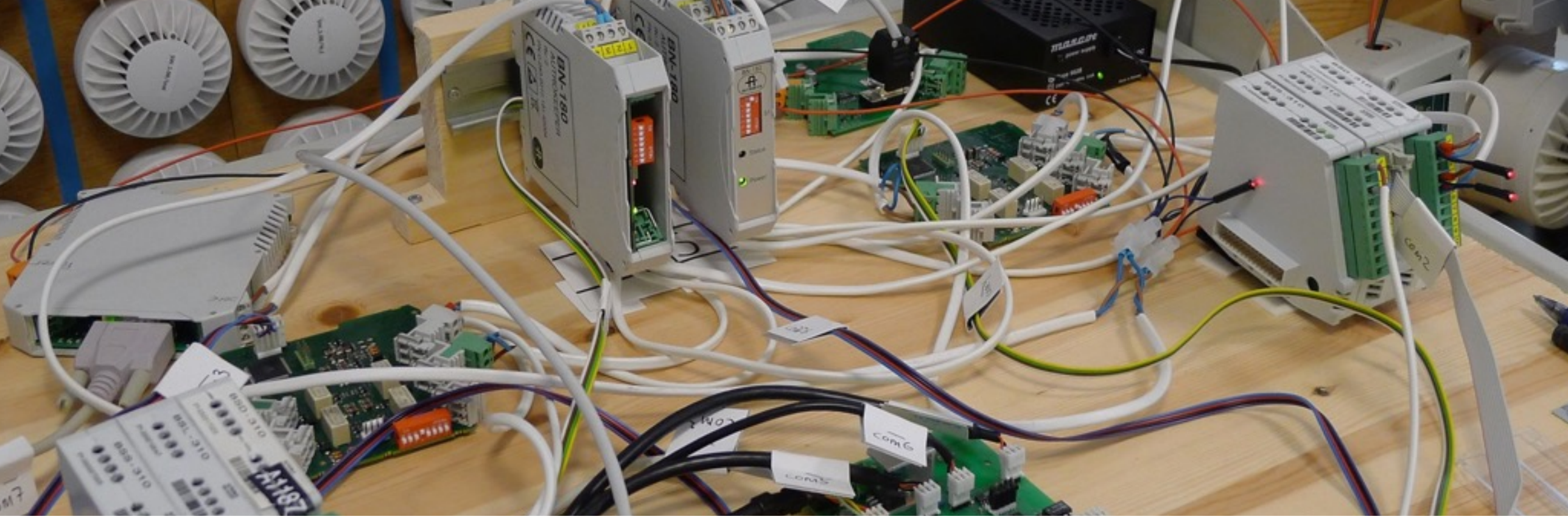
FROM HARD MICROSECONDS TO SPEEDY YEARS

# REAL TIME IN THE INDUSTRY

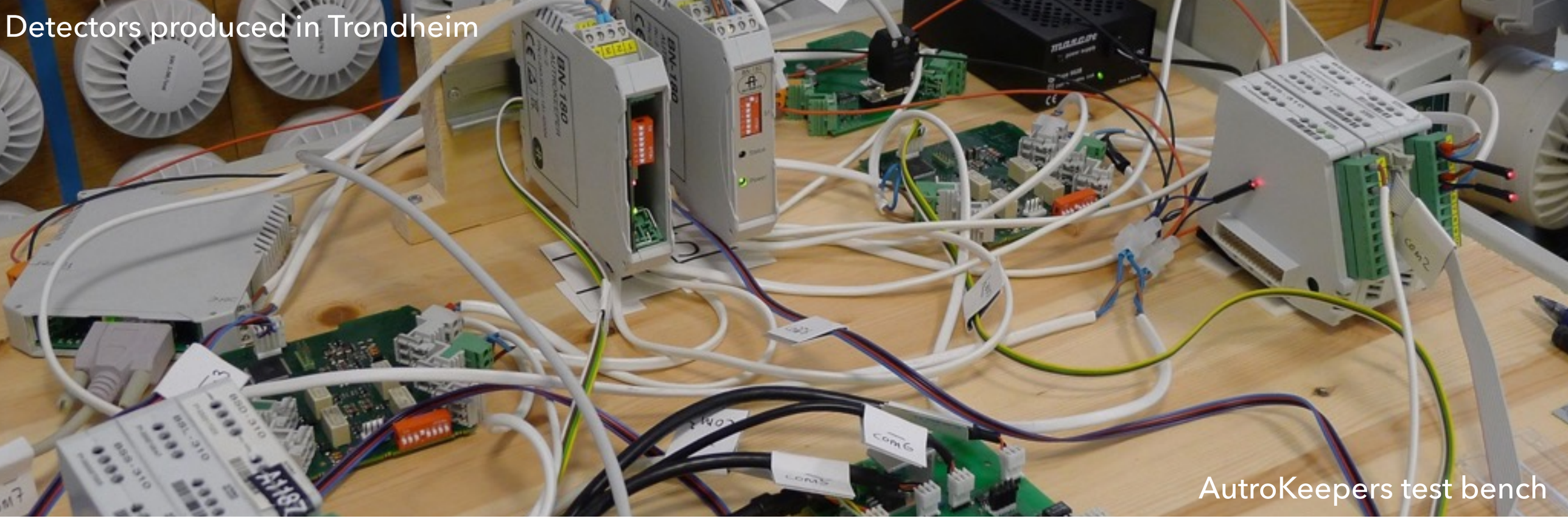



ØYVIND TEIG

SENIOR DEVELOPMENT ENGINEER, AUTRONICA

# INVITED SPEAKER, 26. APRIL 2016 AT NTNU, TTK4145 SANNTIDSPROGRAMMERING (REAL-TIME PROGRAMMING)

Version of 26.April 2016 14.10




ØYVIND TEIG
SENIOR DEVELOPMENT ENGINEER, AUTRONICA

# INVITED SPEAKER, 26. APRIL 2016 AT NTNU, TTK4145 SANNTIDSPROGRAMMERING (REAL-TIME PROGRAMMING)

AUTRONICA

FIRE AND SECURITY

PART OF UTC SINCE 2005

# FIRE DETECTION SINCE 1957


## PART OF UTC SINCE 2005

# FIRE DETECTION SINCE 1957

Autronica didn’t start with fire detection at once. Road work beacons were among the first products

Above: copper detail from a rather new circuit board

# AUTRONICA FIRE AND SECURITY (AFS)

- UTC = United Technologies Corporation
- UTC Climate, controls and security

- Based in Trondheim: 480 staff worldwide
- R&D *also* in Gdansk
- Production *mostly* in Trondheim

- Pratt & Whitney (airplane engines)
- UTC Aerospace Systems
- Otis (elevators)

## WORLD WIDE MARKETS, EXAMPLES:

From autronicafire web page right now

Onshore Market

Maritime Market

- Hospitals, industry, airports
- Cruise ships, ferries, megayachts(!)
- Offshore rigs, onshore petrochemical

**Maritime Market**

A long-standing player in the maritime market - safeguarding all types of vessels; from the worlds largest and most prestigious cruiseships to sm...Full Story >

View More Stories >

Oil and Gas Market

# DISCLAIMERS

- Life at work from my desk: embedded controllers
- Computer science related: concurrency
- Personal experience and opinions: ..CSP-type systems..

- No Autronica-sensitive information here
- Even when different practices may shine through
  do remember that..
  - ..all Autronica products are approved
    to applicable norms and standards

- NTH, 1975
- Autronica:1976-2016 and student-work 72-74
  - All of Autronica's logos in my career
- HW and SW (mostly SW)
- I have worked with embedded systems all the time
- Published some – rather unusual in the industry
  - All publications are on my web page
- Technical writer writing blog notes

# I HAVE BEEN SOLD WITH ABOUT 20 OTHERS

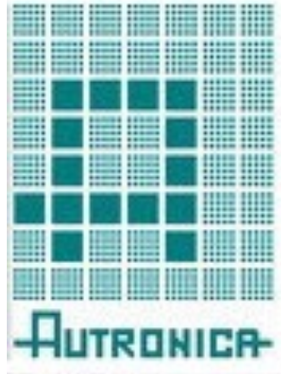




2002: Back to AFS


Kongsberg


Navia

# YEAR BY YEAR (PHOTOS)

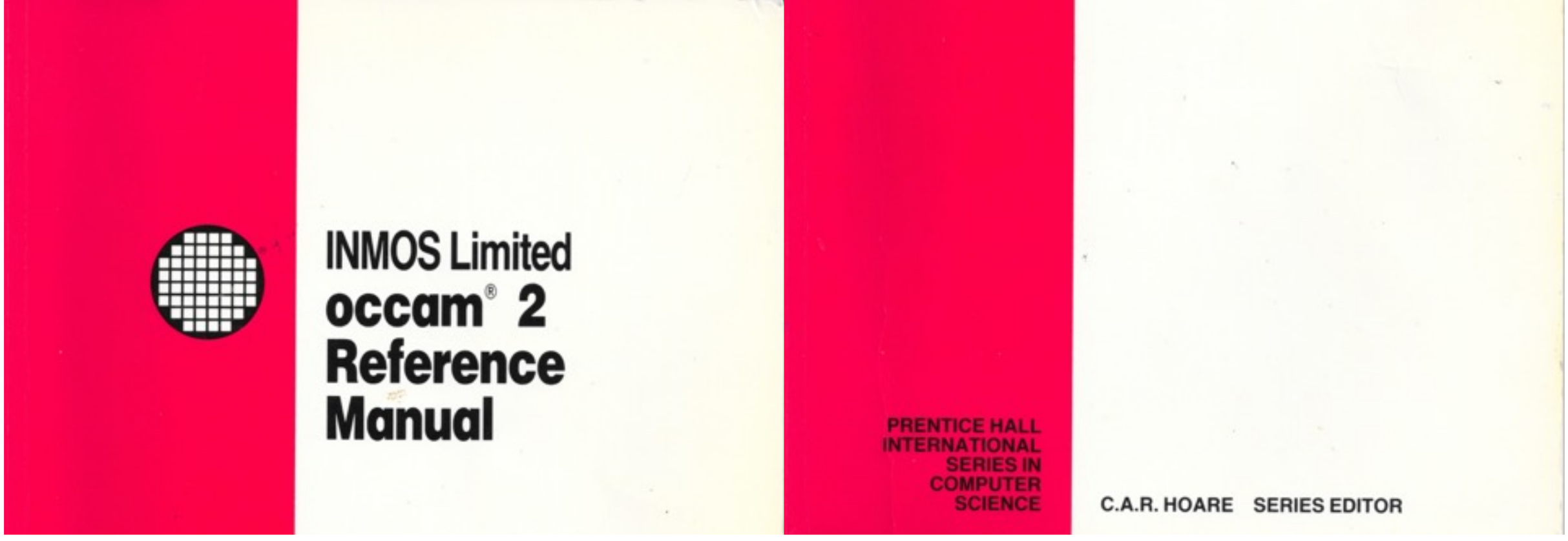
Discussing new runtime (1981)

At Whessoe in Newton-Aycliffe (UK) working with a 16-bits transputer (1995)

Back to AFS (2002)

# LANGUAGES (THAT I HAVE USED)

- Assembler (1975-1980)
- PL/M (1980-1990)
- Modula-2 (1988-1990)
- MPP-Pascal (1982-1988)




# LANGUAGES (THAT I HAVE USED)

- Assembler (1975-1980)
- PL/M (1980-1990)
- Modula-2 (1988-1990)
- MPP-Pascal (1982-1988)
- occam (1990-2001)

# LANGUAGES (THAT I HAVE USED)

- Assembler (1975-1980)
- PL/M (1980-1990)
- Modula-2 (1988-1990)
- MPP-Pascal (1982-1988)
- occam (1990-2001)
- C (2002-present)
- [Java (1997-2000)]
- [Perl (2002)]

# RUNTIMES / SCHEDULERS

- Runtimes/schedulers is about which «process models» we have used:
  rather than all code in a big loop in main

# RUNTIMES – ONE WITH EARLY PROCESS SUPPORTED BY IDE

- 1978: No runtime system, assembly only
  *Diesel start/stop for emergency power (AA)*
- 1979: MPP Pascal with early «process» term
  *Protocol conversion (EAC), fluid level measurement (GL) and fire detection (BS)*
- 1980: PL/M with NTH-developed run-time
  *Ship's machine room monitoring (EA)*

# EARLY HOME BREWN PROCESSES

- 1982: Assembler with runtime
  *Fire detection (BS-30)*

- 1988: PL/M with runtime
  *Fire detection*
  *(BS-100 loop controller)*

# EARLY HOME BREWN PROCESSES

- 1982: Assembler with runtime
  *Fire detection (BS-30)*
- 1988: PL/M with runtime
  *Fire detection*
  *(BS-100 loop controller)*

## The anatomy of a process

A process often consists of an initializing part and an eterna
Comments in PL/M were standard /* .. */, but I have us
here.

```
rc0__:
DO;
  process_0: PROCEDURE PUBLIC;
  DECLARE <process_0 variables>;

    subroutine: PROCEDURE;
    DECLARE <subroutine variables>;
      <subroutine (stack) level>
    END subroutine;

    <process initializing part>
    <process (stack) level>
    DO WHILE 1=1;
      <process eternal loop part>
      <process (stack) level>
    END;
  END process_0;
  <system level, no code in processes>
END prc0__;
```

Later, "FOREVER" is used for "WHILE 1=1". The only f
has system level code, is the scheduler itself. Thus the linl
that code part at power-up

## Call to the scheduler (rx)

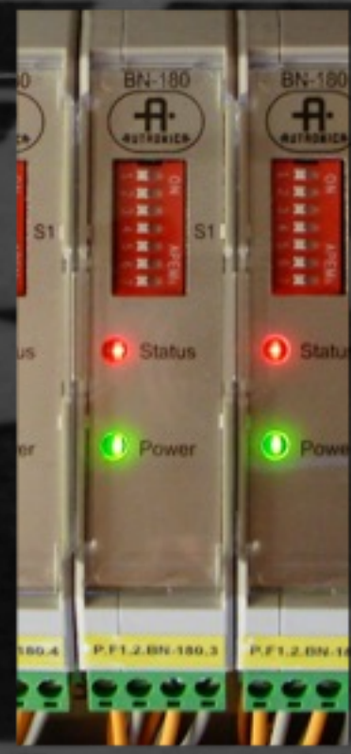
Usually the initializing part is run without calls to th
because of the scheduling philosophy all processes must c
to the scheduler. A call to the scheduler will look like this:

```
disable;
call rx (next);
```

# PROCESSES BY OTHERS

- 1988: Modula-2 with purchased run-time and coroutines
  *Fire detection (BS-100 panel)*
- 1995: C with VxWorks os
  *Fire detection (AutroSafe)*



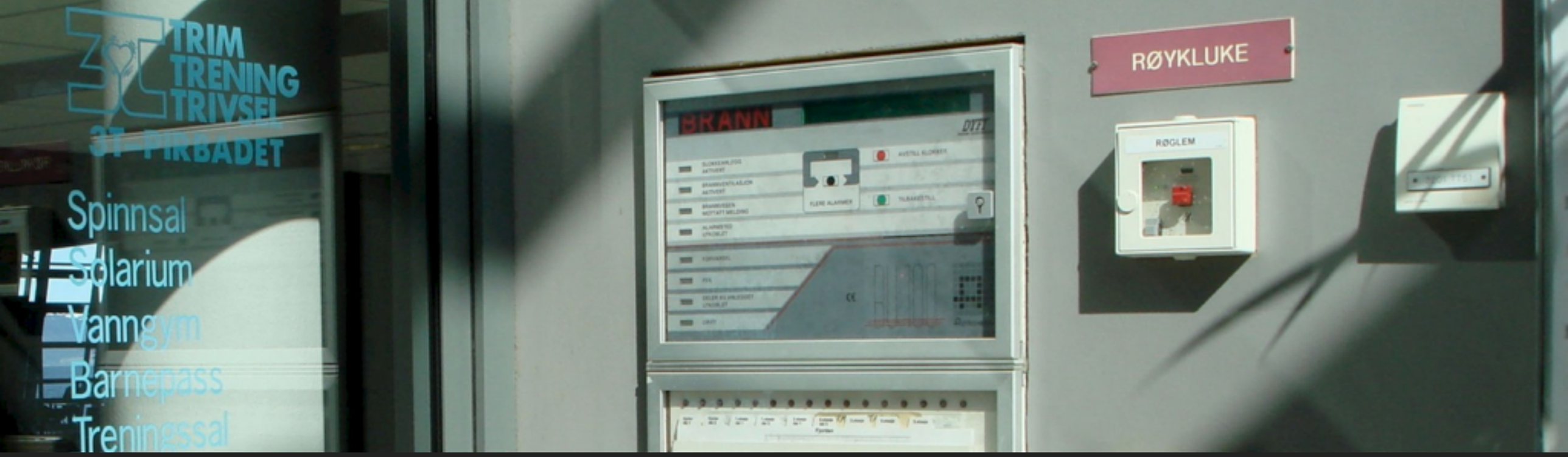



# YEAR BY YEAR (PHOTOS)

BS-100 fire panel (1990..)

Last BS-100 for a ship (2011)

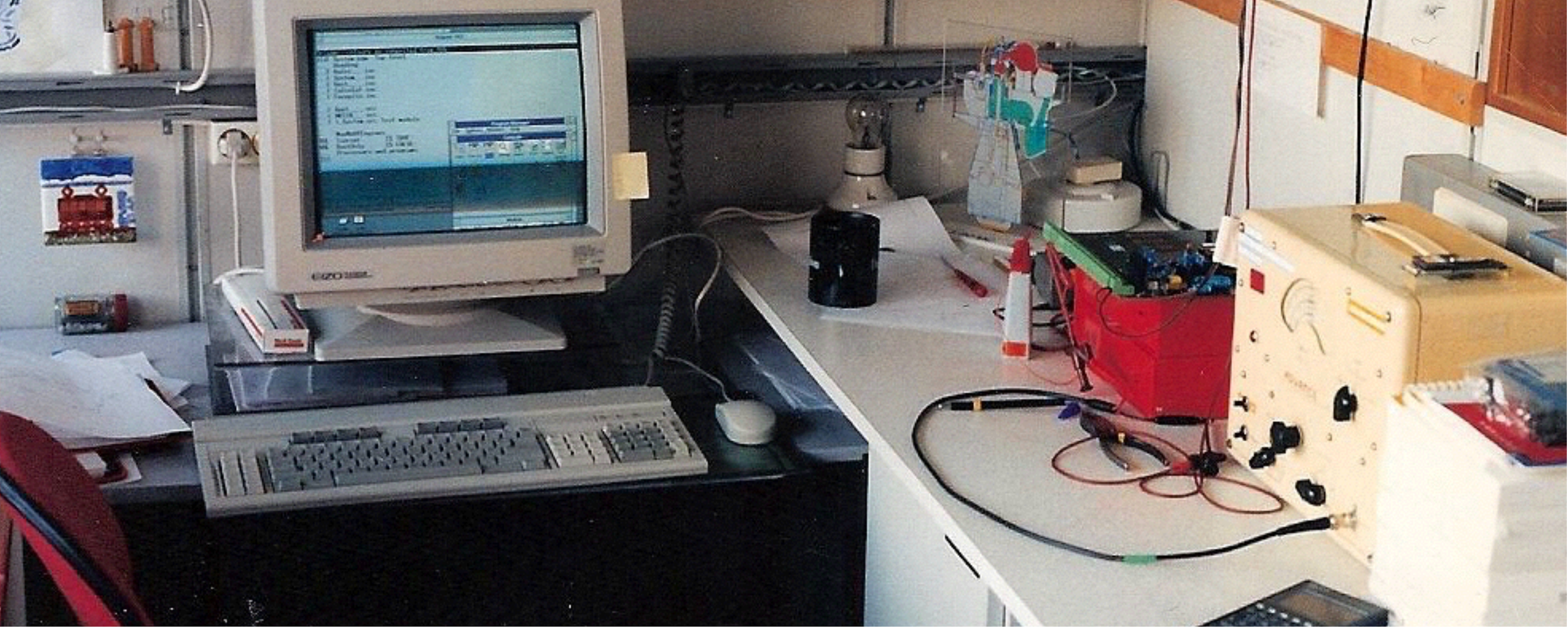
AutroKeeper (2010..)




# BS-100 WITH PROCESSES: 1988-2016..

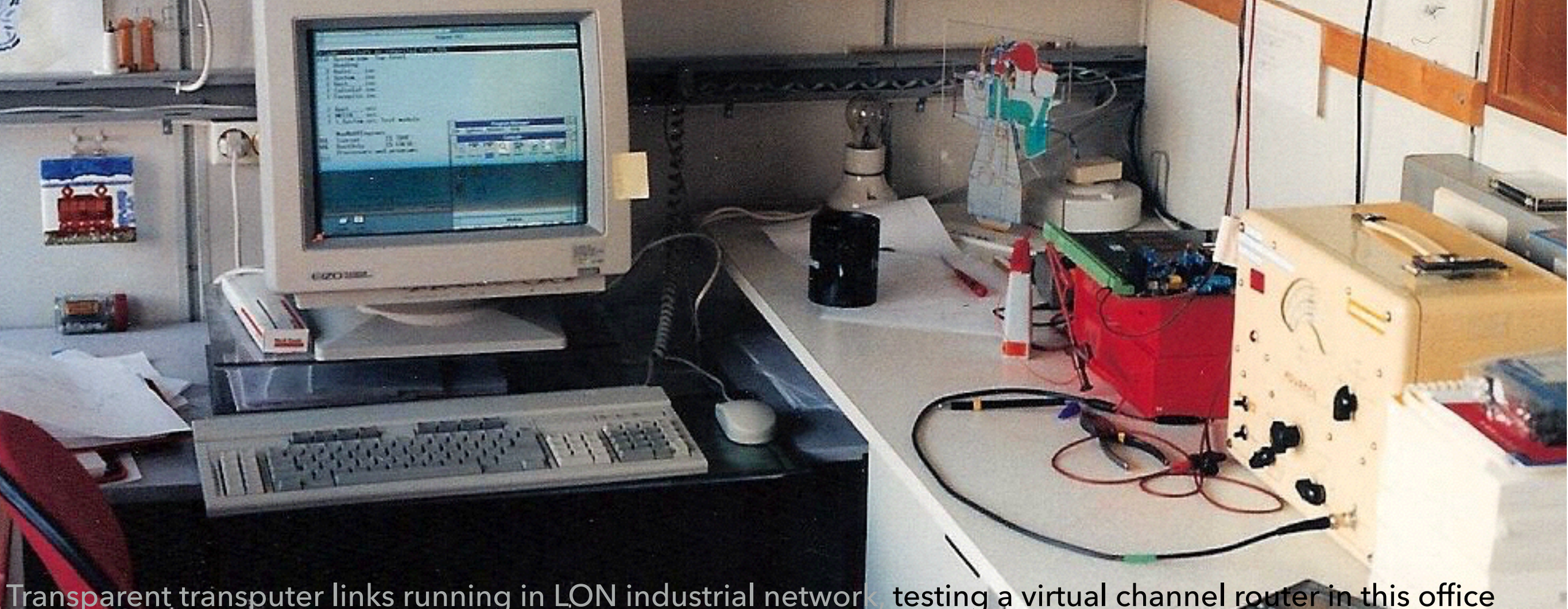
# SIMPLICITY THAT PAID OFF

From autronicafire web page right now



# IN WORK: NOTHING THE SAME AFTER

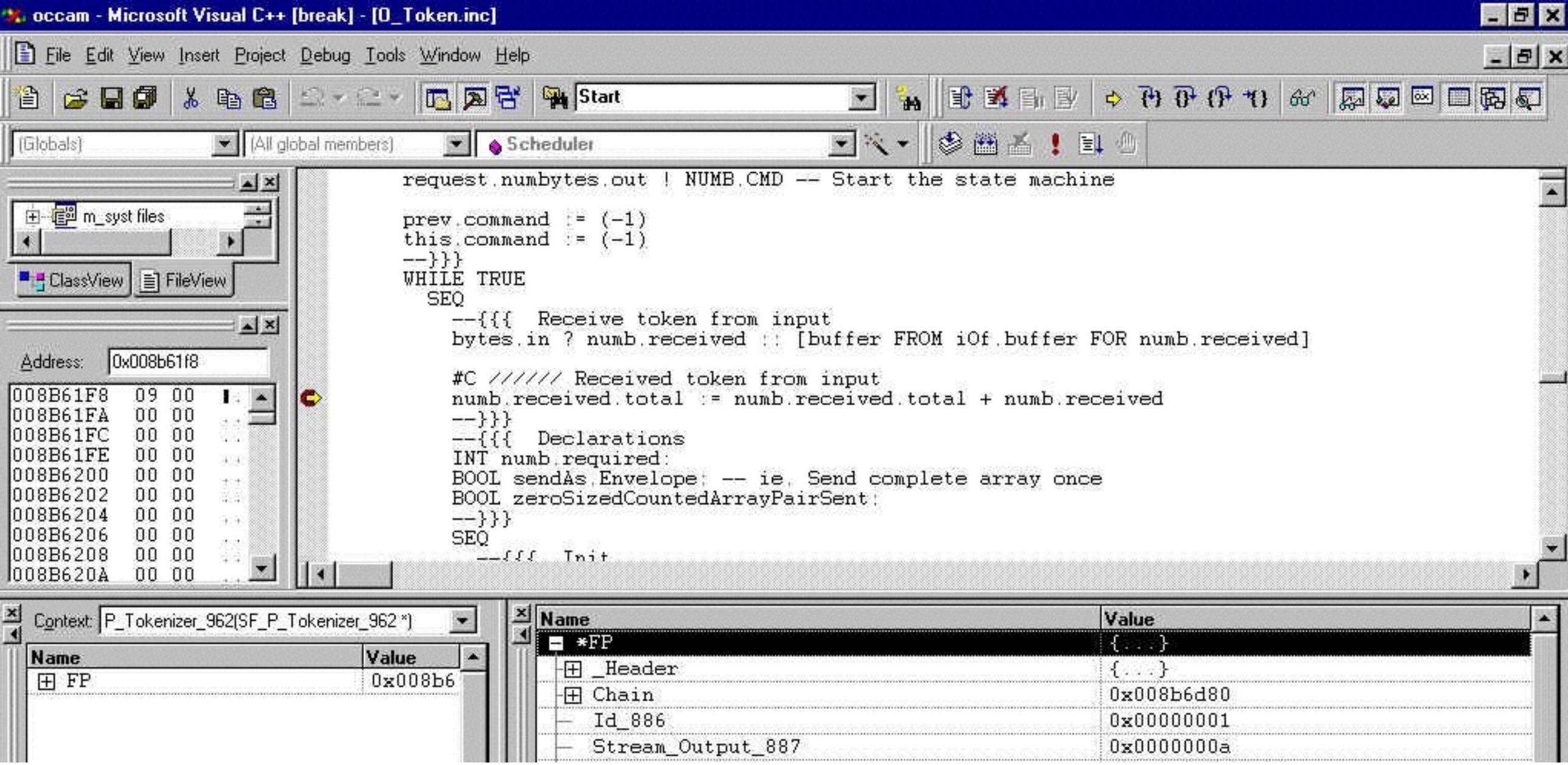
## 1990: OCCAM WITH PROCESS AND CHANNELS. SHIP'S ENGINE CONDITION MONITORING (MIP-CALCULATOR: NK-100)



Transparent transputer links running in LON industrial network, testing a virtual channel router in this office

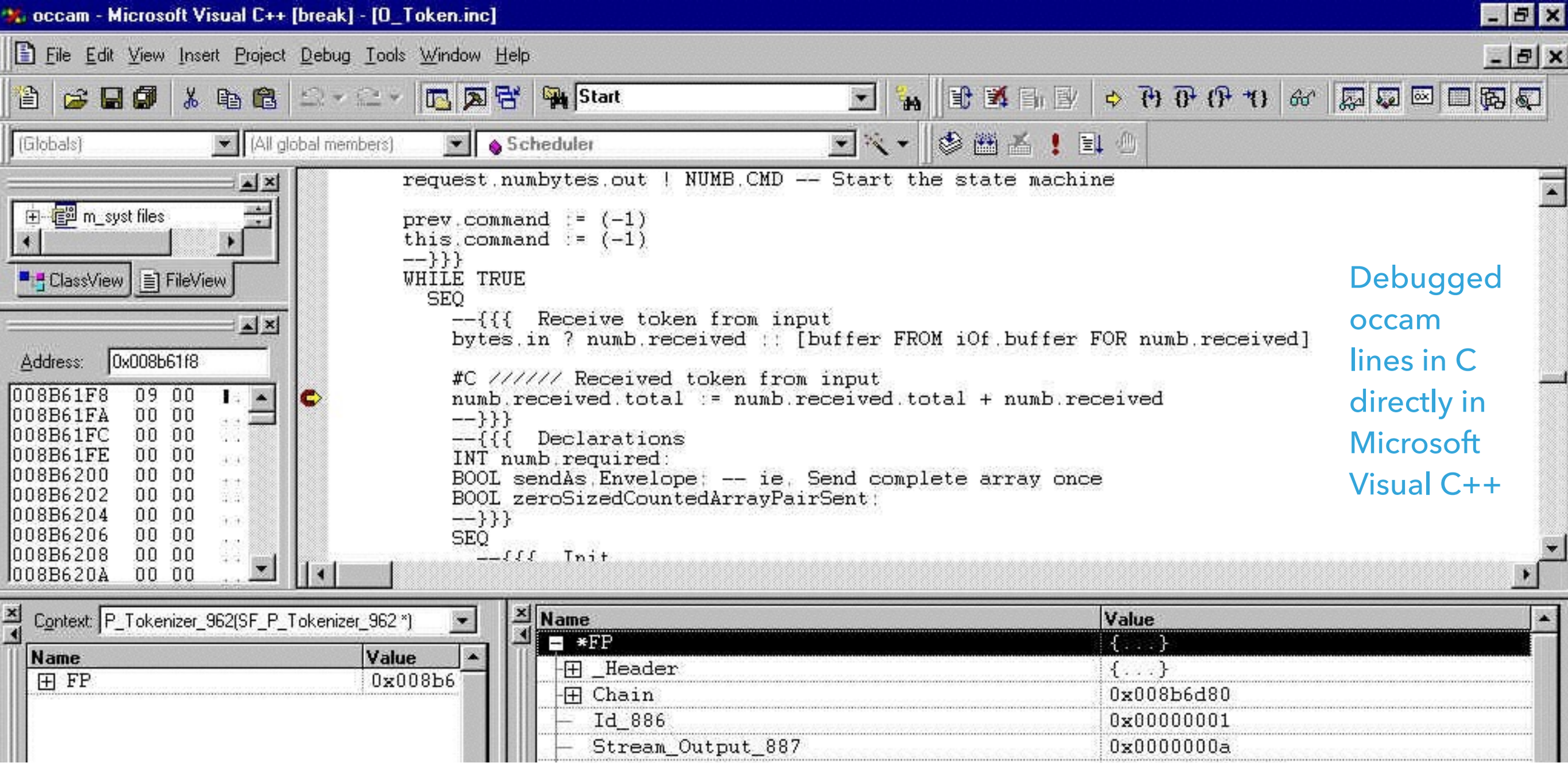
# IN WORK: NOTHING THE SAME AFTER

## 1990: OCCAM WITH PROCESS AND CHANNELS. SHIP'S ENGINE CONDITION MONITORING (MIP-CALCULATOR: NK-100)




# C? YES: OCCAM TO C: SPOC

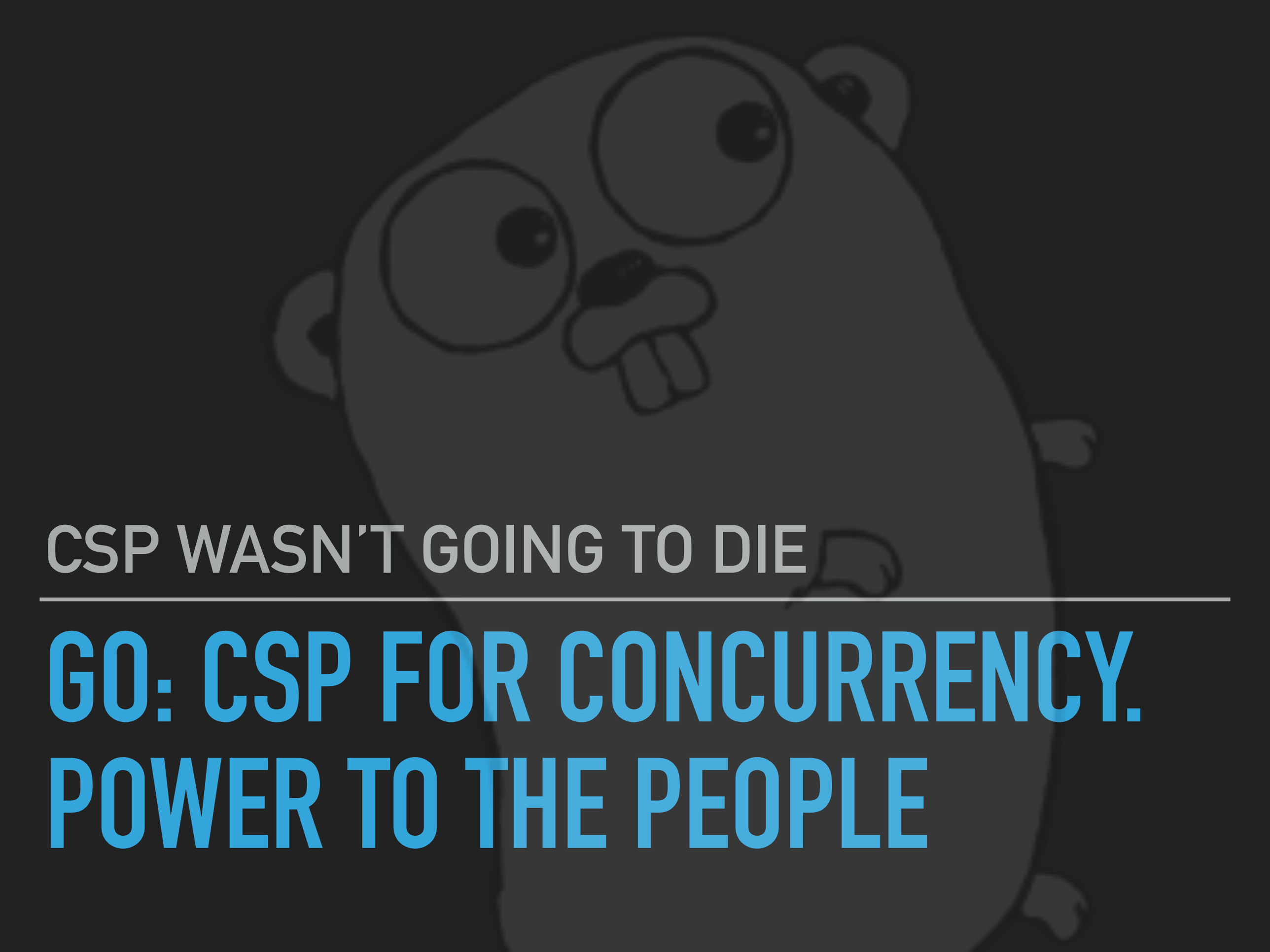
# 1995: OCCAM TO C ON SIGNAL PROCESSOR (MIP-CALCULATOR: NK-200) & NTH DIPLOMA




Debugged occam lines in C directly in Microsoft Visual C++

# C? YES: OCCAM TO C: SPOC

# 1995: OCCAM TO C ON SIGNAL PROCESSOR (MIP-CALCULATOR: NK-200) & NTH DIPLOMA

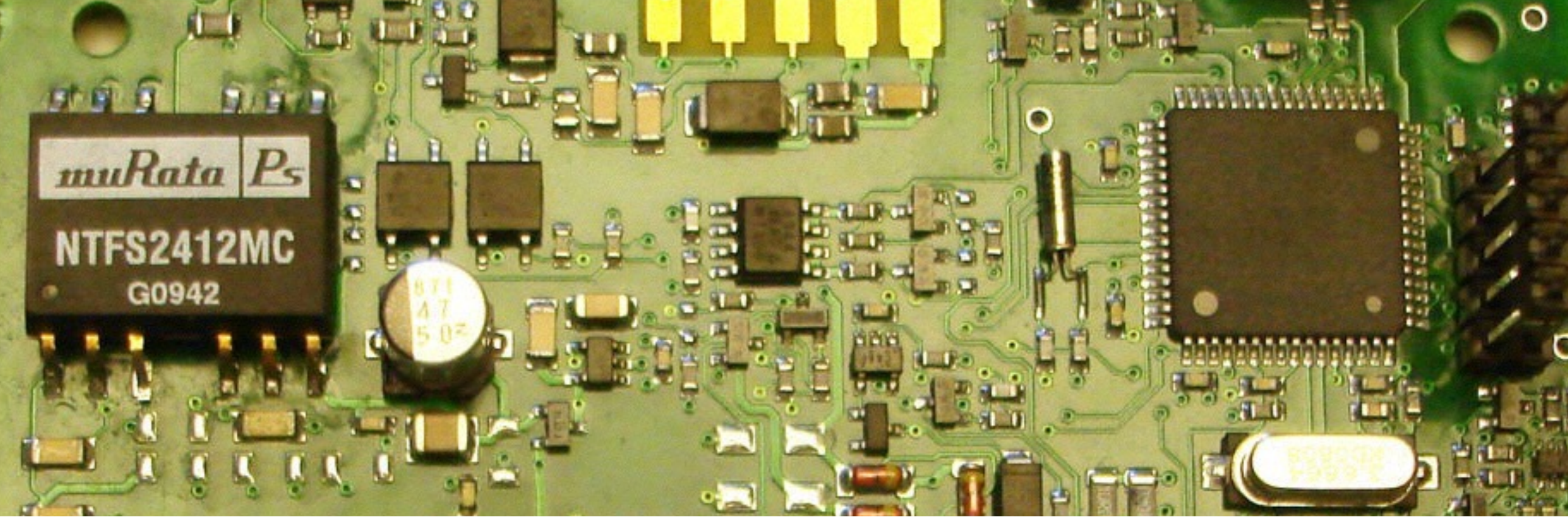


CSP WASN'T GOING TO DIE

# GO: CSP FOR CONCURRENCY. POWER TO THE PEOPLE

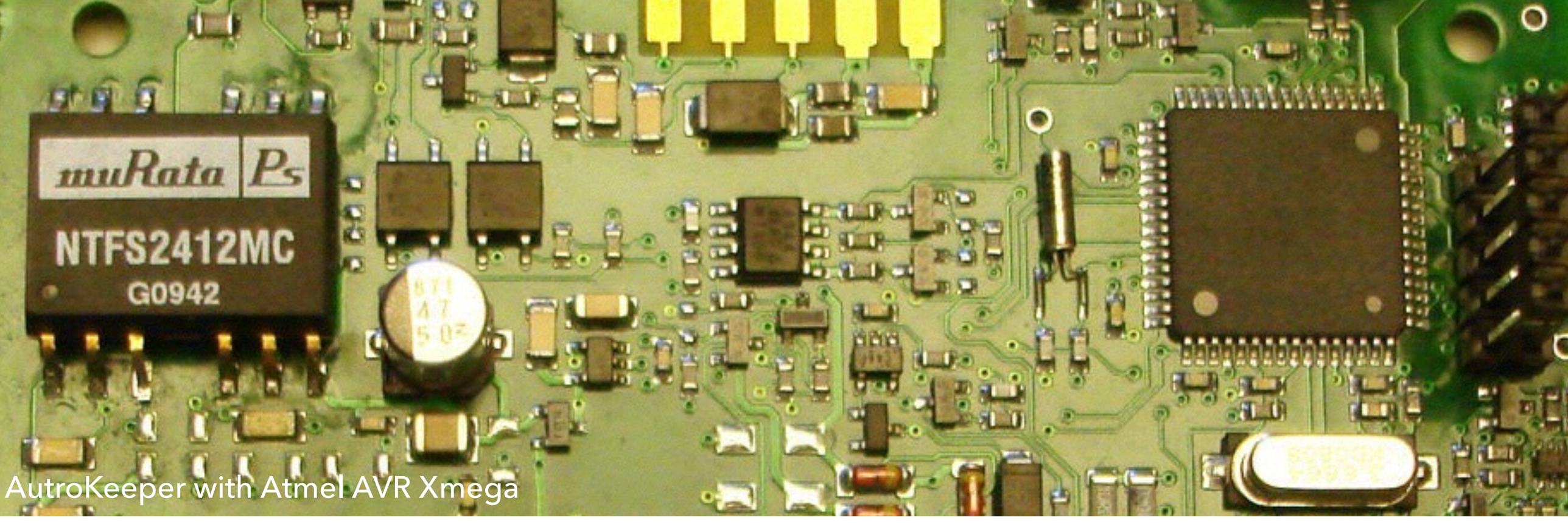
- Go from Google picked up some occam!
- chan, go, select, explicit type conversion, array-slicing, no pointer arithmetics
- However, no «parallel usage rules»
- ..but several tools to analyse, like the runtime Go Race Detector

- I did occam at work for some ten years..
- but Go isn’t for embedded..
- so I can only read (& dream &blog) about Go (again: Since it has channels and all)..
- and go for C plus our runtimes




# SMALL EMBEDDED SYSTEMS

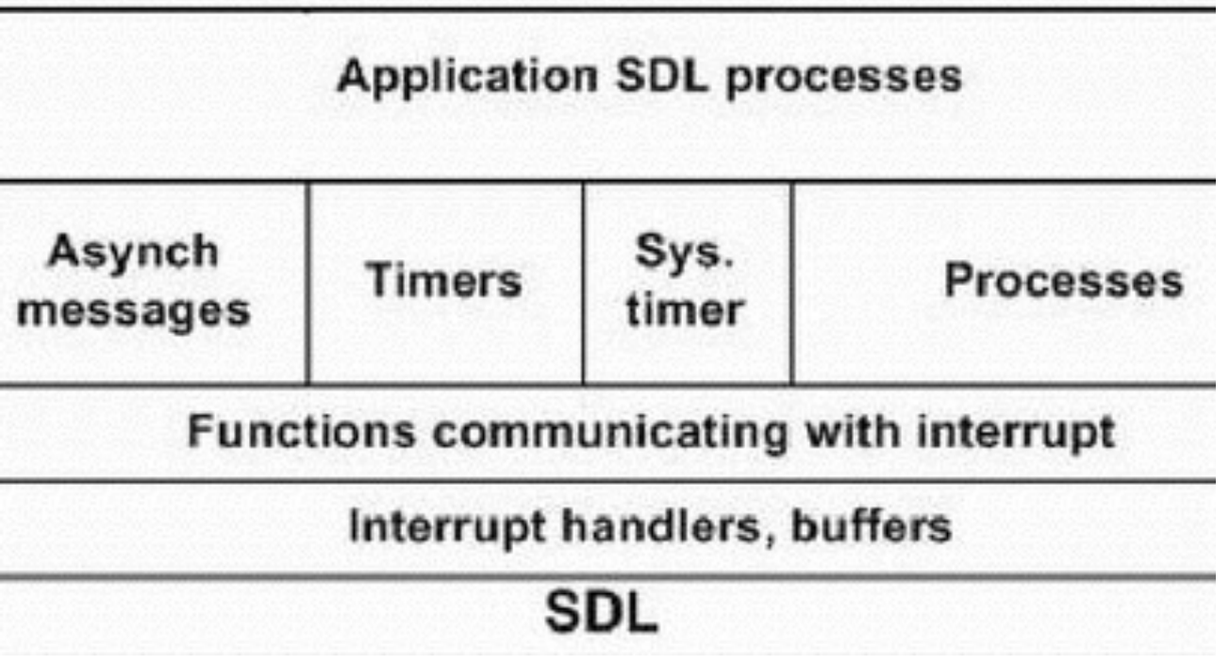
- Will probably keep C for a long time!
- Project managers need to learn about the «Go potential»
- Don’t take over their toolset without adding your knowledge
  - Like channels and tight processes
    - Even if it will be hard to find runtimes. However..

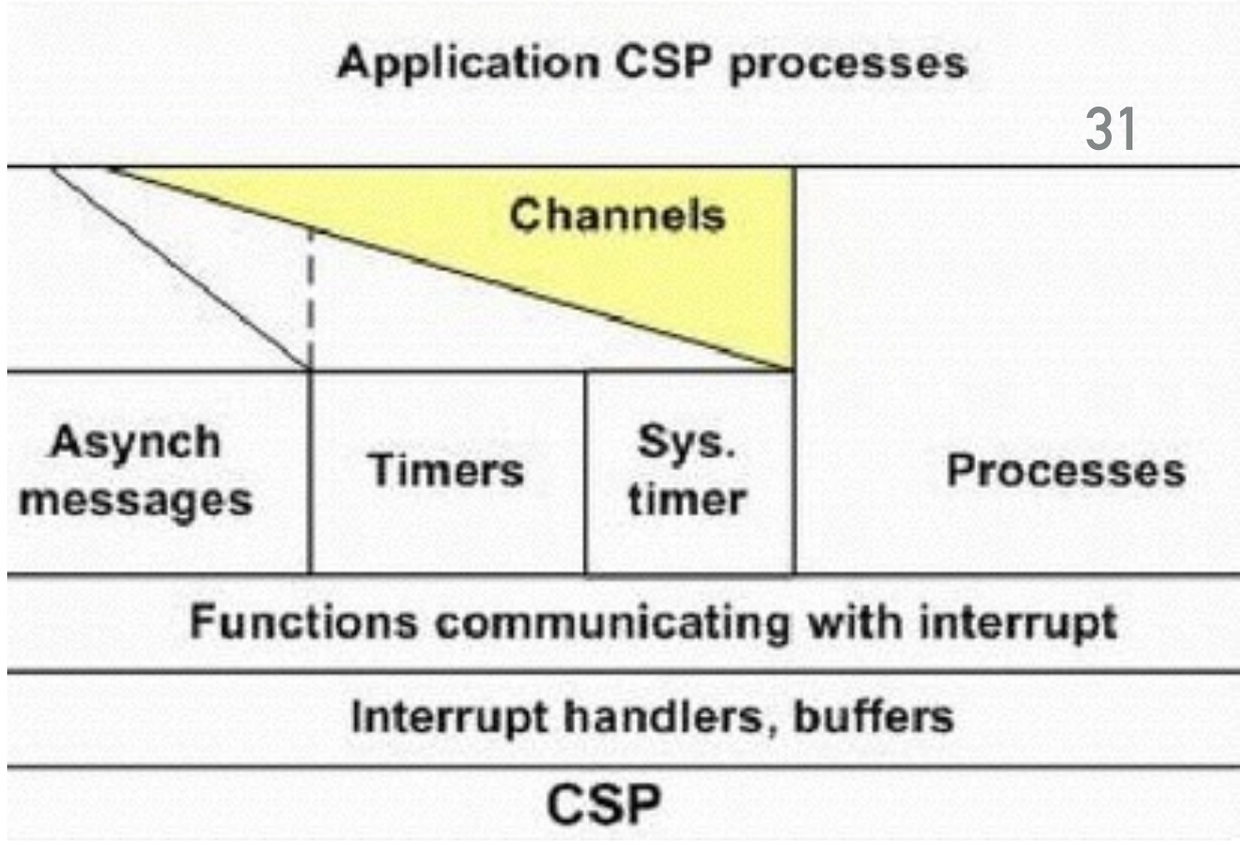


AutroKeeper with Atmel AVR Xmega

# SMALL EMBEDDED SYSTEMS

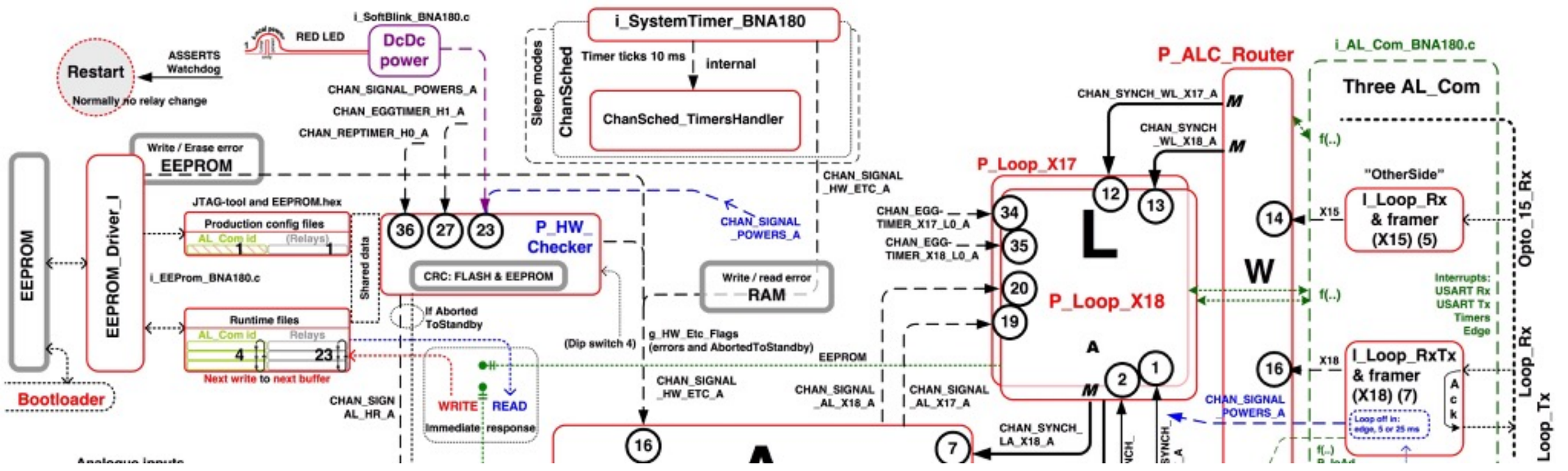
- Will probably keep C for a long time!
- Project managers need to learn about the «Go potential»
- Don’t take over their toolset without adding your knowledge
  - Like channels and tight processes
    - Even if it will be hard to find runtimes. However..



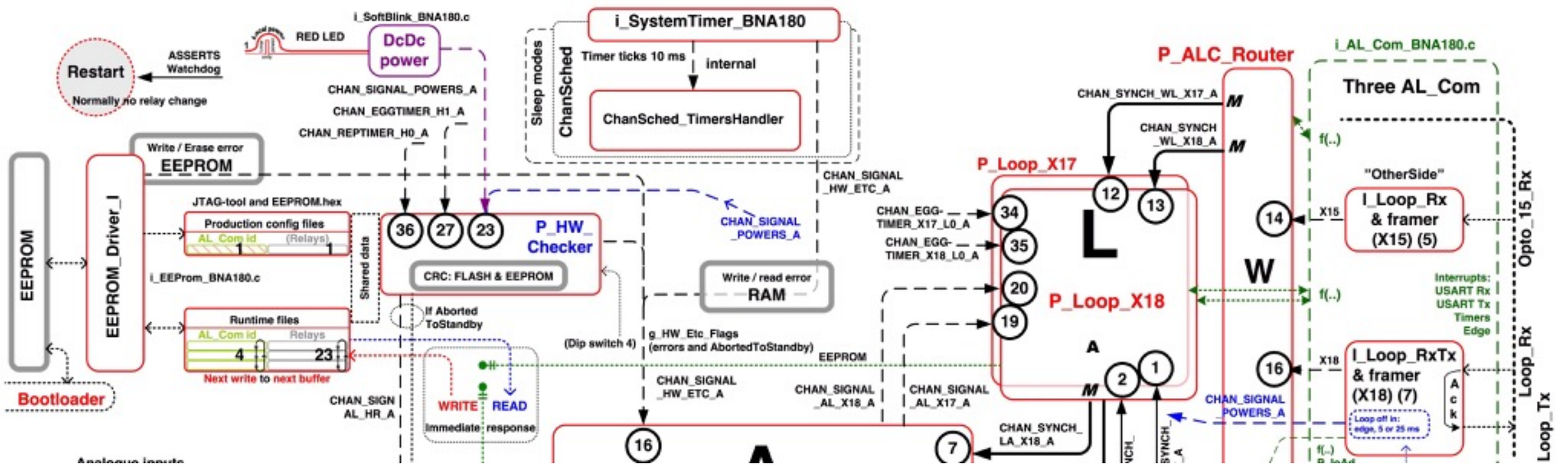

# OUR TWO SOLUTIONS

- FSM scheduler: Most of our controllers use this asynchronous SDL-based scheduler
- CHAN_CSP: However: in two of the controller there's synchronous channels on top of it




# PLUS A THIRD: «CHANSCHED»

- ChanSched: finally in one of the controllers synchronous channels on top of no other runtime («naked»)
- The runtime was more visible to the application code than I thought (later)




Part of process/data flow diagram

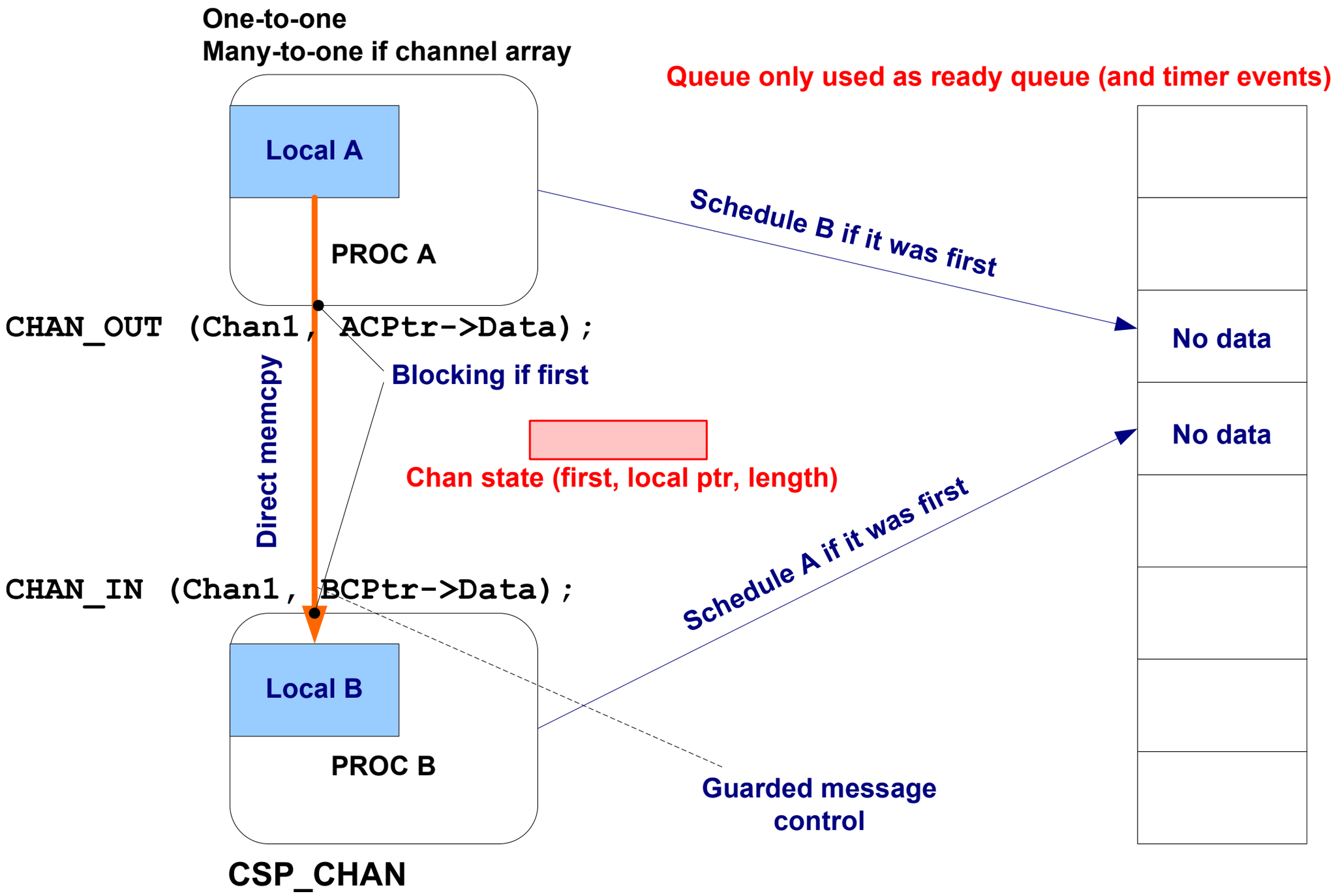
# PLUS A THIRD: «CHANSCHED»

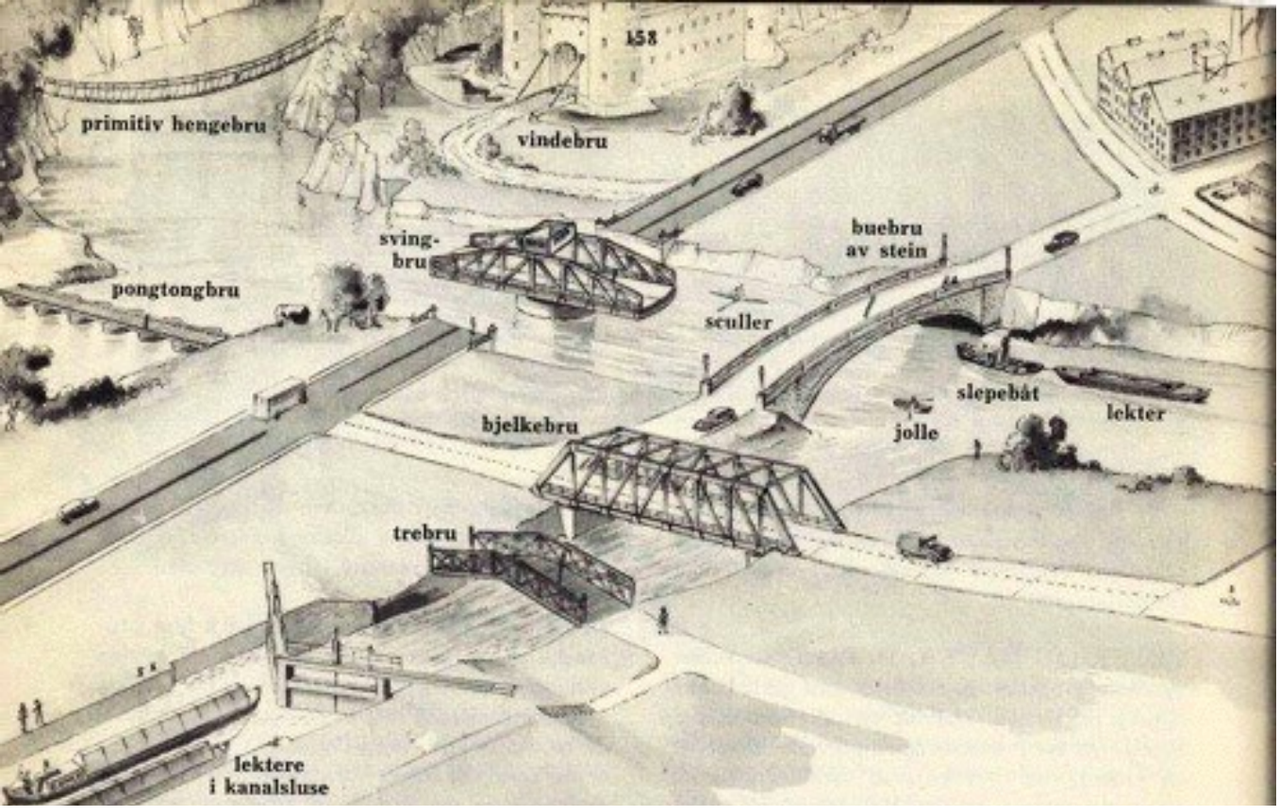
- ChanSched: finally in one of the controllers synchronous channels on top of no other runtime («naked»)
- The runtime was more visible to the application code than I thought (later)

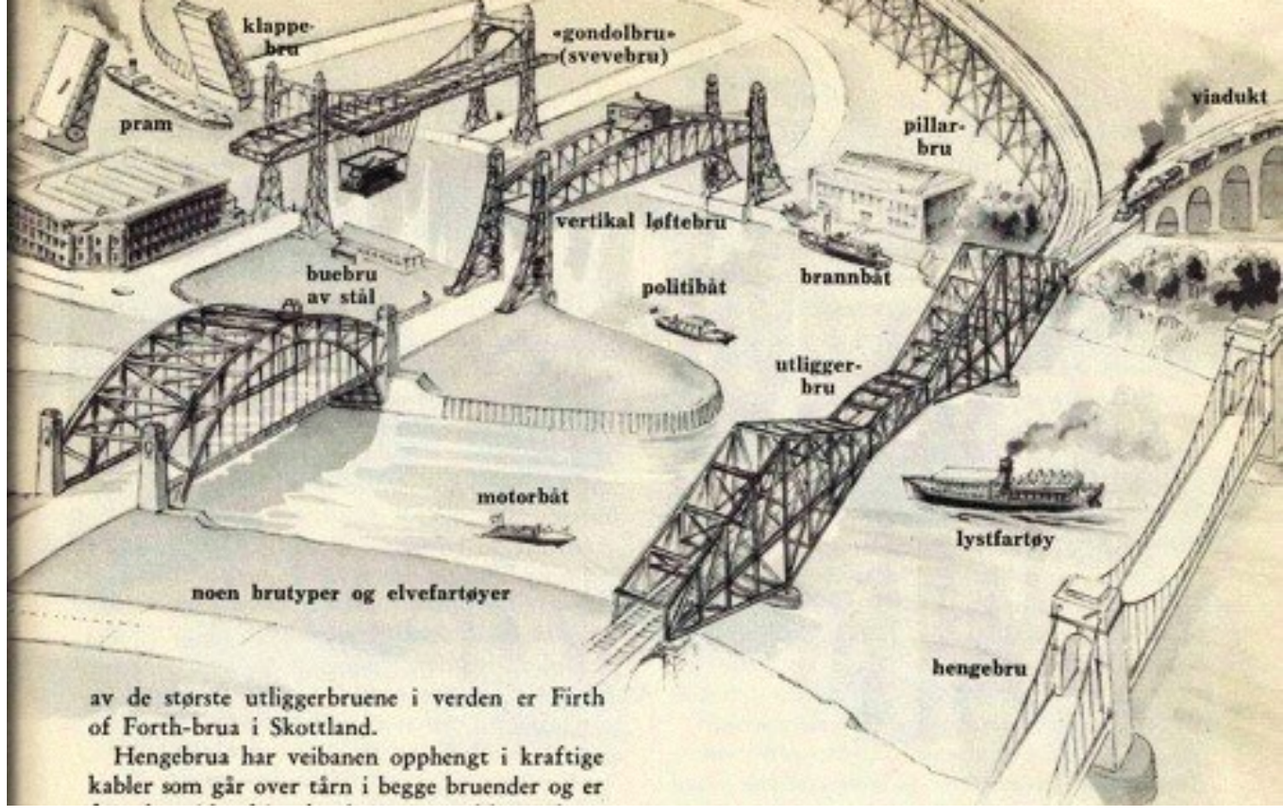
# SAFE MEMCPY, NO POINTERS TO SHARED DATA

```
CHAN_OUT (Chan1, ACPtr->Data);




CHAN_IN (Chan1, BCPtr->Data);
```

# SAFE MEMCPY, NO POINTERS TO SHARED DATA






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

# BRIDGING THE WORLD

- Some road bridges have access control
- Waiting ships and waiting cars are orthogonal
- Some bridges are for cars, some for trains
- Some bridges are tall enough to let most ships through
- Which part of this drawing might most resemble a CSP type system? (Even if CSPm may model everything)


# THE CASTLE AND DRAWBRIDGE

- Process has itself full control of when it it takes new input
- When the drawbridge is raised work inside is undisturbed
- Internal rooms also have drawbridges (doors)
- How would life have been if no drawbridge or doors?
- The metaphor doesn’t match drawbridge outputs well


# A CANAL LOCK HAS SEMANTICS

- Ship in one direction per turning
- The lock keeper operates it
- It has states (because it has)
- Channels, buffers, queues, pipes also have their semantics
- Even if there is so little buffering above several centuries survived well with thes canal locks

# SYNCHRONOUS VS. ASYNCHRONOUS(?)

- It's about fulfilling the requirement, not «sync vs. async»
- Synchronous and asynchronous methodologies are tools
- In the «CSP world» it mostly goes like this:
  - Processes are abundant, not single-threaded
  - Channels are zero-buffered by default (synchronous)
  - Use buffered channels if red blocking might happen
  - Asynch design at the edges; synchronous inside

Rob Pike


Google I/O 2012 - Go Concurrency Patterns

C++ and Beyond 2012: Herb Sutter - C++ Concurrency

Posted: Jan 04, 2013 at 7:42 AM

By: Charles


Discussed in a blog note of mine: «**Pike & Sutter: Concurrency vs. Concurrency**»

http://www.teigfam.net/oyvind/home/technology/072-pike-sutter-concurrency-vs-concurrency/

## Rob Pike

(transcript)

"Now for those experts in the room who know about buffered channels in Go – which exist – you can create a channel with a buffer. And buffered channels have the property that they don't synchronize when you send, because you can just drop a value in the buffer and keep going. So they have different properties. And they're kind of subtle. They're very useful for certain problems, but **you don't need them**. And we're not going to use them at all in our examples today, because I don't want to complicate life by explaining them."

## Herb Sutter

(collection of  my scribblings)

Queue is way more scalable because you don't wait. Don't stop! We hate to block! Blocking is almost always harmful. You can always turn blocking "the bad thing" into non-blocking "the good thing" via async() at the cost of occupying a thread. Anything shared is evil. **Blocking code is not good for the universe**.

# Which block ing do you mean?

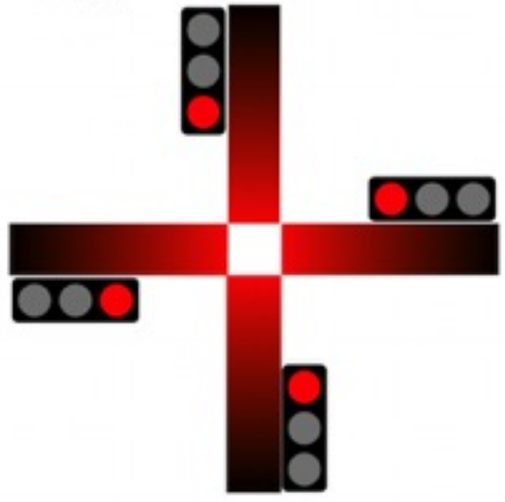
= yielding?

The show goes on with this blocking

= blocking?


This blocking stops the show

= deadlock!



This blocking stops the world



# «BLOCKING» CHANNEL

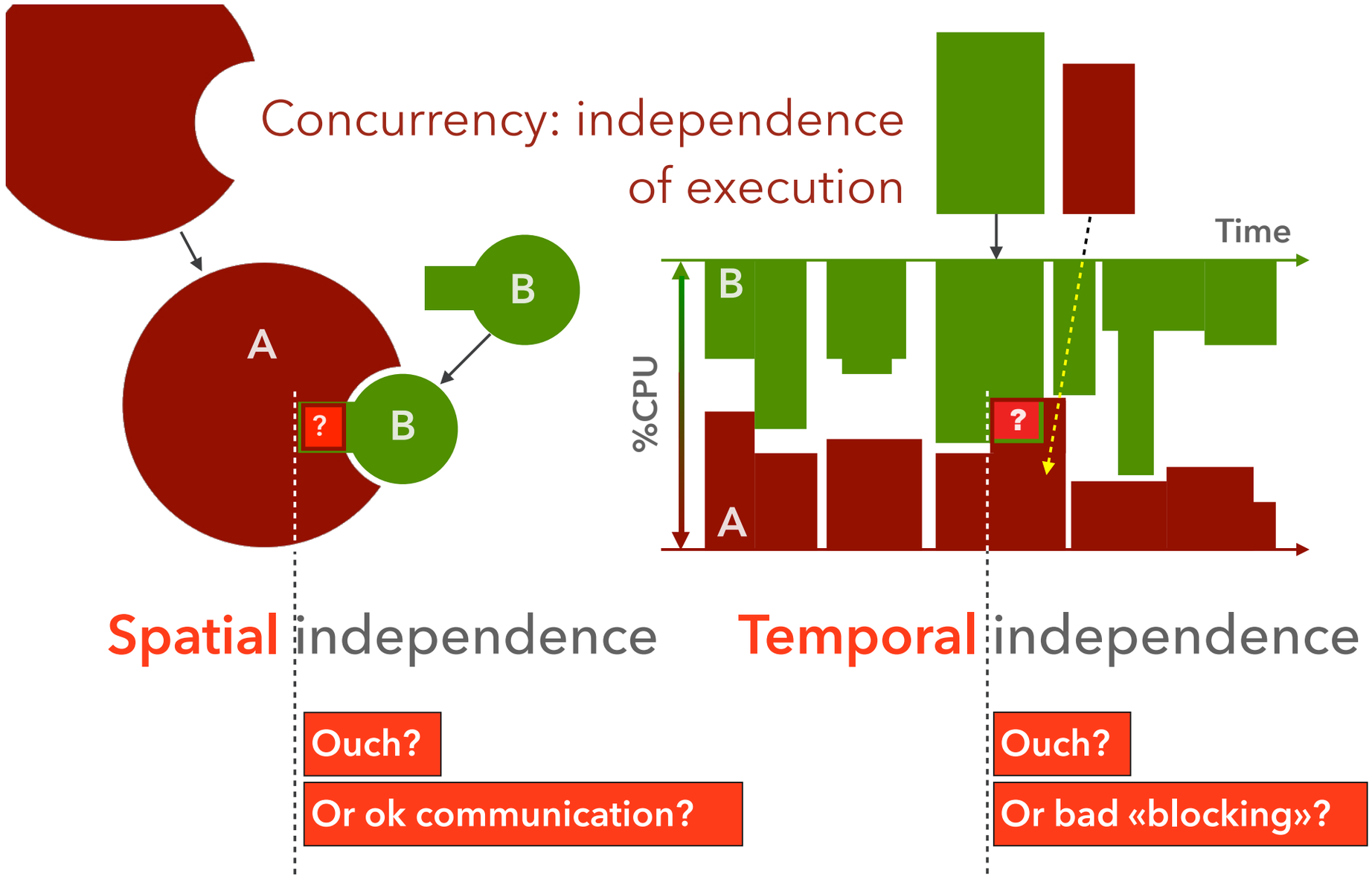
- The green channel «blocking» is normal waiting
  - Still called «blocking semantics»
- The red «blocking» is blocking of others that need to proceed according to specification (too few threads?)
- The black blocking is deadlock, pathological, system freeze

# THE PROGRAMMING MODEL

- Event loop and callbacks
  - Threading often creeps in: problems (shared state, nesting)
- Channels and conditional choice (select, alt)
  - In proper processes, concurrency solved
- Connecting channels to event loops and callbacks when that's what you have in a library (like in Closure core.async, see Further reading)
- At Autronica: event loops - callbacks, messages and channels

# INDEPENDENCE BETWEEN SAFE AND (UN!?)SAFE SW

An attempt at drawing it:




Worries: deadline, deadlock, race, safety, livelock, «blocking», (non)determinism

# C CODE ON TOP OF ASYNCH RUNTIME (LEFT) AND NAKED (RIGHT)

```
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;
    }

  }
}
```

Sync chan comm needs states

```
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;
      }
    }
  }
}
```

Synchronization points no visible state

# C CODE ON TOP OF ASYNCH RUNTIME (LEFT) AND NAKED (RIGHT)

```
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;
    }

  }
}
```

Sync chan comm needs states

```
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;
      }
    }
  }
}
```

Synchronization points no visible state

Same

# A TYPICAL ChanSched PROCESS BODY (OVERVIEW)

```
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. }
```

# SETTING OF TIMERS (LIKE DATA, PICKED UP BY CHANNELS)

```
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. }
```

# THE while(FOREVER) LOOP

```
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. }
```

# THE ALT CONSTRUCT THEN switch/case ON INPUT

```
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. }
```

# THE `synchronisation` POINTS AND HIDDEN JUMP TABLE

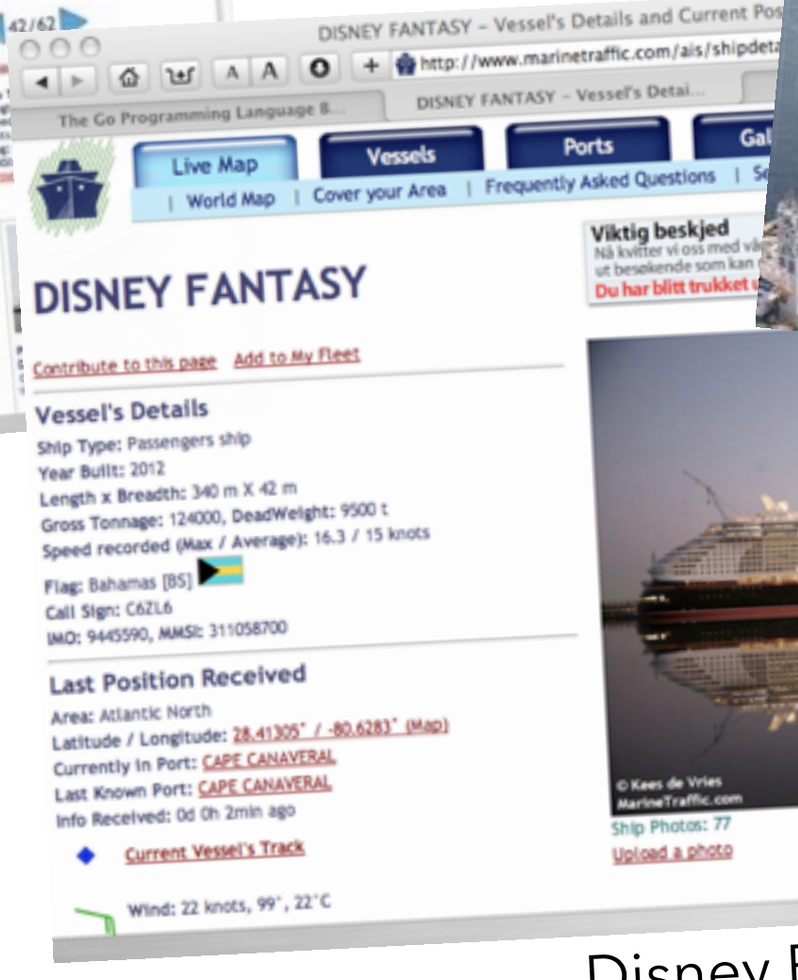
```
1.  Void P_Prefix (void)                        // extended "Prefix"
2.  {
3.    Prefix_CP_a CP = (Prefix_CP_a)g_CP; // get process Context from Scheduler
4.    PROCTOR_PREFIX()                          // jump table (see Section 2)
5.    ...  some initialisation
6.    SET_EGGTIMER (CHAN_EGGTIMER, 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. }
```

# A TYPICAL ChanSched PROCESS BODY (SUMMARY)

```
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. }
```

# «SAFE RETURN TO PORT» (IMO) OR JUST EXTRA SAFETY

Disney Dream (2011)

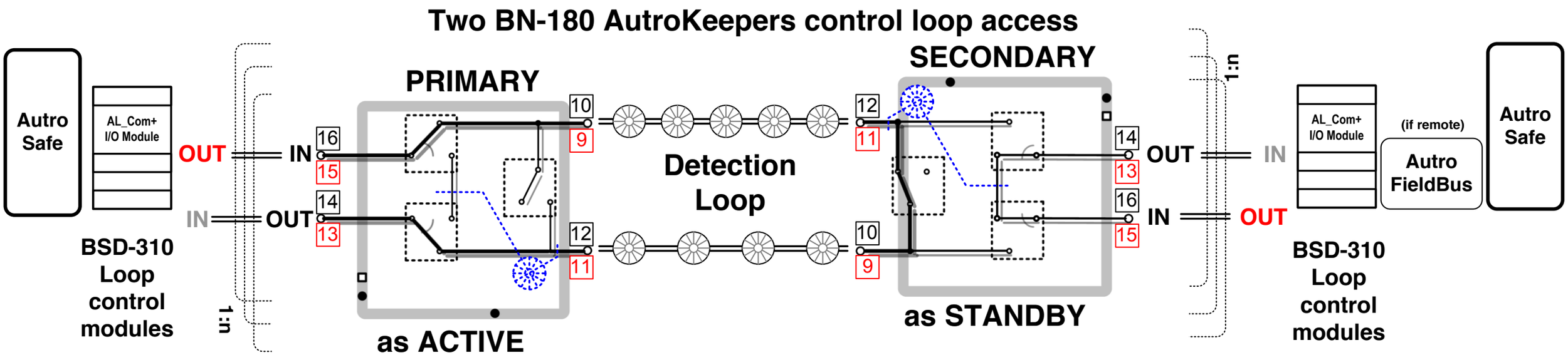




Pioneering Spirit (2013)

++

Disney Fantasy (2012)

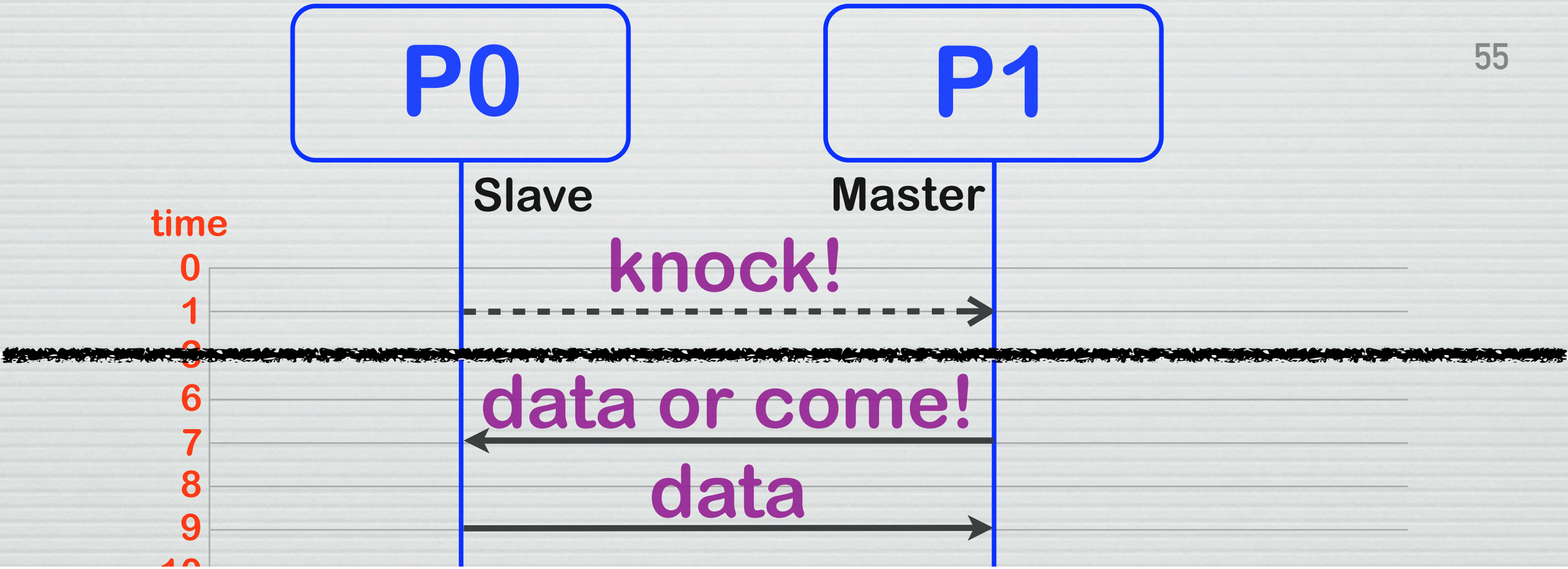
**AutroKeeper: patent 329859 in Norway,**
**PCT/NO2009/000319 international (granted as #2353255)**




# FORMAL MODELLING OF ROLES

- Safe, not simultaneous dual access of detector loop
- Always one side connected
- No oscillations
- Keeps track of the sanity and possibilities of each side
- Switches over in milliseconds when needed
- Formal model gave us roles and protocol elements

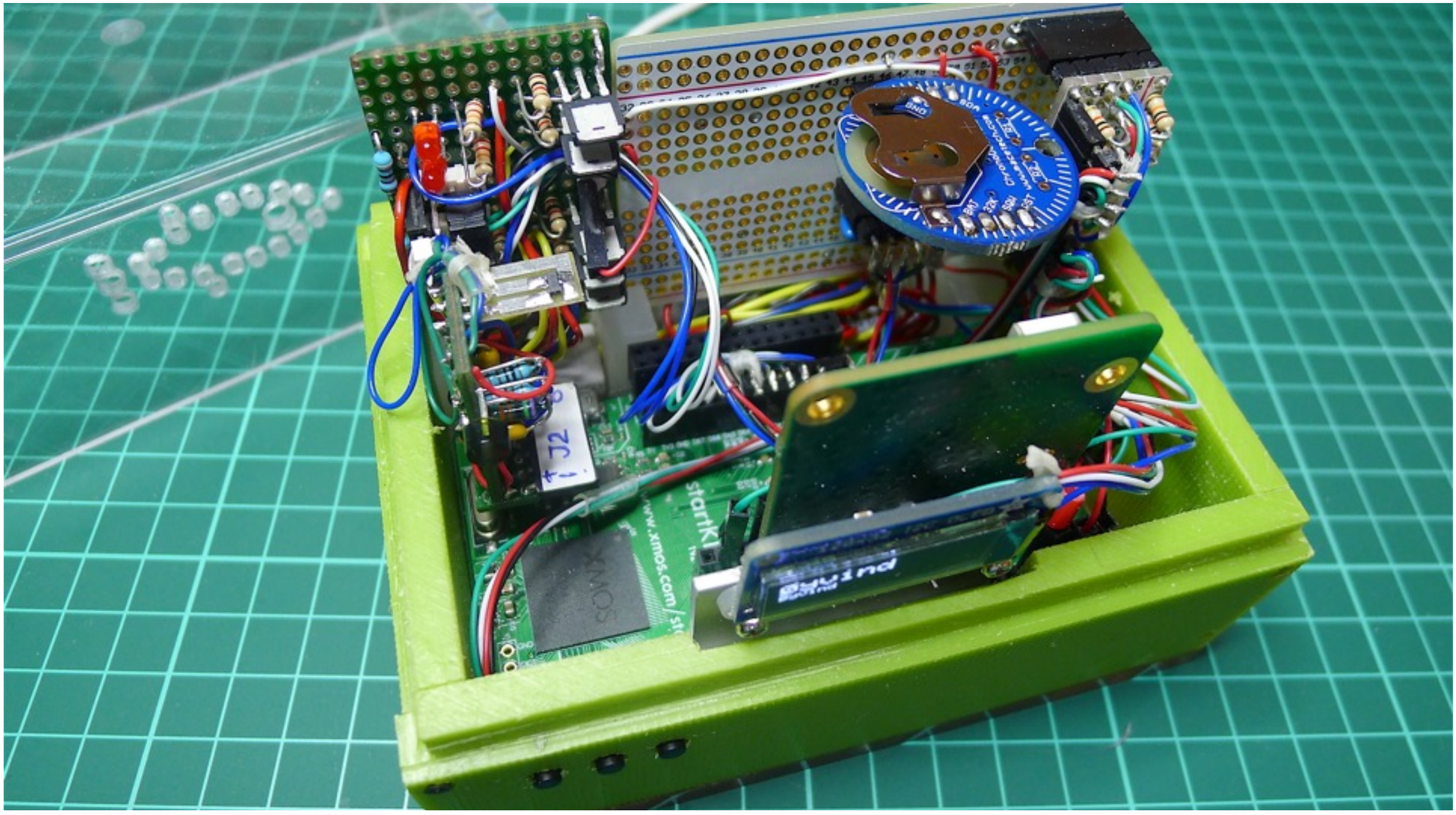
produkt.autronica.no/fileshare/filArkivRoot/produkt/pdf/dokumentasjon/bn180_cn.pdf




# KNOCK-COME, THEN DATA

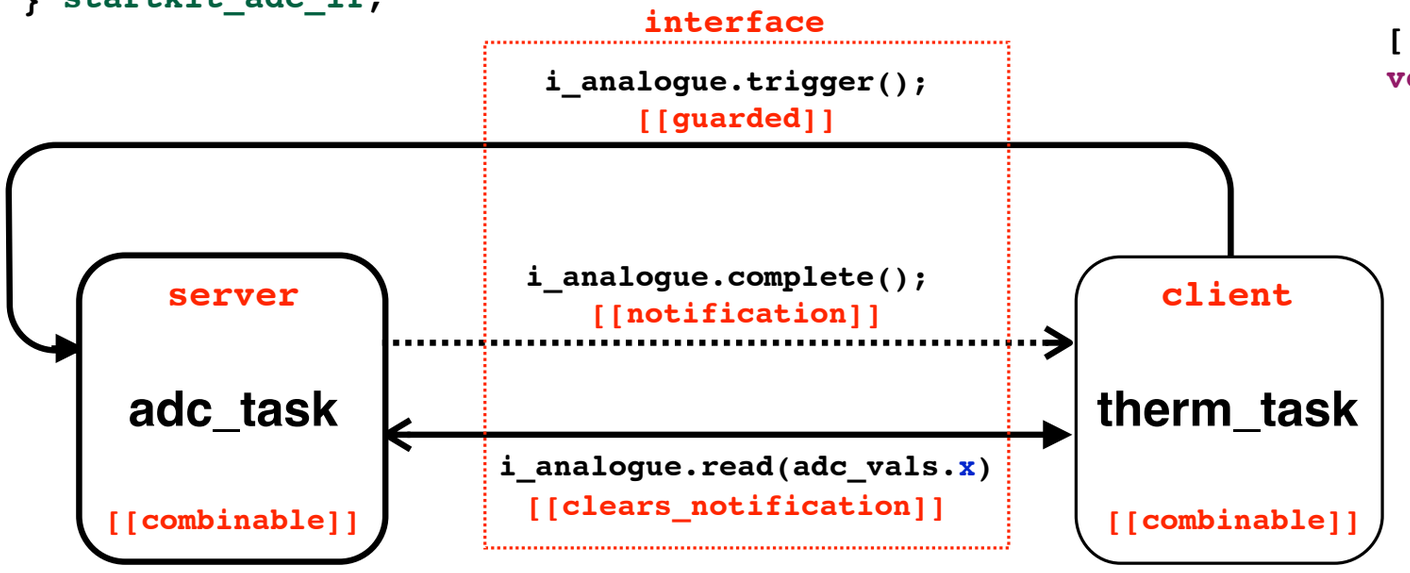
- Deadlock free communication pattern
- Master can send data any time
- Slave must «knock» (asynch signal channel, no data, doesn't block)
- Master replies with «come»
  - after any time or any number of messages to slave
- Slave then must reply immediately with the data
  - since Master waits for it
- If channels were buffered, such a pattern is still needed at the edges for the possible full situation
- XCHAN (paper by me) is a suggestion of a CHANnel type that does this, plus more

# AQUARIUM CONTROL UNIT WITH XMOS `STARTkit`, 8 LOGICAL CORES IN `xC`



# KEYWORDS interface, server, client AND slave etc.

```
typedef interface startkit_adc_if {
  [[guarded]]                 void trigger(void);
  [[clears_notification]]     int read(unsigned short
adc_val[4]);
  [[notification]]            slave void complete(void);
} startkit_adc_if;
```



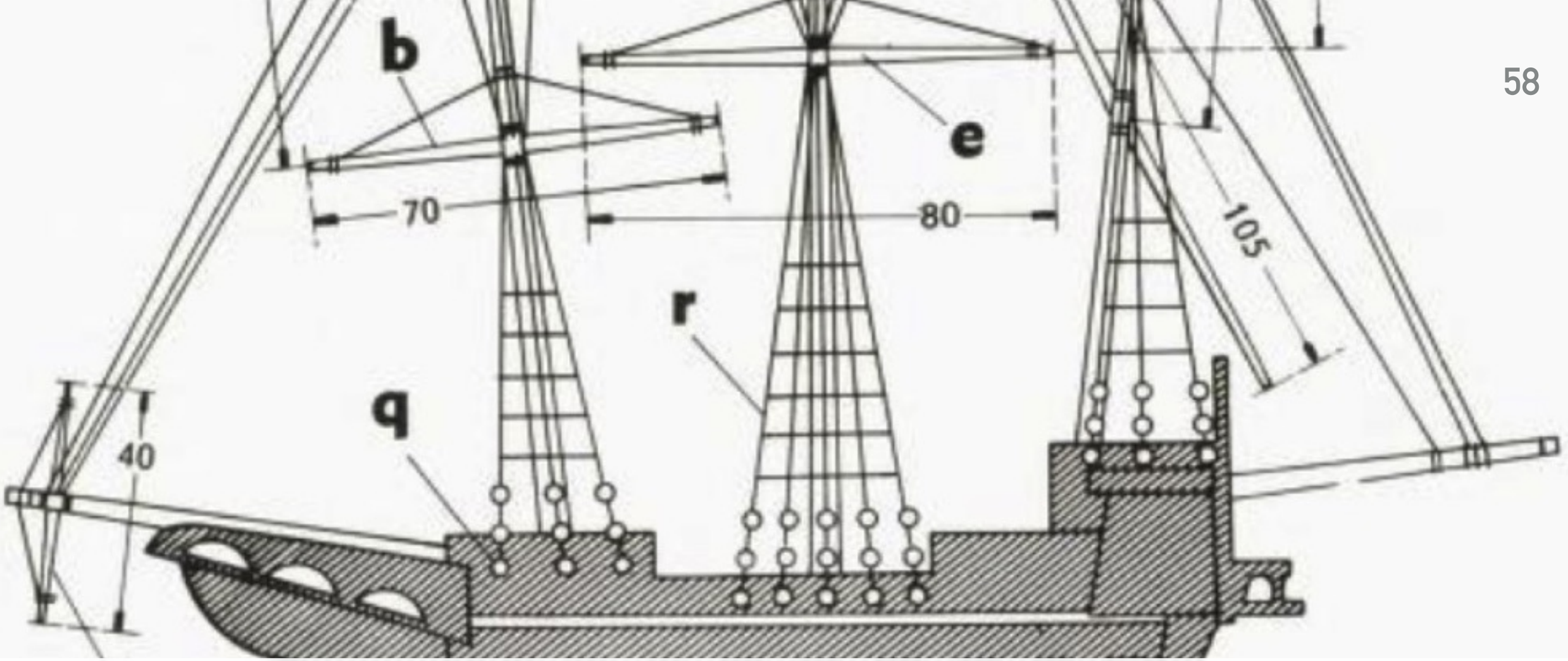

Also has traditional `chan` (untyped)
Guaranteed deterministic real-time response

```
interface startkit_adc_if i_analogue;
```

```
[[combinable]]
void therm_task
  // ...
  while(1) {
    select {
      case wait_for_button => c_button_2 :> int x: {
        // ...
        i_analogue.trigger();
        break; }
      case wait_for_adc => i_analogue.complete(): {
        // ...
        if (i_analogue.read(adc_vals.x)) {
          // Use it
        } break;
      }
    }
  }
}
```

Drawing by Øyvind Teig

- This pattern is understood by the compiler and it is deadlock free
- I have a private goal to learn xC fluently. Multi-core!




# ADVICE

- Make things so well that you can look at it after five years and think it well done
- Watch out that you have moved so much those five years that you might want to make it better now

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

- This lecture
  http://www.teigfam.net/oyvind/pub/NTNU_2016/foredrag.pdf
- This course
  NTNU, TTK4145 Sanntidsprogrammering (Real-Time Programming) http://www.itk.ntnu.no/fag/TTK4145/information/
- My blog
  http://www.teigfam.net/oyvind/home/technology/

- **Bell Labs and CSP Threads**
  by Russ Cox at https://swtch.com/~rsc/thread/, referred at one of my blog notes: http://www.teigfam.net/oyvind/home/technology/072-pike-sutter-concurrency-vs-concurrency/

- **Clojure core.async**
  Lecture (45 mins). Rich Hickey explains callback and event loops vs. processes, select and channels at http://www.infoq.com/presentations/clojure-core-async

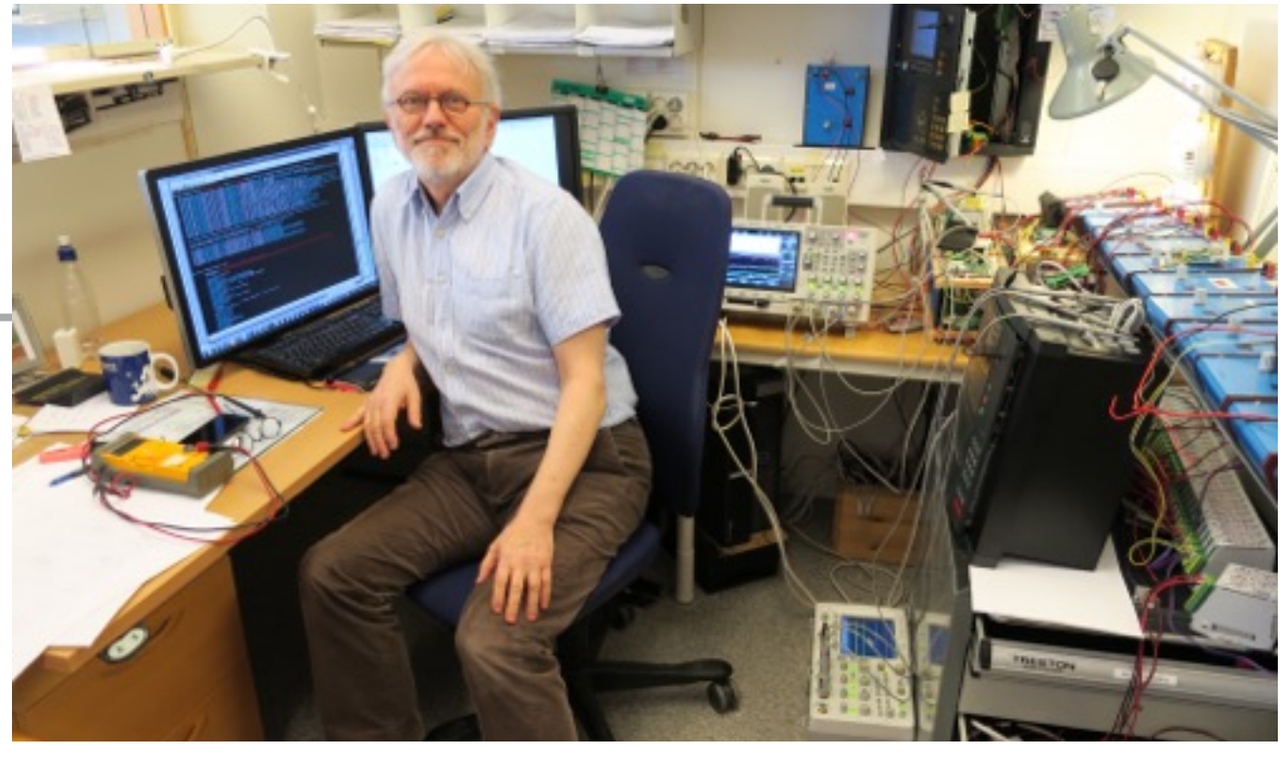
- **New ALT for Application Timers and Synchronisation Point Scheduling**
  CPA-2009. Per Johan Vannebo, Øyvind Teig. Read at http://www.teigfam.net/oyvind/pub/pub_details.html#NewALT. About ChanSched

# REAL TIME GOES SO FAST

All of a sudden you may be grandparents, too!

40 years

1976 2016



All of a sudden you may be grandparents, too!


40 years

2016 2056

# Questions?

# Thank you!