en skal føre en vei over svære vannflater, hvor det er umulig å bygge brupilarer ute i vannet. De moderne hengebruene av stål og betong har ikke stor likhet med de enkle hengebruene av tau eller flettede grener, slik en kan se dem ennå i verdens fjerne avkroker.

Over vann der det er skipstrafikk, må vi — hvis det er umulig å bygge et høyt bruspenn — bruke en bevegelig bru. Tower Bridge i London er en bru av denne typen. Den er en klaffebru som folder veibanen opp som to klaffer, akkurat som bladene på en foldekniv, når skip skal passere under. Svingbruene er vribare om en tapp som hviler på et fundament midt under brua. Løftebrua er en sjeldnere type. Den kan heves loddrett til værs. Enda sjeldnere er typen med en hengende plattform under en høytliggende bærekonstruksjon. Her transporteres folk og kjøretøyer fram og tilbake i en slags gondol. Den er festet til en tralle som løper på skinner oppe på bærebjelkene.

Pongtongbruer består av en veibane som er lagt over flytende metalltanker. (Før brukte man båter.) Denne brutypen brukes ofte i krig, fordi den fort kan slåes over en elv og like fort rives.



noen brutyper og elvefartøyer

KANALER OG SEILBARE ELVER.

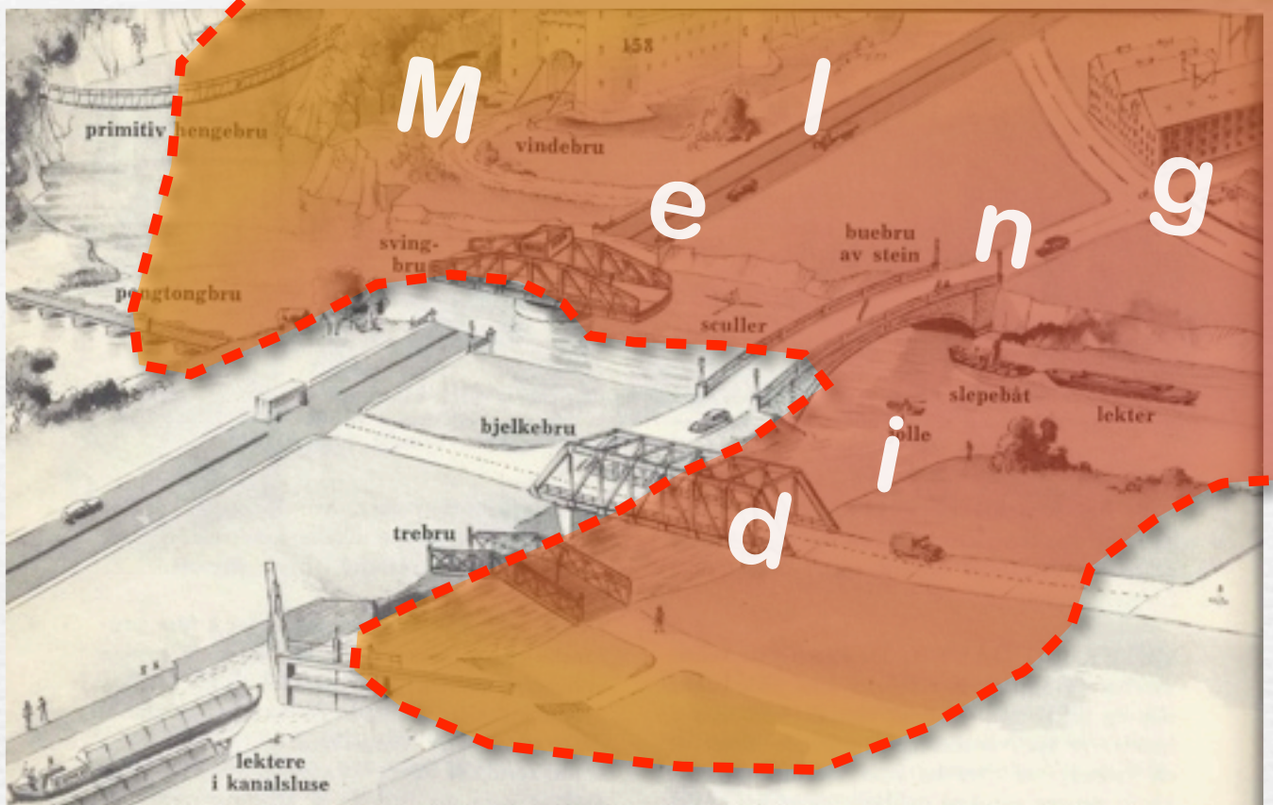
Før det ble bygd veier, foregikk reisene over land helst på elvene, og da var vassdragene de viktigste forbindelsesveiene. I våre dager spiller slike seilbare elver fremdeles stor rolle, særlig for godstrafikken, og mange havner ligger ved munningen av store elver, f. eks. Fredrikstad, Göteborg, Hamburg, London og New York.

Innlandstransport til vanns har kunnet økes atskillig mange steder i verden ved at det er blitt bygd kanaler. Noen kanaler har samme nivå hele strekningen, mens andre klatrer opp og ned. Fartøyene blir hevet og senket ved hjelp av sluser.

En sluse er et kort stykke av kanalen som har vanntette porter i begge ender. En båt som er på vei oppover en kanal, og som altså skal heves til et høyere nivå, går inn i slusen, og så stenges portene. Deretter åpnes sluselukene i den øvre porten, og vannet strømmer inn fra nivået som ligger høyere. Nå løftes båten av det stigende vannet til nivået er det samme som i slusen ovenfor. Så åpnes den øvre porten, og båten kan seile videre. Når en båt skal senkes, slippes vannet ut av slusen. Kanalene må ofte mudres opp så de er dype nok for fartøyene.

i kanalslusen slippes vannet inn så vannspeilet stiger og løfter lekteren, eller det slippes ut så lekteren senkes og kan gå nedover til lavere nivå

"Verden omkring oss", 1955 ("Odhams Encyclopedia for Children")



Verdens lengste jernbanetunnel under vann er den 7,2 km lange Severn-tunnel fra Cornwall-halvøya og over til Wales.

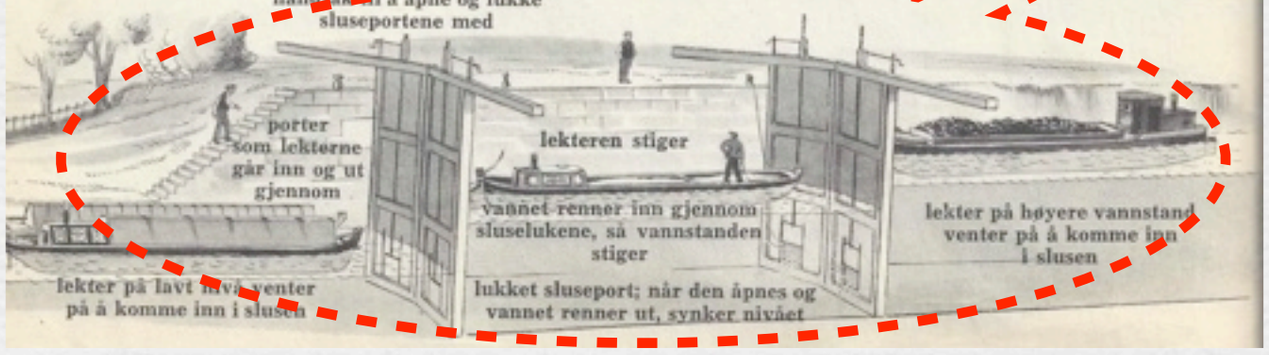
Verdens lengste tunnel for biltrafikk — og den bredeste tunnel som noen gang er bygd — er Merseytunnelen, mellom Liverpool og Birkenhead. Den er 3,4 km lang, 13,5 meter i diameter og har fire kjørebaner, to i hver retning — en for langsom og en for rask trafikk.

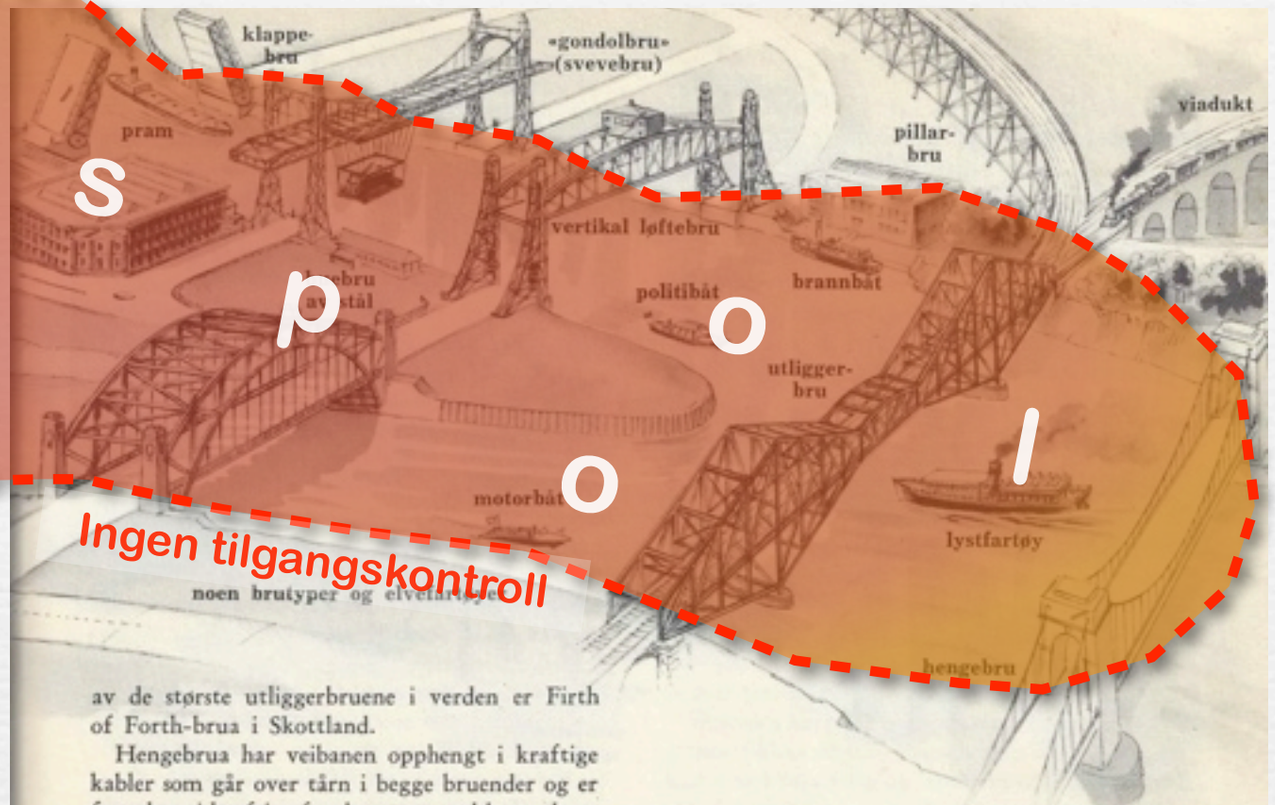
BRUER. En bygger bruer når en vil føre en vei eller en jernbanelinje over en elv, et sund, en kanal, en dal eller over en annen vei eller jernbanelinje.

Hvilken brutype en velger, avhenger av hvor lang brua må være. Bjelkebrua brukes bare når det gjelder et nokså kort spenn. Bruer som er bygd som hvelv — kanskje den vanligste brutypen — består av en eller flere buer etter hverandre som bærer veibanen.. En bruker stein, murstein eller betong som byggemateriale. Mange av de store moderne buebruene er bygd av stål. Den største i verden er brua over havnebassenget i Sydney i Australia. En annen vakker buebru av stål går over Sambesifloden i Afrika, og så tett inntil Victoria-fallene at togene som kjører over, ofte blir overspr[illegible] av dusjen fra fossen.

Når det gjelder svære spenn, bruker en ofte den såkalte utliggerbru. Den hviler på en rekke pilarer, og de enkelte bruspenn rager dels ut over pilarene, dels kan de være opphengt mellom de utstikkende brudelene. Stabiliteten av slike bruer er basert på en viktig tilpassing i lengde og vekt mellom utliggerne og de innhengte spennene. En

i kanalslusen slippes vannet inn så vannspeilet stiger og løfter lekteren, eller det slippes ut så lekteren senkes og kan gå nedover til lavere nivå





noen brutyper og elvefartøyer

av de største utliggerbruene i verden er Firth of Forth-brua i Skottland.

Hengebrua har veibanen opphengt i kraftige kabler som går over tårn i begge bruender og er forankret i kraftige fundamenter ved bruendene. Berømte hengebruer er Brooklyn-brua i New York og Golden Gate-brua i San Francisco. En bruker gjerne hengebruer når en skal føre en vei over svære vannflater, hvor det er umulig å bygge brupilarer ute i vannet. De moderne hengebruene av stål og betong har ikke stor likhet med de enkle hengebruene av tau eller flettede grener, slik en kan se dem ennå i verdens fjerne avkroker.

Over vann der det er skipstrafikk, må vi — hvis det er umulig å bygge et høyt bruspenn — bruke en bevegelig bru. Tower Bridge i London er en bru av denne typen. Den er en klaffebru som folder veibanen opp som to klaffer, akkurat som bladene på en foldekniv, når skip skal passere under. Svingbruene er vribare om en tapp som hviler på et fundament midt under brua. Løftebrua er en sjeldnere type. Den kan heves loddrett til værs. Enda sjeldnere er typen med en hengende plattform under en høytliggende bærekonstruksjon. Her transporteres folk og kjøretøyer fram og tilbake i en slags gondol. Den er festet til en tralle som løper på skinner oppe på bærebjelkene.

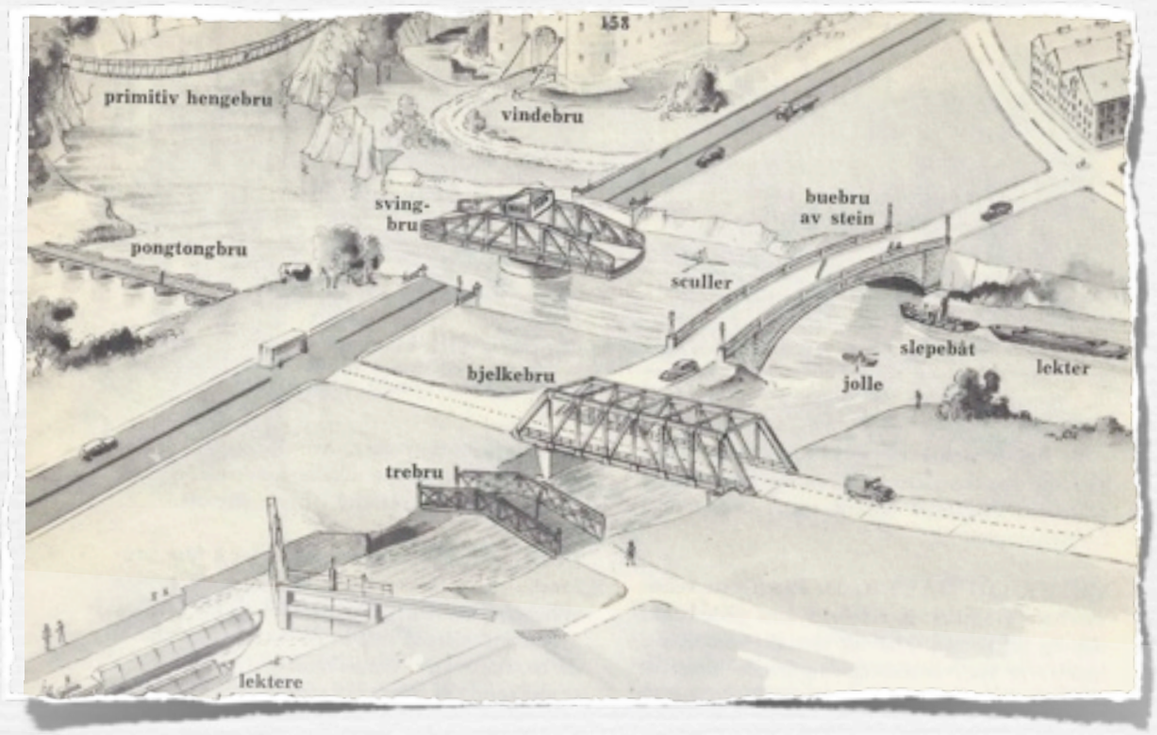
Pongtongbruer består av en veibane som er lagt over flytende metalltanker. (Før brukte man båter.) Denne brutypen brukes ofte i krig, fordi den fort kan slåes over en elv og like fort rives.

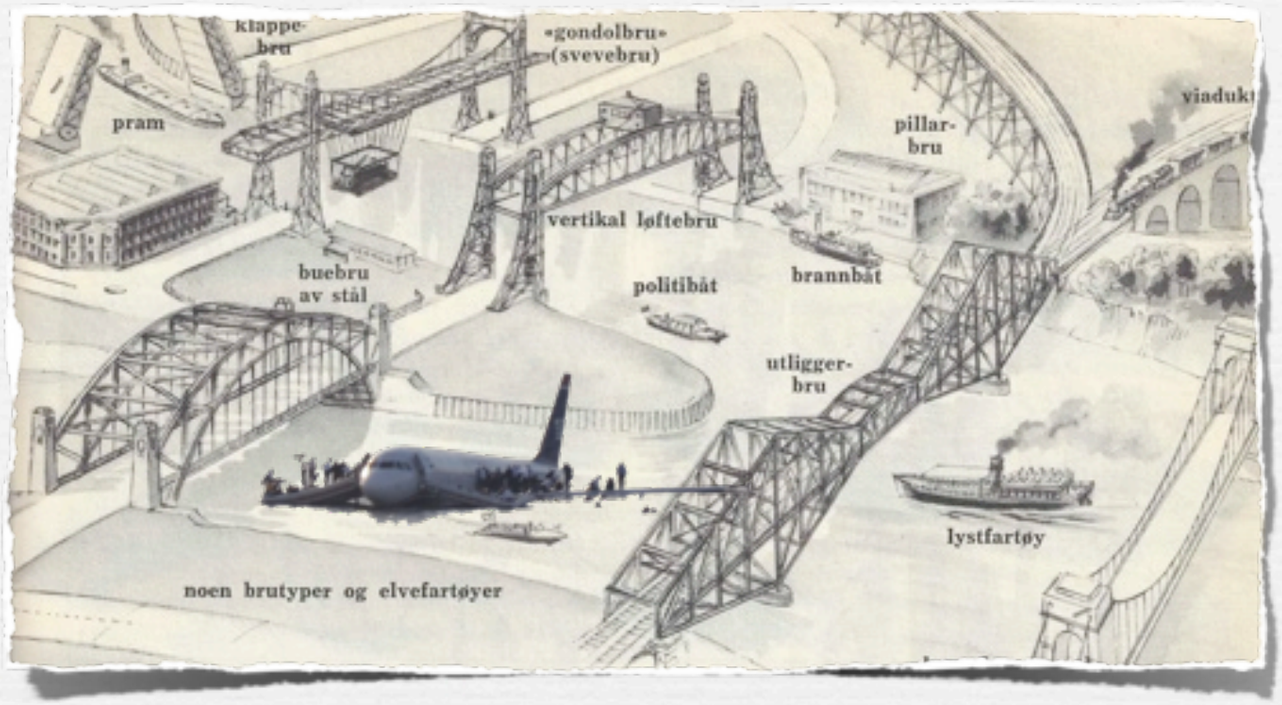
KANALER OG SEILBARE ELVER.

Før det ble bygd veier, foregikk reisene over land helst på elvene, og da var vassdragene de viktigste forbindelsesveiene. I våre dager spiller slike seilbare elver fremdeles stor rolle, særlig for godstrafikken, og mange havner ligger ved munningen av store elver, f. eks. Fredrikstad, Göteborg, Hamburg, London og New York.

Innlandstransport til vanns har kunnet økes atskillig mange steder i verden ved at det er blitt bygd kanaler. Noen kanaler har samme nivå hele strekningen, mens andre klatrer opp og ned. Fartøyene blir hevet og senket ved hjelp av sluser.

En sluse er et kort stykke av kanalen som har vanntette porter i begge ender. En båt som er på vei oppover en kanal, og som altså skal heves til et høyere nivå, går inn i slusen, og så stenges portene. Deretter åpnes sluselukene i den øvre porten, og vannet strømmer inn fra nivået som ligger høyere. Nå løftes båten av det stigende vannet til nivået er det samme som i slusen ovenfor. Så åpnes den øvre porten, og båten kan seile videre. Når en båt skal senkes, slippes vannet ut av slusen. Kanalene må ofte mudres opp så de er dype nok for fartøyene.



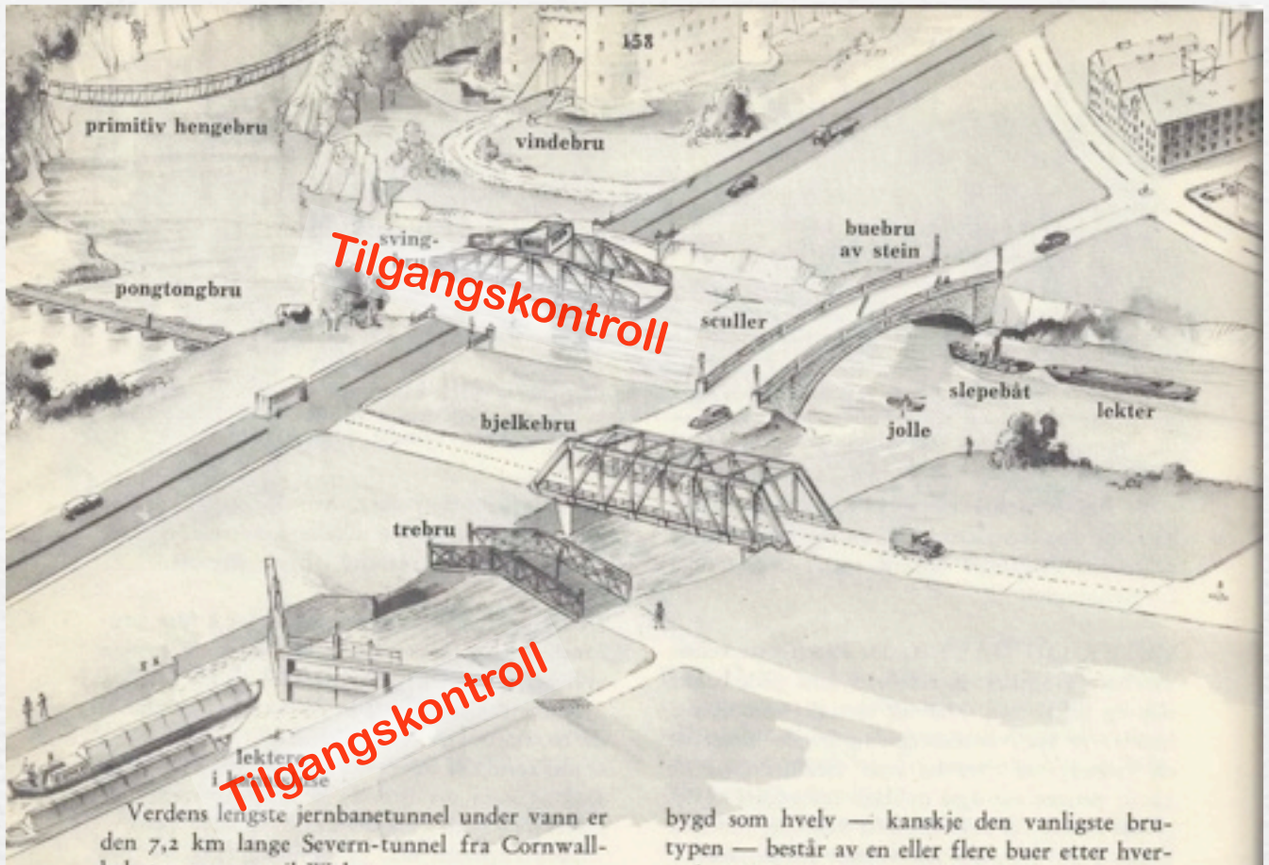


Om det sendes en eneste melding for mye så kræsjer systemet og alle meldingene mistes...

..men da øker man meldingspool-størrelse ved neste versjon til «litt mer»

..for å analysere en slik situasjon er nesten umulig

(polemikken viser et viktig punkt!)



Verdens lengste jernbanetunnel under vann er den 7,2 km lange Severn-tunnel fra Cornwall-halvøya og over til Wales.

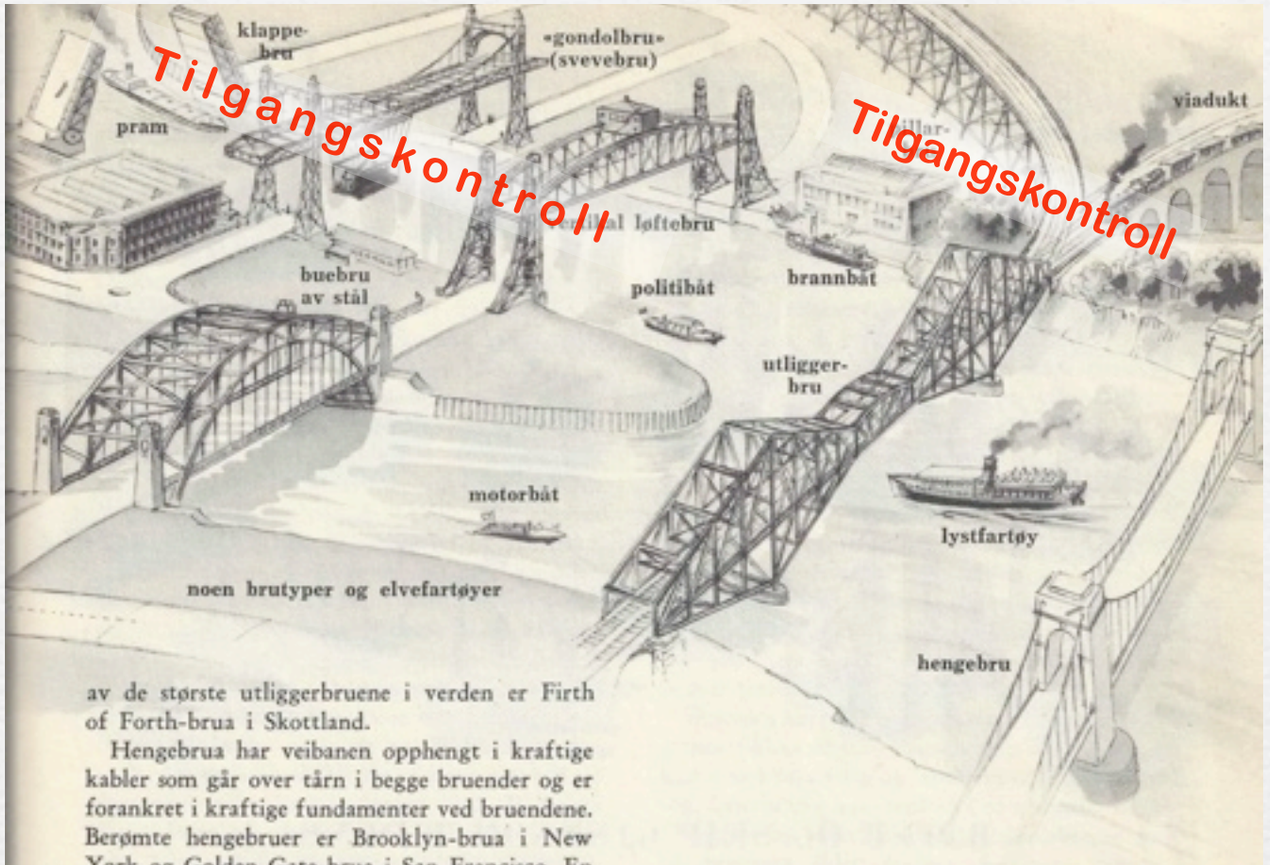
Verdens lengste tunnel for biltrafikk — og den bredeste tunnel som noen gang er bygd — er Merseytunnelen, mellom Liverpool og Birkenhead. Den er 3,4 km lang, 13,5 meter i diameter og har fire kjørebaner, to i hver retning — en for langsom og en for rask trafikk.

BRUER. En bygger bruer når en vil føre en vei eller en jernbanelinje over en elv, et sund, en kanal, en dal eller over en annen vei eller jernbanelinje.

Hvilken brutype en velger, avhenger av hvor lang brua må være. Bjelkebrua brukes bare når det gjelder et nokså kort spenn. Bruer som er bygd som hvelv — kanskje den vanligste brutypen — består av en eller flere buer etter hverandre som bærer veibanen.. En bruker stein, murstein eller betong som byggemateriale. Mange av de store moderne buebruene er bygd av stål. Den største i verden er brua over havnebassenget i Sydney i Australia. En annen vakker buebru av stål går over Sambesifloden i Afrika, og så tett inntil Victoria-fallene at togene som kjører over, ofte blir oversprøytet av dusjen fra fossen.

Når det gjelder svære spenn, bruker en ofte den såkalte utliggerbru. Den hviler på en rekke pilarer, og de enkelte bruspenn rager dels ut over pilarene, dels kan de være opphengt mellom de utstikkende brudelene. Stabiliteten av slike bruer er basert på en viktig tilpassing i lengde og vekt mellom utliggerne og de innhengte spennene. En

i kanalslusen slippes vannet inn så vannspeilet stiger og løfter lekteren, eller det slippes ut så lekteren senkes og kan gå nedover til lavere nivå



noen brutyper og elvefartøyer

av de største utliggerbruene i verden er Firth of Forth-brua i Skottland.

Hengebrua har veibanen opphengt i kraftige kabler som går over tårn i begge bruender og er forankret i kraftige fundamenter ved bruendene. Berømte hengebruer er Brooklyn-brua i New York og Golden Gate-brua i San Francisco. En bruker gjerne hengebruer når en skal føre en vei over svære vannflater, hvor det er umulig å bygge brupilarer ute i vannet. De moderne hengebruene av stål og betong har ikke stor likhet med de enkle hengebruene av tau eller flettede grener, slik en kan se dem ennå i verdens fjerne avkroker.

Over vann der det er skipstrafikk, må vi — hvis det er umulig å bygge et høyt bruspenn — bruke en bevegelig bru. Tower Bridge i London er en bru av denne typen. Den er en klaffebru som folder veibanen opp som to klaffer, akkurat som bladene på en foldekniv, når skip skal passere under. Svingbruene er vribare om en tapp som hviler på et fundament midt under brua. Løftebrua er en sjeldnere type. Den kan heves loddrett til værs. Enda sjeldnere er typen med en hengende plattform under en høytliggende bærekonstruksjon. Her transporteres folk og kjøretøyer fram og tilbake i en slags gondol. Den er festet til en tralle som løper på skinner oppe på bærebjelkene.

Pongtongbruer består av en veibane som er lagt over flytende metalltanker. (Før brukte man båter.) Denne brutypen brukes ofte i krig, fordi den fort kan slåes over en elv og like fort rives.

KANALER OG SEILBARE ELVER.

Før det ble bygd veier, foregikk reisene over land helst på elvene, og da var vassdragene de viktigste forbindelsesveiene. I våre dager spiller slike seilbare elver fremdeles stor rolle, særlig for godstrafikken, og mange havner ligger ved munningen av store elver, f. eks. Fredrikstad, Göteborg, Hamburg, London og New York.

Innlandstransport til vanns har kunnet økes atskillig mange steder i verden ved at det er blitt bygd kanaler. Noen kanaler har samme nivå hele strekningen, mens andre klatrer opp og ned. Fartøyene blir hevet og senket ved hjelp av sluser.

En sluse er et kort stykke av kanalen som har vanntette porter i begge ender. En båt som er på vei oppover en kanal, og som altså skal heves til et høyere nivå, går inn i slusen, og så stenges portene. Deretter åpnes sluselukene i den øvre porten, og vannet strømmer inn fra nivået som ligger høyere. Nå løftes båten av det stigende vannet til nivået er det samme som i slusen ovenfor. Så åpnes den øvre porten, og båten kan seile videre. Når en båt skal senkes, slippes vannet ut av slusen. Kanalene må ofte mudres opp så de er dype nok for fartøyene.

Communicating Process Architectures 2012
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2012

XCHANs: Notes on a New Channel Type

Øyvind TEIG [1]
Autronica Fire and Security AS [2], Trondheim, Norway

Abstract. This paper proposes a new channel type, **XCHAN**, for communicating messages between a sender and receiver. Sending on an **XCHAN** is asynchronous, with the sending process informed as to its *success*. **XCHAN**s may be *buffered*, in which case a successful send means the message has got into the buffer. A successful send to an unbuffered **XCHAN** means the receiving process has the message. In either case, a failed send means the message has been discarded. If sending on an **XCHAN** fails, a built-in feedback channel (the ***x**-channel*, which has conventional channel semantics) will signal to the sender when the channel is ready for input (i.e., the next send will succeed). This **x**-channel may be used in a **select** or **ALT** by the sender side (only input guards are needed), so that the sender may passively wait for this notification whilst servicing other events. When the x-channel signal is taken, the sender should send as soon as possible – but it is free to send something other than the message originally attempted (e.g. some freshly arrived data). The paper compares the use of **XCHAN** with the use of output guards in **select**/**ALT** statements. **XCHAN** usage should follow a design pattern, which is also described. Since the **XCHAN** never blocks, its use contributes towards deadlock-avoidance. The **XCHAN** offers one solution to the problem of overflow handling associated with a fast producer and slow consumer in message passing systems. The claim is that availability of **XCHANs** for channel based systems gives the designer and programmer another means to simplify and increase quality.

Keywords. Channels, synchronous, asynchronous, buffers, overflow, flow control, CSP.

CPA-2012

Introduction

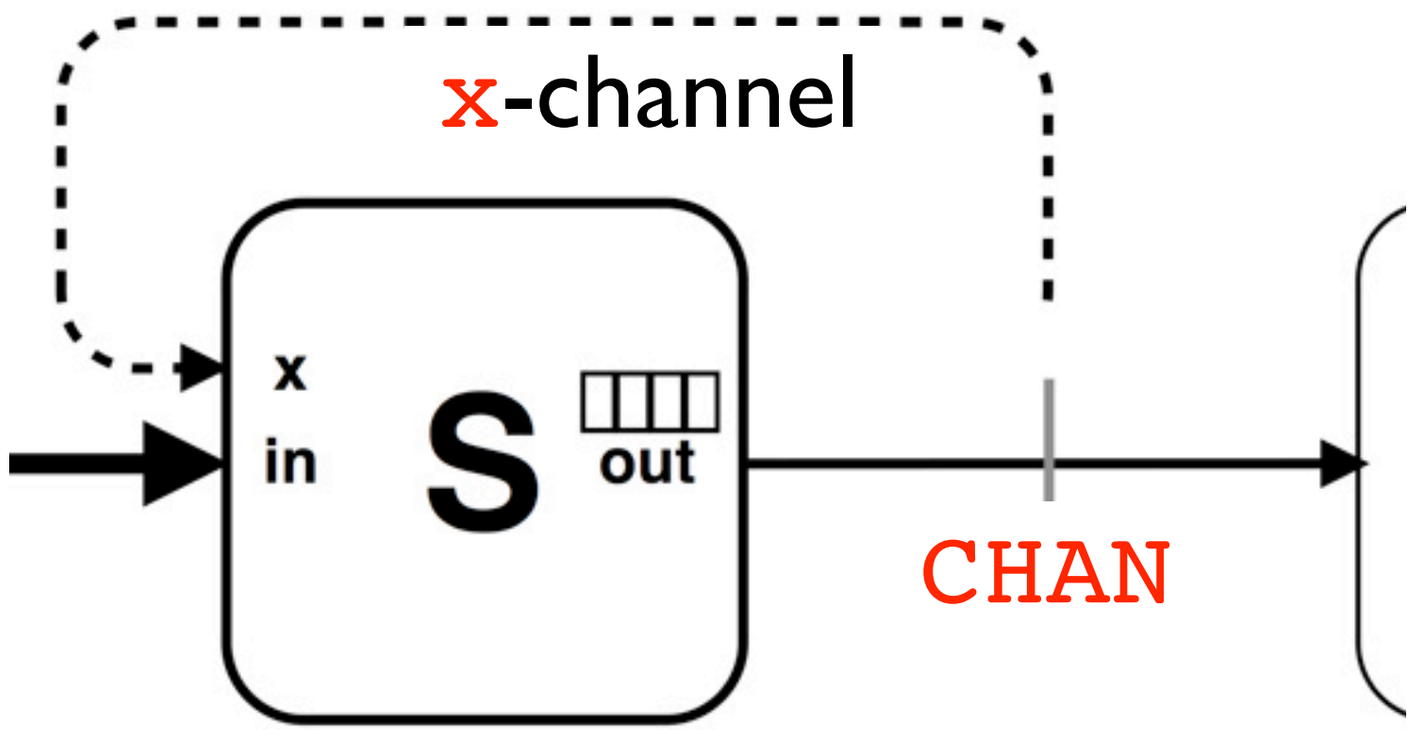
With the advent of the Go programming language [1], channel communication based on the CSP paradigm [2] again seems to have a potential to becoming mainstream. A previous attempt was with the occam programming language [3-5], which gained significant industrial traction during the 19080s and early 1990s (and, of course, is still being developed and applied in academic research [6-8]). Whether new languages with concurrency based on CSP repeat this success remains to be seen.

Nevertheless, channels come in more flavours than the simple channels of occam. This paper suggests a new type of channel that could be added to CSP libraries or become a new primitive in future versions of CSP-based languages. We call this channel an **XCHAN**, which contains a communication channel and a built-in *channel-ready-channel* (the ***x-channel***) for flow control. An **XCHAN** may be sent to (asynchronously) and received from, but the sender must listen on the **x**-channel (usually in an **ALT**/**select**) when sending fails. An **XCHAN** may be buffered.

[1] The author works with concurrent software for fire detection systems, but this "industrial paper" does not necessarily reflect views taken by the company. See http://www.teigfam.net/oyvind/work/work.html.
[2] A *UTC Fire & Security* company, see http://www.autronicafire.com.

XCHAN =

x-channel + CHAN



50

Background

- From discussions at Autronica
- Not implemented
- Goal for me was to try to merge asynchronous and synchronous "camps"..
- ..to arrive at a common methodology
- To make it "easier" to comply to SIL (Safety Integrity Level) approving according to IEC 61508 standard for safety critical systems
- Assumed implementation *loosely* based on implemented ideas with `EGGTIMER` and `REPTIMER`. ([9] CPA-2009 paper)

51

nonbuffered

asynchronous,

```
XCHAN           OF BYTE my_xchan:
```


52

asynchronous, buffered

```
XCHAN (100) OF BYTE my_xchan:
```


53

XCHAN (...) OF BYTE my_xchan:

Sender is notified as to its *success* or "*failure*"

54

XCHAN (...) OF BYTE my_xchan:

Sender is notified as to its *success* on return of send:

- data moved to buffer
- data moved to receiver

55

XCHAN (...) OF BYTE my_xchan:

Sender is notified as to its "*failure*" on return of send:

- buffer full
- receiver not present

56

XCHAN (...) OF BYTE my_xchan:

Sender is notified as to its "*failure*" on return of send:

- buffer full
- receiver not present

It *always* `returns`!

57

If "*failed*" to send on XCHAN:

58

If "*failed*" to send on XCHAN:

"Not sent" is *no fault*!

59

If "*failed*" to send on XCHAN:

"Not sent" is *no fault*!

But a contract to send later

60

If not sent on `XCHAN`:

- listen to x-channel (in an `ALT` or `select`)
- resend old or fresher value when it arrives
- this send will always succeed

61

If not sent:

"channel-ready-channel"

- listen to `x`-channel (in an `ALT` or `select`)
- resend old or fresher value when it arrives
- this send will always succeed

62

If not sent:

- listen to x-channel (in an `ALT` or `select`)
- resend old or fresher value when it arrives
- this send will always succeed

This contract (design pattern) between sender and receiver must be adhered to

63

Fast producer, slow consumer and XCHAN

When Server S cannot get rid of *this* data,
it can still input more,
and finally send *newer* data

64

An XCHAN solution (code)

```
01 while (TRUE) {
02   ALT();
03     ALT_SIGNAL_CHAN_IN (XCHAN_READY);        // data-less
04     ALT_CHAN_IN (CHAN_DATA_IN, Value);
05?  ALT_END(); // Delivers ThisChannelId:
06
07   switch (ThisChannelId) {
08     case XCHAN_READY: {                      // sending will succeed
09!      CP->Sent_Out = CHAN_OUT (XCHAN_DATA_OUT,Value);
10     } break;
11     case CHAN_DATA_IN: {
12       if (!CP->Sent_Out) {
13         ...  handle overflow (decide what value(s) to discard)
14       }
15       else {                                 // sending may succeed:
16!        CP->Sent_Out = CHAN_OUT (XCHAN_DATA_OUT,Value);
17       }
18     } break;
19     _DEFAULT_EXIT_VAL (ThisChannelId)
20   }
21 }
```

Listing 1. Overflow handling and output to buffered channels (ANSI C and macros)

65

Another XCHAN solution

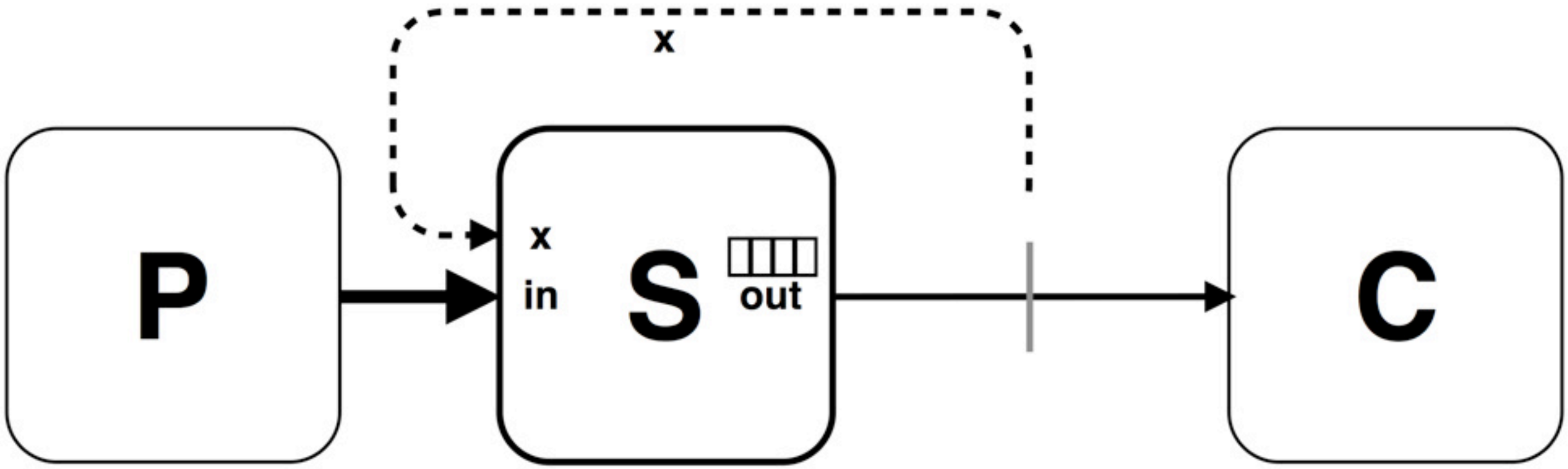


Figure 3. Zero buffered **XCHAN**

66

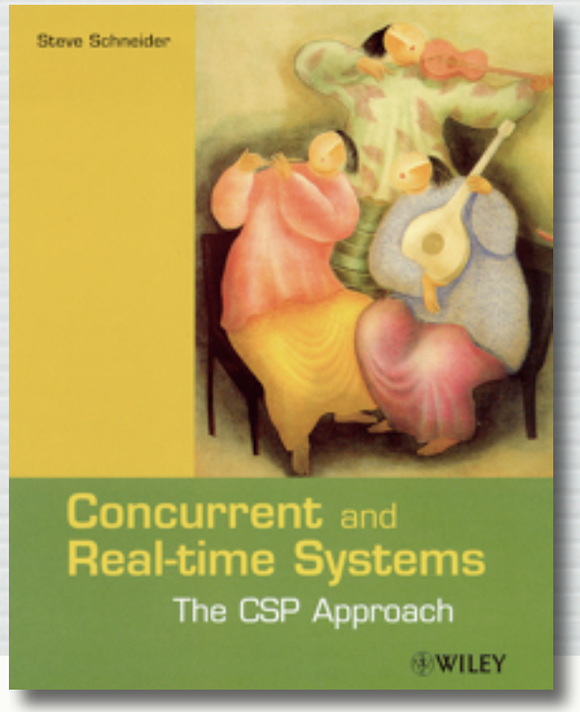
Example 7.4.1 (Buffers) The property of being a buffer of type T is expressible in a process-oriented way, by means of a mutual recursion. Internal choice is used to describe the various possibilities:

Schneider hadde funnet opp dette før meg (og "noe" i unix(?)):

$$NBUFF_T(\langle\rangle) = in?x : T \rightarrow NBUFF_T(\langle x \rangle)$$

$$NBUFF_T(s \frown \langle y \rangle) = out!y \rightarrow NBUFF_T(s) \Box (STOP \sqcap in?x : T \rightarrow NBUFF_T(\langle x \rangle \frown s \frown \langle y \rangle))$$

The parameter to $NBUFF$ consists of the sequence of messages that the buffer currently contains. If this sequence is the empty sequence $\langle\rangle$, then the buffer is empty and must be ready for input. If the sequence contains some messages, then the buffer must be ready to output the next message required. It is also possible that it will accept further input, but it does not have to. These possibilities are represented by the internal choice between $STOP$ and a further input.



Concurrent and Real-time Systems. The CSP Approach. Steve Schneider

The buffer specification $NBUFF_T = NBUFF_T(\langle\rangle)$ also specifies that the alphabet of the buffer is restricted to its input and output channels. It encapsulates the buffer specification given on page 196:

$$NBUFF \sqsubseteq_{SF} IMP \Leftrightarrow IMP \textbf{ sat } (Buff_T(tr), Buff_{SF}(tr, X))$$

If further events are to be possible (such as a channel which can report on whether or not the buffer is empty), then the appropriate specification will be $NBUFF_T \,|||\, CHAOS_A$, where A are the other possible events. The most general specification, which corresponds to $Buff_T(tr, X)$, is given as follows:

IEC 61508
standard for funksjonell sikkerhet

- **Null risiko kan aldri oppnås**
- **Sikkerhet må legges inn fra begynnelsen**
- **Ikke-tolererbare risikoer må begrenses (SIL nivå)**
- **Safety integrity level = SIL**
  SIL 1, SIL 2, SIL 3, SIL *4* (best)
- **Svært opptatt av (at) concurrency (er "vanskelig")**
- **... men standarden er nok tilsvarende vanskelig!**
- **IEC 26262 er en substandard for bilindustrien**

Jeg prøver å skrive på et lite bloggnotat:
http://oyvteig.blogspot.com/2012/01/038-concurrent-programming-paradigms.html

Fra en artikkel om data*etikk* og formelle metoder

For example, if a program changes a single "1" to a "0", the entire program could result in completely opposite behavior. De Millo, Lipton, and Perlis call this property the discontinuity property of software...

.. imagine if one missing bolt in an American aircraft carrier turned it into a Soviet submarine. That's software.

"When Formal Systems Kill: Computer Ethics and Formal Methods"
Darren Abramson Lee Pikey
https://www.cs.indiana.edu/~lepike/pubs/fm-ethics.pdf

61508 & *etikk*valg?

- Språk:
  - C, MISRA-C
  - C++, MISRA-C++
  - Java
  - Go
  - ...
- Operativsystem, rammeverk. Hyllevare:
  - Linux (subset, *L4* mikrokjerne)?
  - ThreadX, VxWorks, Integrity, QNX?
  - AUTOSAR?
  - ...
- Hjemmesnekret?
  - SDL kjøresystem
  - ChanSched
  - BufXChanSched?
  - ...

http://oyvteig.blogspot.com/2012/01/038-concurrent-programming-paradigms.html

Communicating Process Architectures 2012
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2012

XCHANs: Notes on a New Channel Type

Øyvind TEIG [1]
Autronica Fire and Security AS [2], Trondheim, Norway

Abstract. This paper proposes a new channel type, **XCHAN**, for communicating messages between a sender and receiver. Sending on an **XCHAN** is asynchronous, with the sending process informed as to its *success*. **XCHAN**s may be *buffered*, in which case a successful send means the message has got into the buffer. A successful send to an unbuffered **XCHAN** means the receiving process has the message. In either case, a failed send means the message has been discarded. If sending on an **XCHAN** fails, a built-in feedback channel (the **x**-*channel*, which has conventional channel semantics) will signal to the sender when the channel is ready for input (i.e., the next send will succeed). This x-channel may be used in a **select** or **ALT** by the sender side (only input guards are needed), so that the sender may passively wait for this notification whilst servicing other events. When the x-channel signal is taken, the sender should send as soon as possible – but it is free to send something other than the message originally attempted (e.g. some freshly arrived data). The paper compares the use of **XCHAN** with the use of output guards in **select**/**ALT** statements. **XCHAN** usage should follow a design pattern, which is also described. Since the **XCHAN** never blocks, its use contributes towards deadlock-avoidance. The **XCHAN** offers one solution to the problem of overflow handling associated with a fast producer and slow consumer in message passing systems. The claim is that availability of **XCHAN**s for channel based systems gives the designer and programmer another means to simplify and increase quality.

Keywords. Channels, synchronous, asynchronous, buffers, overflow, flow control, CSP.

CPA-2012

Introduction

With the advent of the Go programming language [1], channel communication based on the CSP paradigm [2] again seems to have a potential to becoming mainstream. A previous attempt was with the occam programming language [3-5], which gained significant industrial traction during the 19080s and early 1990s (and, of course, is still being developed and applied in academic research [6-8]). Whether new languages with concurrency based on CSP repeat this success remains to be seen.

Nevertheless, channels come in more flavours than the simple channels of occam. This paper suggests a new type of channel that could be added to CSP libraries or become a new primitive in future versions of CSP-based languages. We call this channel an **XCHAN**, which contains a communication channel and a built-in *channel-ready-channel* (the ***x-channel***) for flow control. An **XCHAN** may be sent to (asynchronously) and received from, but the sender must listen on the **x**-channel (usually in an **ALT**/**select**) when sending fails. An **XCHAN** may be buffered.

[1] The author works with concurrent software for fire detection systems, but this "industrial paper" does not necessarily reflect views taken by the company. See http://www.teigfam.net/oyvind/work/work.html.
[2] A *UTC Fire & Security* company, see http://www.autronicafire.com.

Communicating Process Architectures 2013
P.H. Welch et al. (Eds.)
Draft 2013

Teig-r1.doc 22.April 2013

Selective choice 'feathering' with XCHANs

Øyvind TEIG[1]
Autronica Fire and Security AS[2], Trondheim, Norway

Abstract. This paper suggests an additional semantics to **XCHAN**s, where a sender to a synchronous channel that ends up as a component in a receiver's selective choice (like **ALT**) may (if wanted) become signaled whenever the **ALT** has been (or is being) set up with the actual channel *not* in the active channel set. Information about this is either received as the standard return on **XCHAN**'s attempted sending or on the built-in feedback channel (called **x**-channel) if initial sending failed. This semantics may be used to avoid having to send (and receive) messages that have been seen as *uninteresting*. We call this scheme *feathering*, a kind of low level *implicit* subscriber mechanism. The mechanism may be useful for systems where channels that were not listened to while listening on some *other* set of channels, will not cause a later including of those channels to carry already declared uninteresting messages. It is like not having to treat earlier bus-stop arrival messages for the wrong direction after you sit on the first arrived bus for the correct direction. The paper discusses the idea as far as possible, since modeling or implementation has not been possible. This paper's main purpose is to present the idea.

Keywords. Channels, synchronous, asynchronous, buffers, overflow, flow control, CSP, modeling, semantics, *feathering*

Introduction

Sendt til CPA-2013

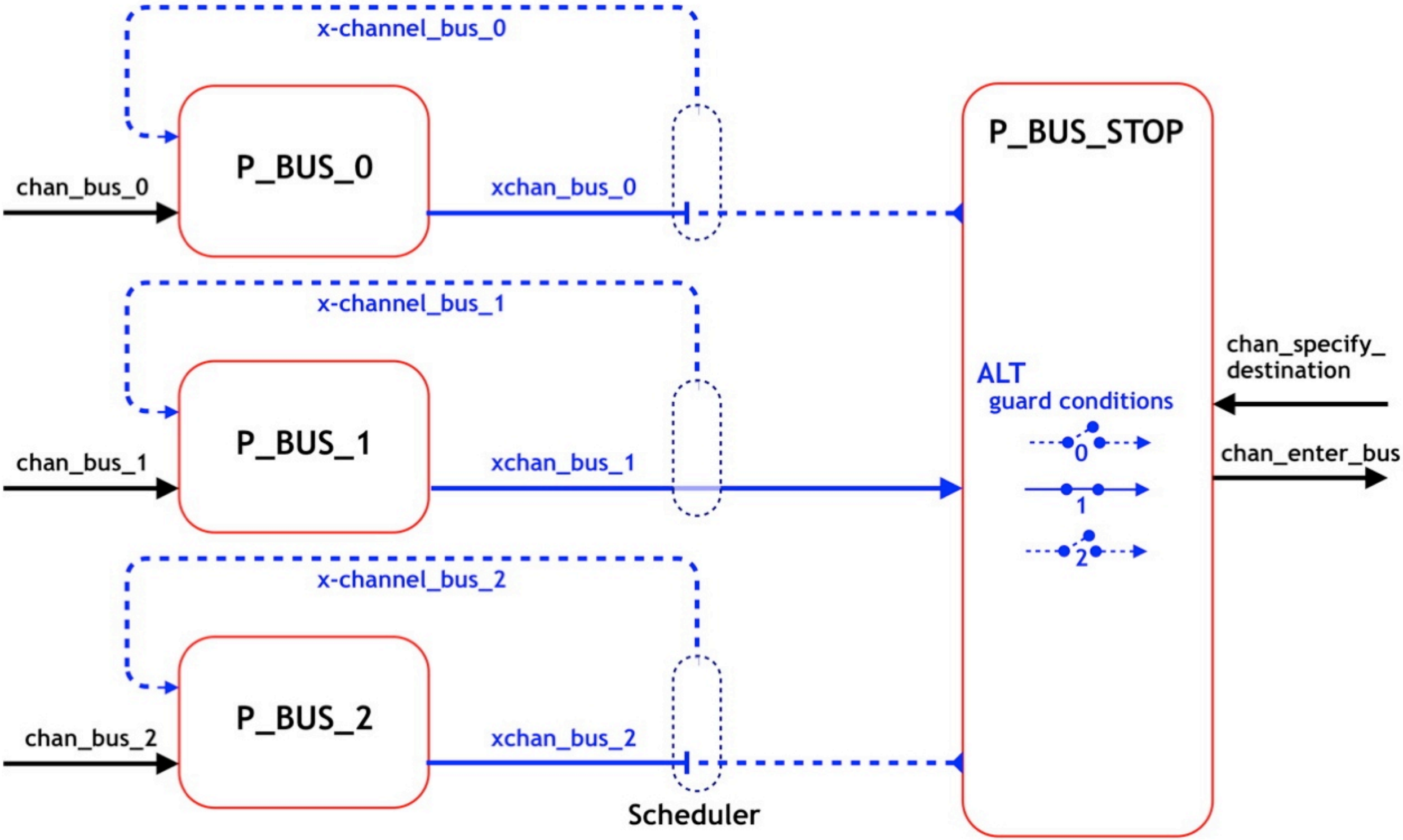
The idea of the suggested semantics appeared after reading Tony Hoare's lecture "Concurrent programs wait faster" from 2003 [1]. After some pondering of a situation described there, we saw that it might be viable to use **XCHAN** [2] as a vehicle for a secondary problem not mentioned in the paper. The problem is how to avoid having to relate to information of busses that would potentially stop at a bus stop but heading in the wrong destination. It was believed that this could map to a new software pattern: *feathering*.

The word *feathering* is used like "turning an oar parallel to the water between pulls" [3]. Metaphorically a pull is like an **ALT** selective choice. The oar is pushed into the water at one place: only one channel is taken. But we can hear the oar whip the top of the small waves on its way saying "was there, but not interested". So we take the step to name barely touching the small waves for *feathering*.

The author is not aware of **XCHAN** having been implemented in any language or run-time system. The time when a "channel was a channel" seems to be over; the plethora of channel types and channel usage seems to increase. Worth mentioning here is Go's channel usage and semantics [4], which are quite different from what we are used to in the **occam** tradition. The **XCHAN** is here perceived as being an expansion of the traditional channel (**CHAN**) type.

This feathering semantics pattern has not been modeled or formally proven. The paper informally introduces the ideas. Also, no literature search has been done to find similar or

[1] The author works with concurrent software for fire detection systems, but this "industrial paper" does not necessarily reflect views taken by the company. See http://www.teigfam.net/oyvind/work/work.html
[2] A UTC Fire & Security company, see http://www.autronicafire.com


x-channel_bus_0
P_BUS_0
chan_bus_0
xchan_bus_0
x-channel_bus_1
P_BUS_1
chan_bus_1
xchan_bus_1
x-channel_bus_2
P_BUS_2
chan_bus_2
xchan_bus_2
Scheduler
P_BUS_STOP
ALT
guard conditions
0
1
2
chan_specify_
destination
chan_enter_bus

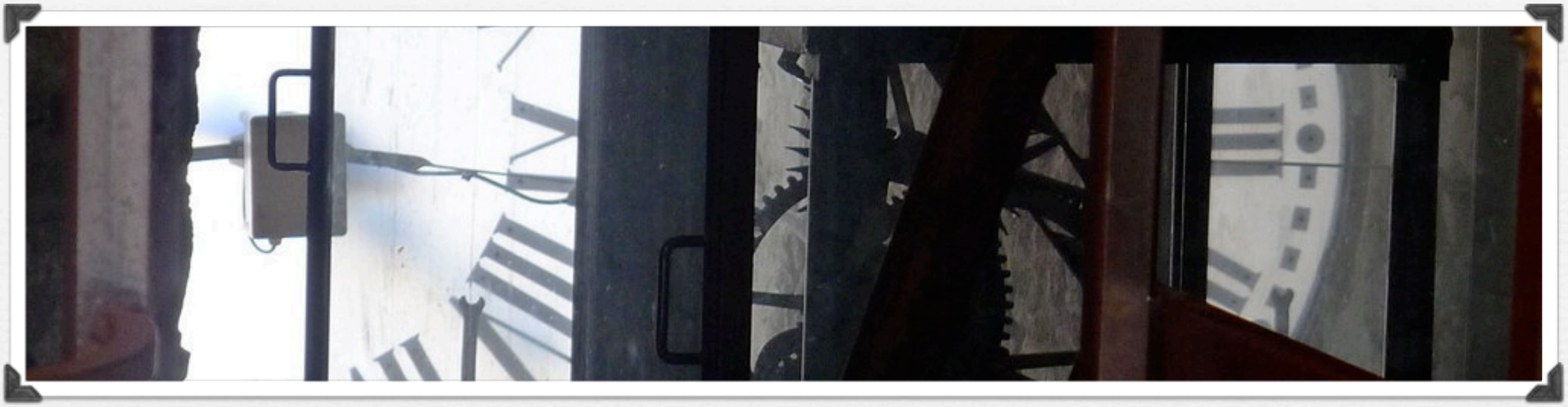
Kontaktinfo

Dette foredraget: http://www.teigfam.net/oyvind/pub/NTNU_2013/foredrag.pdf

Dette faget på NTNU: http://www.itk.ntnu.no/fag/TTK4145/information/


```
oyvind.teig@autronicafire.no
oyvind.teig@teigfam.net
```


Les dette og mer på http://www.teigfam.net/oyvind/pub/



- *Plutselig* er dere besteforeldre!
  - for i 2050 er det 37 år siden 2013

1976

- *Plutselig* er dere besteforeldre!
  - for i 2050 er det 37 år siden 2013

Da har du jobbet i 37 år!

2013

Fra harde µ-sekunder til forte år: sann tid i industrien

takk
for meg!

forresten..

Mastergradsoppgaveforslag våren 2013 på
http://www.itk.ntnu.no/ansatte/Onshus_Tor/Master.htm

«Analyse av meldingssystemer i sikkerhetskritiske systemer begrenset av IEC 61508»