Designing with Freescale
-
Embedded Linux for i.MX Applications Processors - Real-World Projects

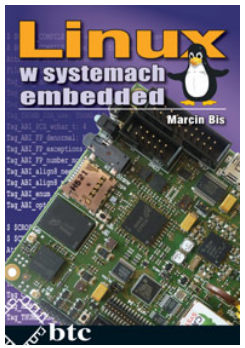
Marcin Bis

http://bis-linux.com
martin@bis-linux.com

Warsaw, Poland - June 26th 2013

About me

- **Marcin Bis**
- Embedded Linux (from administration to programming: system, kernel, device drivers)
- Trainings, consulting, support (`http://bis-linux.com`)
- Industrial appliances (Linux + Real-Time)



Goals

- To get familiar with Embedded Linux @ **Freescale** platforms.
- To provide an in-depth uderstanding of Embedded Linux, and Real-Time derivatives of Linux.
- To enable **You** to put together a working Embedded Linux system with "rich GUI" application.
- To teach how an Open Source softare is developed and how to make advantage of this process.

We will also build a working ICS (industrial control system) - a laser.

Agenda I

1. Why to consider Linux?
2. Let's build ICS
3. What we will need (hardware + tools)?
4. Real-Time
5. ICS using PREEMPT-RT
6. ICS using Xenomai
7. ICS using multiple processors (cores)

BIS-LINUX

Why to consider Linux?

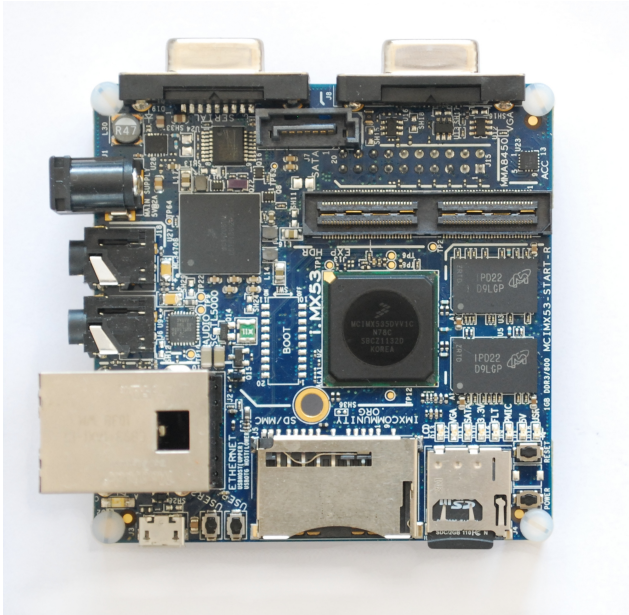
Plan

1 Why to consider Linux?
Pros ...
... and cons

Why to consider Linux?

Booming usage of Linux in embedded environments. From consumer electronics to industrial control devices.

- Hardware capable of running Linux become cheaper.
- Linux supports lots of hardware, as well as communication protocols.
- Security is easy to achieve.
- Code is easy to develop and reusable.



- Android - customized Linux system.

BIS-LINUX

No additional fees: no per-device licence cost, no need to upgrade constantly.

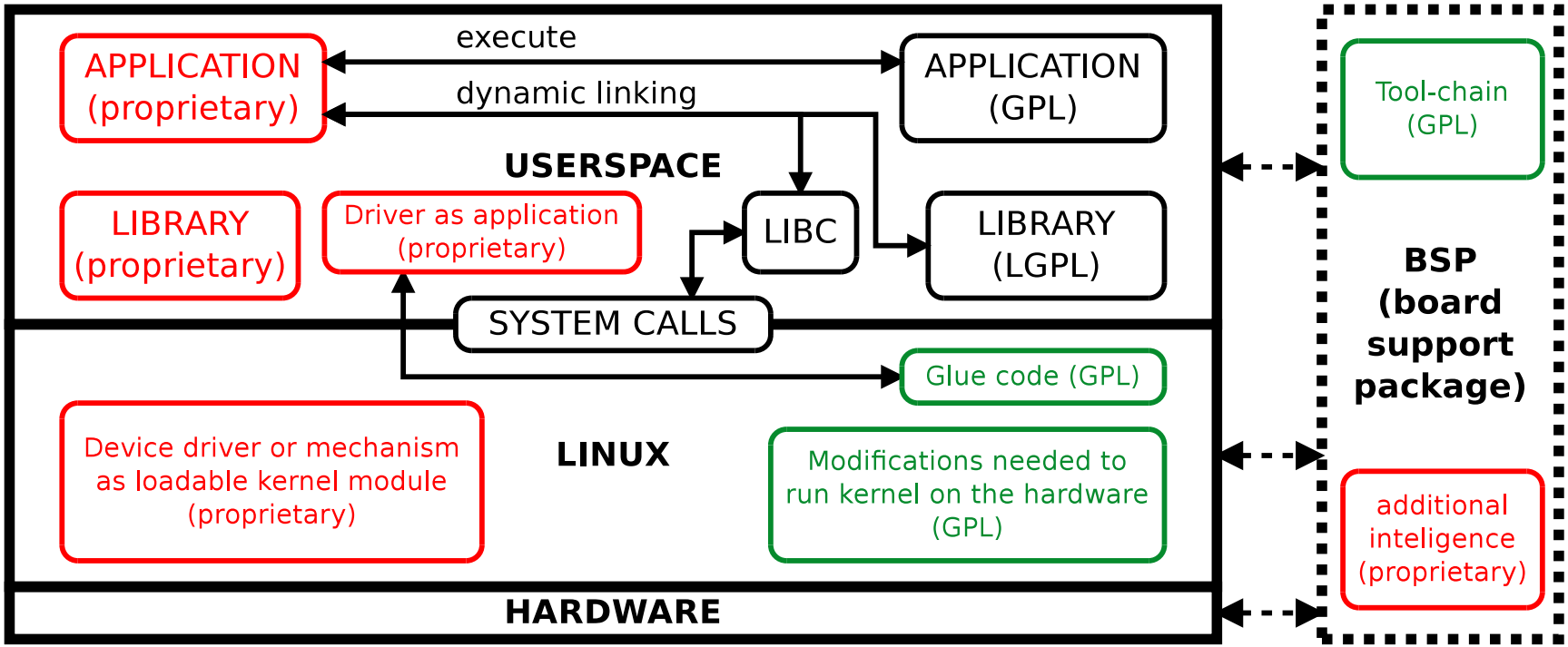
E.g. Availability of the source code, lets you easily upgrade old system/product to the latest protocols and drivers.

Linux support all embedded SoC features. Device connectivity features are implemented in universal frameworks.

- SOC drivers
- I2C, SPI, GPIO
- Flash (including flash-transaction-layers) and flash aware file systems (UBIFS, JFFS2)
- EEPROM
- I/O expanders, custom embedded parts
- networking controllers, including Wireless.

Mixing GPL with proprietary

You can add your proprietary applications, of course. . .



Vast array of development tools.

Do you want to build a customer-programmable device?

Let user program it in C, C++, Java, Python, Perl, PHP, Lua and even more.

Tool-chain can be provided at no additional cost (because it is open-source).

E.g. Python programmed "software PLC" device. Customer likes it to be Java programmable - not a big issue in Linux.

BIS-LINUX

Stable, secure and efficient networking stack.

TCP/IP and variety of application-level protocols.

Easy encryption integration. WiFi, broadband, Bluetooth and other protocols. VNC, Web Server, ftp, SSH...

Examples:

- Consumer device, connected to local network is recognized by Windows and Apple computers as a network neighbourhood party. In order to emulate windows behaviour it runs a cut down SMB, NetBIOS, Upnp stack.
- Device shall be easily accessible from smartphone (web browser), using secure protocol as well as a VPN connection.
- Secure (encrypted) firmware update over the network - no more STUXNET.

Portability

Design and write our software once. . . compile, run and deploy everywhere.

- Application is Linux-base rather than microcontroller-based.
- Can be developed and tested on a PC.
- There is variety of available libraries and options.
- Easy: GUI, input and display support.

You are no longer facing vendor lock-in!. Software can be easily ported to another Linux SBC.

E.g. Demonstrate application running on RaspberryPI, as well as on some more secure and manufacturing friendly platform.

If application specification needs Real-Time response. There are a few ways to transform a Linux system into Real-Time operating system.

Scalability

Easily implement customers demand. Even one-time wishes.

Lower cost of developing one-time software variants:

- Let's use twice as big screen.
- Or add barcode reader and rotary encoder.
- We want to control device remotely via IP connection.
- . . . but customer's network is running IPv6.
- OK, and now let's display user interface in Russian/Chinese

Caveats and disadvantages

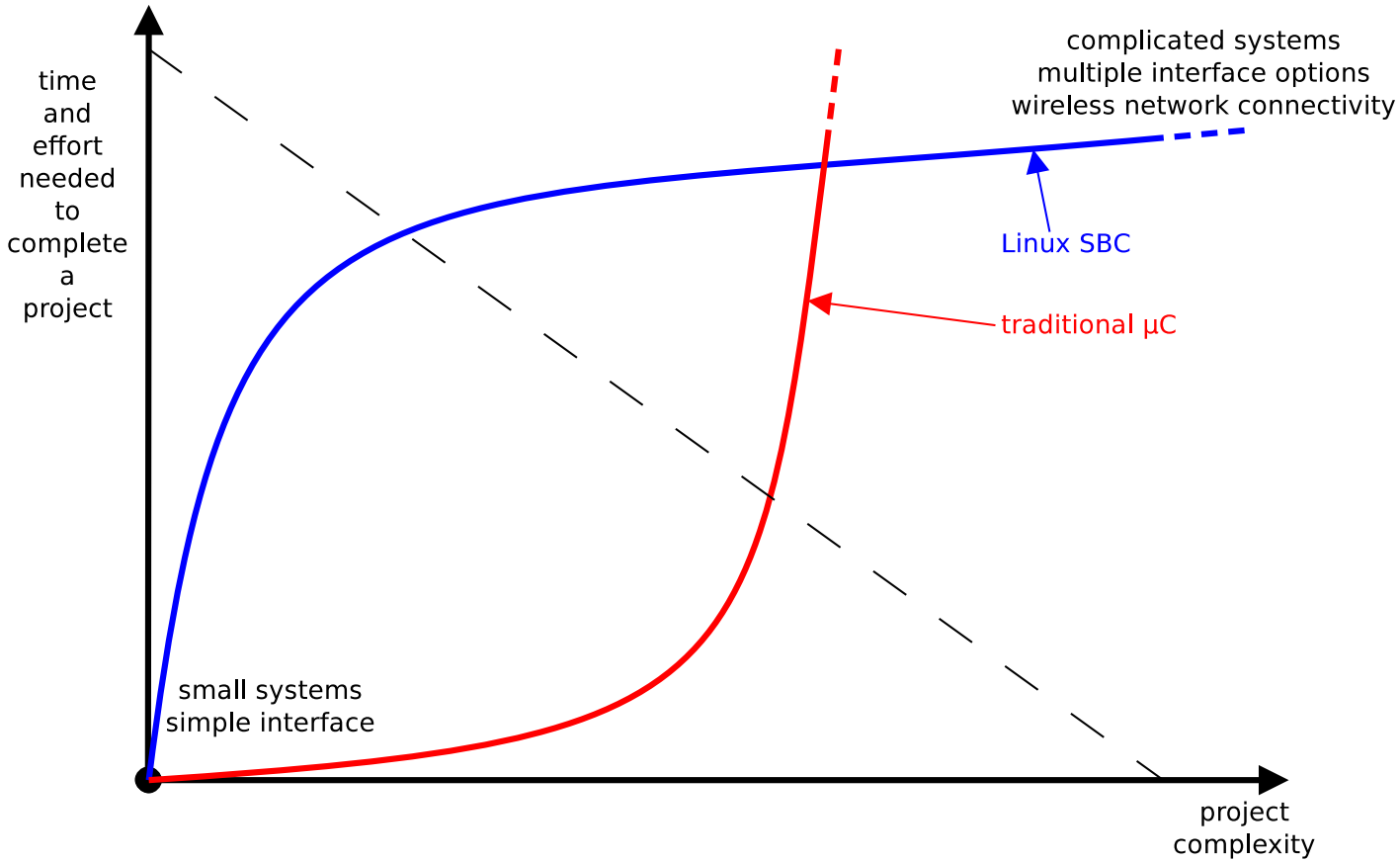
BIS-LINUX

You:

- Learning curve.
  There is a great amount of knowledge to get familiar with. Variety of skills: from electronics design to system administration is needed to make a "helicopter view of a system".
- Fragmented knowledge.
  Linux is a go-fast technology rather than finished project. There is no central knowledge base nor single vendor providing support (actually there are many of them). Relevant support can be difficult to find on the Internet or applicable only to specific version of source code.

Learning curve

Simple project: bare-processor or RTOS applications are easy to establish - just install BSP and get familiar with documentation.



Linux project: knowledge of the whole software stack is needed. But, afterwards, adding sophisticated features is very easy.

Caveats and disadvantages

BIS-LINUX

Your Product:

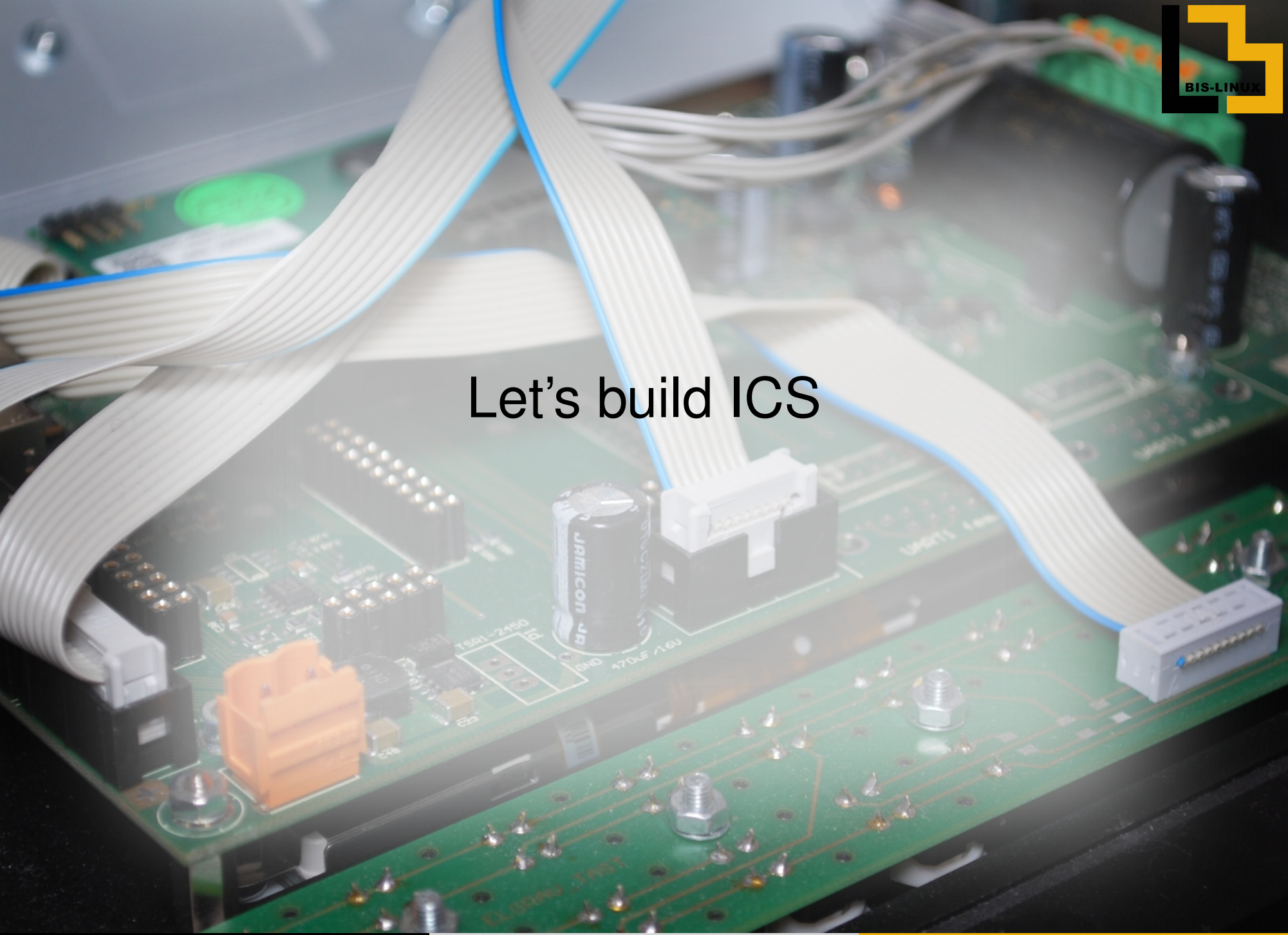
- Memory footprint.
  Power and space limitations. Microcontroller systems used for many tasks has (e.g.) 16kB of RAM and 128kB of Flash in single chip. Typical universal Linux SBC with network connectivity and full user interface has 64MB of RAM and 256MB of Flash. Even small Linux system needs 32MB RAM/8MB Flash.
- Development Tools.
  Most Linux system development and debugging can be done using Eclipse, although there is no install and ready solution (like tools for various microcontrollers). IDE has to be properly configured by a user.

BIS-LINUX

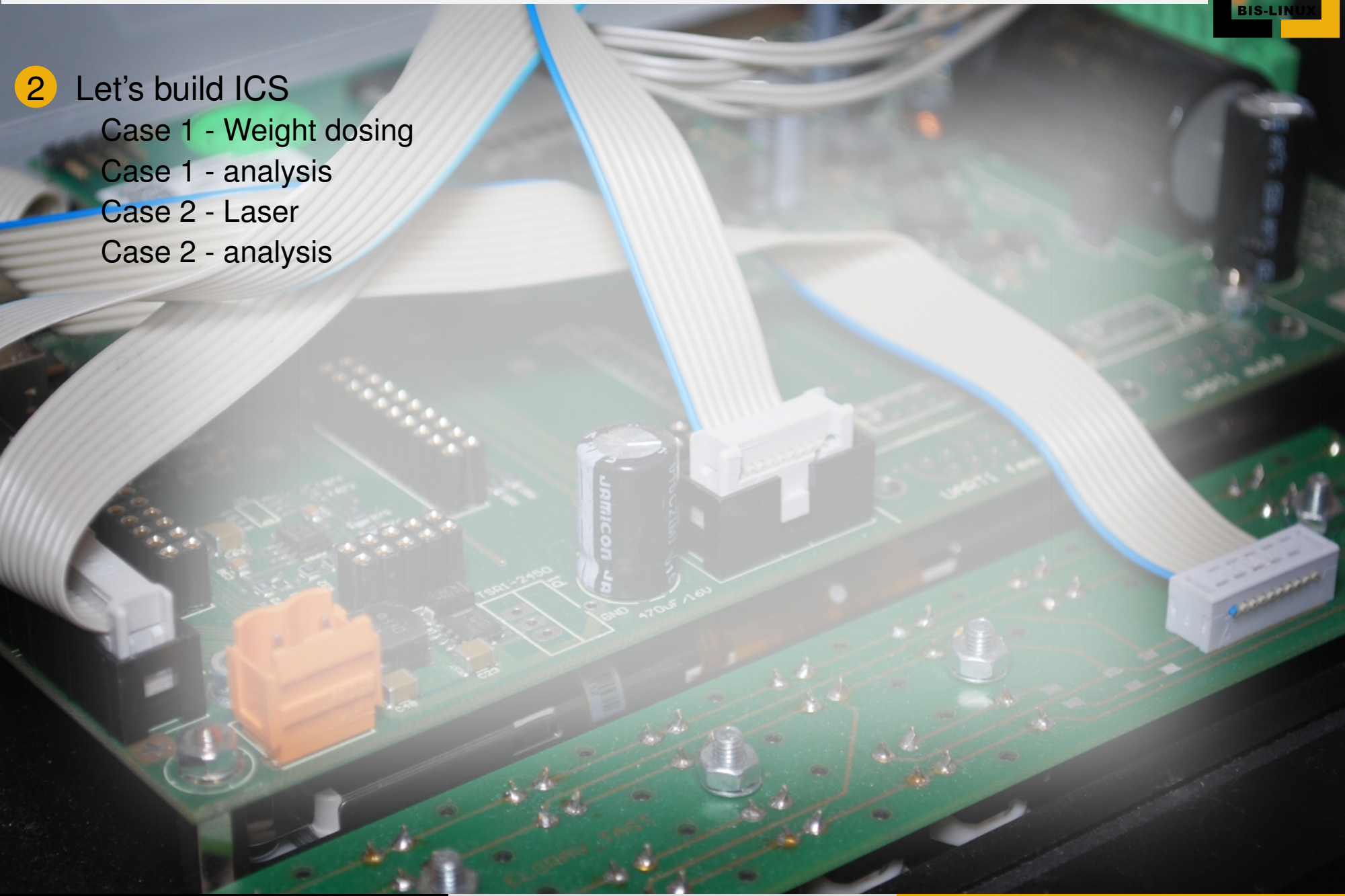
- So: is Linux the answer for You?

- No! Linux is a question to consider.

Definitively worth considering.


BIS-LINUX
Let’s build ICS
JAMICON JR
470uF /16U
BND

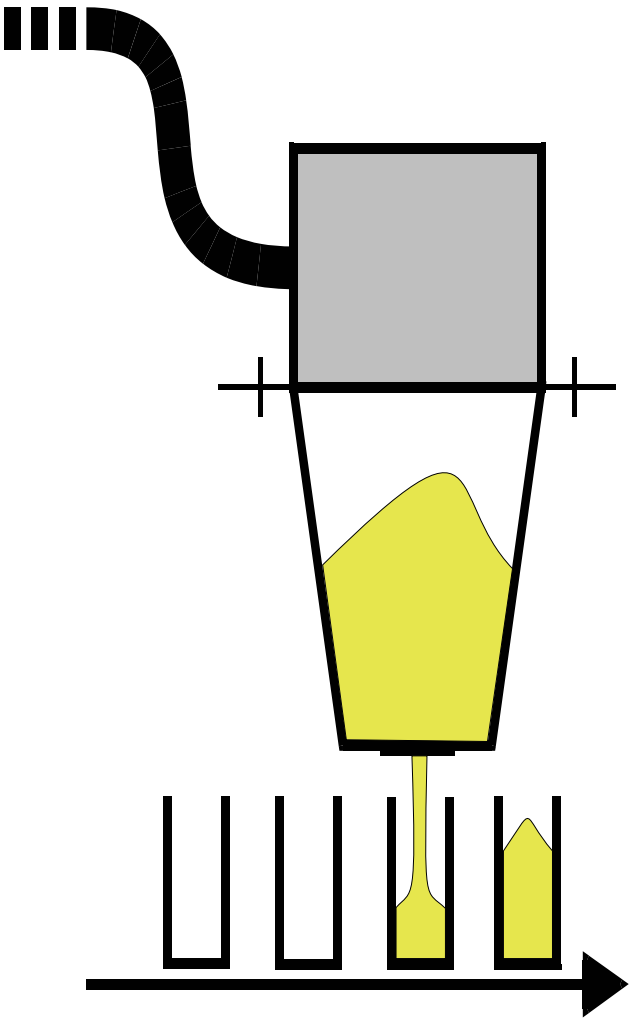
Plan



2. Let's build ICS
   - Case 1 - Weight dosing
   - Case 1 - analysis
   - Case 2 - Laser
   - Case 2 - analysis

Weight-dosing process - specification

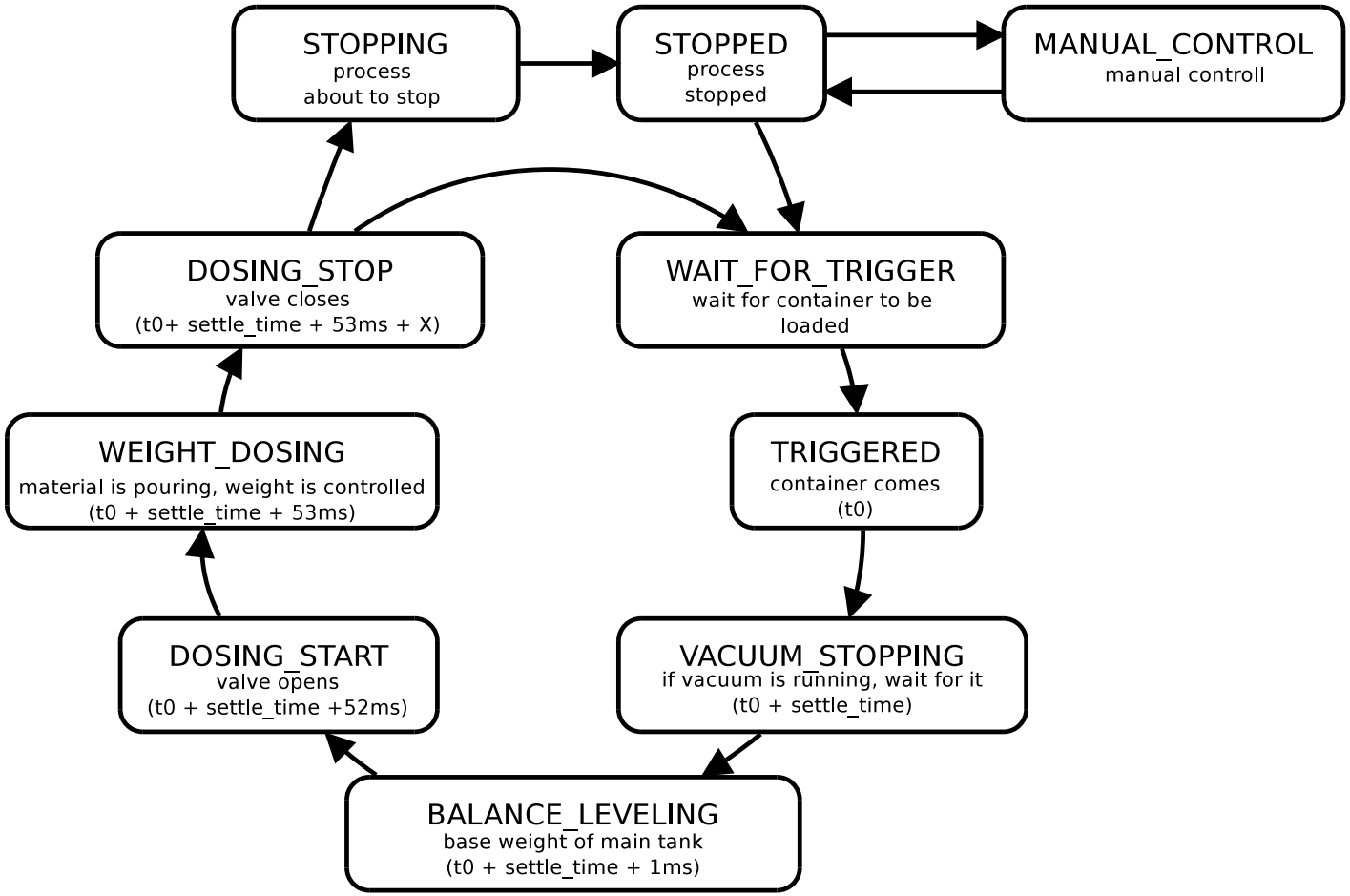
Loose material (or fluid)
is loaded into containers.



- the main tank is suspended on weight (tensometer)
- conveyor or robot provides containers, appearance of the container triggers interrupt
- the valve opens, and the material is poured into a container
- amount of material is measured by reading data from the weight
- main tank has a limited capacity, it can be replenished from main silo by turning on vacuum
- if vacuum is turned on, it has to work for some minimum time, while vacuum is working, material cannot be poured.

Weight-dosing process - analysis

Weight dosing process can be modelled as a finite state machine.



Process starts on `WAIT_ON_TRIGGER` state. If triggered, it runs on timer (1ms).

Weight-dosing process - implementation goals

BIS-LINUX

Hardware

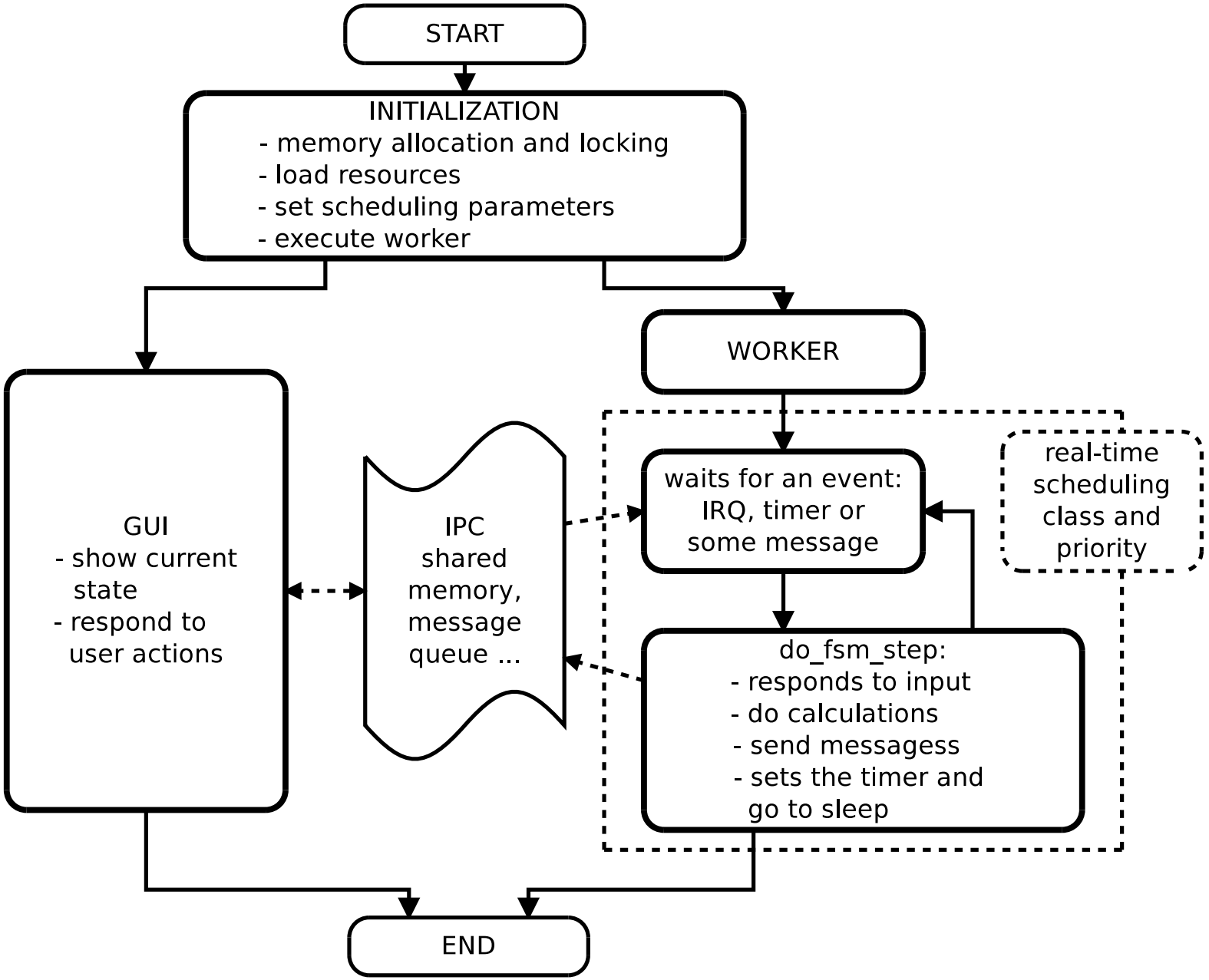
- PC for development, Ubuntu 12.04
- **Freescale** Board

GUI part

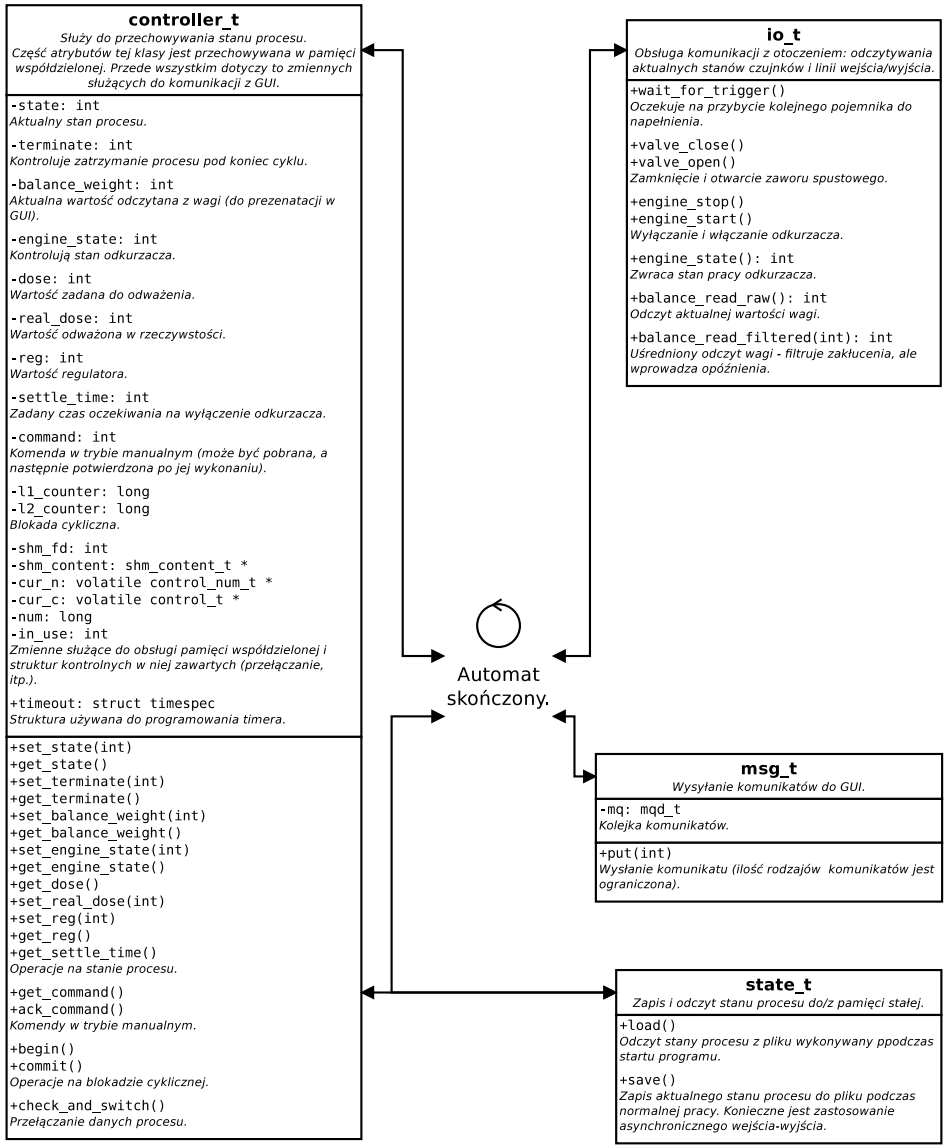
- written in QT/C++
- userspace components provided by: Buildroot or LTIB or YOCTO or something else...
- ext4 on SD card as primary storage or NAND flash storage

Real-Time operations

- implemented as separate process in C
- which allows easy porting
- communicates with GUI using shared memory and message queue, in a lockless way:
  - two control structures are stored in SHM
  - one is utilized by running process, other can be changed by GUI
  - then structs are switched



System boot constraint: must be operational under 10s (2.5s RT task, 8-9s GUI).


controller_t
Służy do przechowywania stanu procesu.
Część atrybutów tej klasy jest przechowywana w pamięci współdzielonej. Przede wszystkim dotyczy to zmiennych służących do komunikacji z GUI.
-state: int
Aktualny stan procesu.
-terminate: int
Kontroluje zatrzymanie procesu pod koniec cyklu.
-balance_weight: int
Aktualna wartość odczytana z wagi (do prezenatacji w GUI).
-engine_state: int
Kontrolują stan odkurzacza.
-dose: int
Wartość zadana do odważenia.
-real_dose: int
Wartość odważona w rzeczywstości.
-reg: int
Wartość regulatora.
-settle_time: int
Zadany czas oczekiwania na wyłączenie odkurzacza.
-command: int
Komenda w trybie manualnym (może być pobrana, a następnie potwierdzona po jej wykonaniu).
-l1_counter: long
-l2_counter: long
Blokada cykliczna.
-shm_fd: int
-shm_content: shm_content_t *
-cur_n: volatile control_num_t *
-cur_c: volatile control_t *
-num: long
-in_use: int
Zmienne służące do obsługi pamięci współdzielonej i struktur kontrolnych w niej zawartych (przełączanie, itp.).
+timeout: struct timespec
Struktura używana do programowania timera.
+set_state(int)
+get_state()
+set_terminate(int)
+get_terminate()
+set_balance_weight(int)
+get_balance_weight()
+set_engine_state(int)
+get_engine_state()
+get_dose()
+set_real_dose(int)
+set_reg(int)
+get_reg()
+get_settle_time()
Operacje na stanie procesu.
+get_command()
+ack_command()
Komendy w trybie manualnym.
+begin()
+commit()
Operacje na blokadzie cyklicznej.
+check_and_switch()
Przełączanie danych procesu.
Automat skończony.
io_t
Obsługa komunikacji z otoczeniem: odczytywania aktualnych stanów czujników i linii wejścia/wyjścia.
+wait_for_trigger()
Oczekuje na przybycie kolejnego pojemnika do napełnienia.
+valve_close()
+valve_open()
Zamknięcie i otwarcie zaworu spustowego.
+engine_stop()
+engine_start()
Wyłączanie i włączanie odkurzacza.
+engine_state(): int
Zwraca stan pracy odkurzacza.
+balance_read_raw(): int
Odczyt aktualnej wartości wagi.
+balance_read_filtered(int): int
Uśredniony odczyt wagi - filtruje zakłucenia, ale wprowadza opóźnienia.
msg_t
Wysyłanie komunikatów do GUI.
-mq: mqd_t
Kolejka komunikatów.
+put(int)
Wysłanie komunikatu (ilość rodzajów komunikatów jest ograniczona).
state_t
Zapis i odczyt stanu procesu do/z pamięci stałej.
+load()
Odczyt stany procesu z pliku wykonywany ppodczas startu programu.
+save()
Zapis aktualnego stanu procesu do pliku podczas normalnej pracy. Konieczne jest zastosowanie asynchronicznego wejścia-wyjścia.

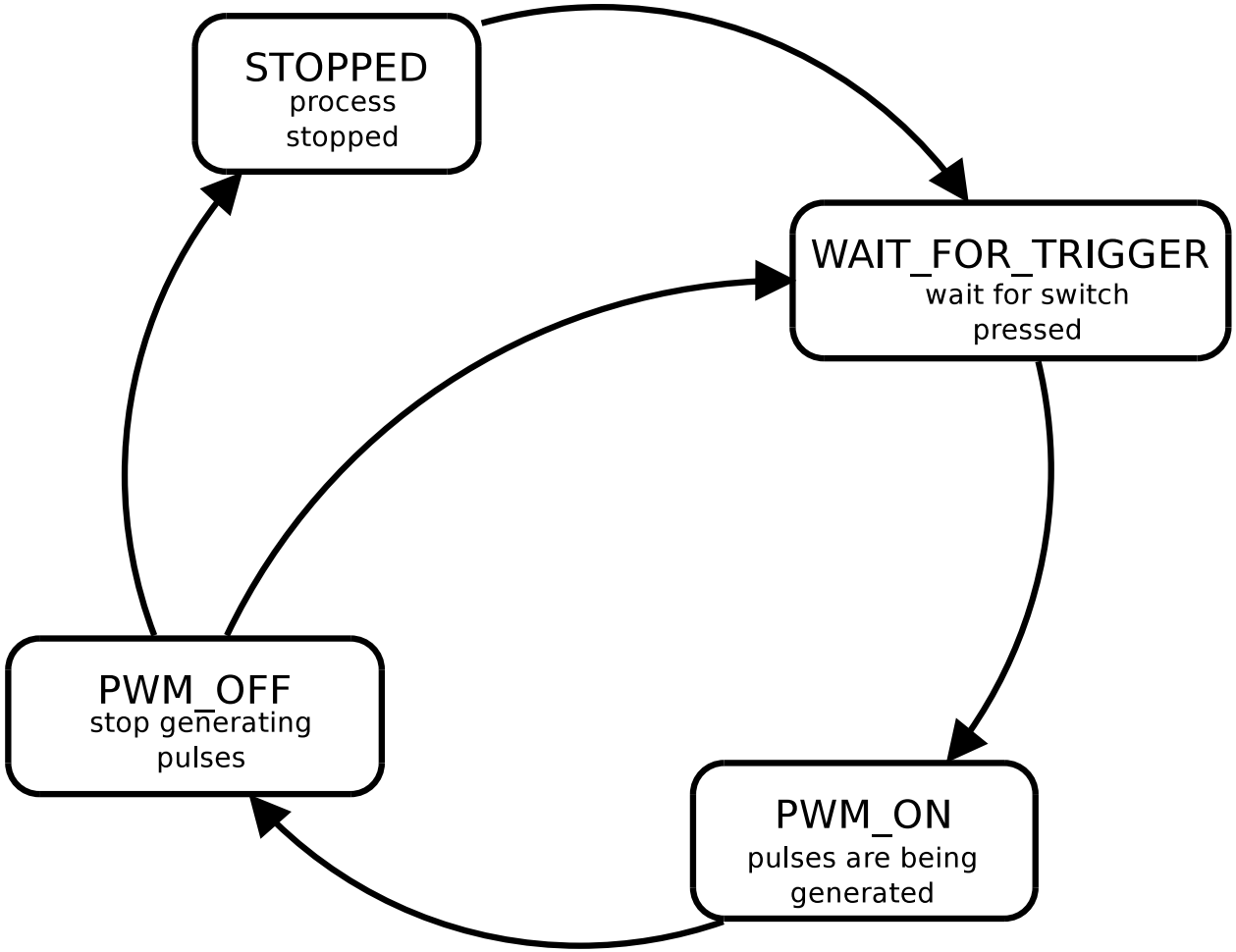
Laser - spec

- PWM controlled laser diode (actually real lasers power circuits are far more complicated)
- period: 200µs
- laser must send a given set of pulses
- laser operation is trigerred by user pressing button

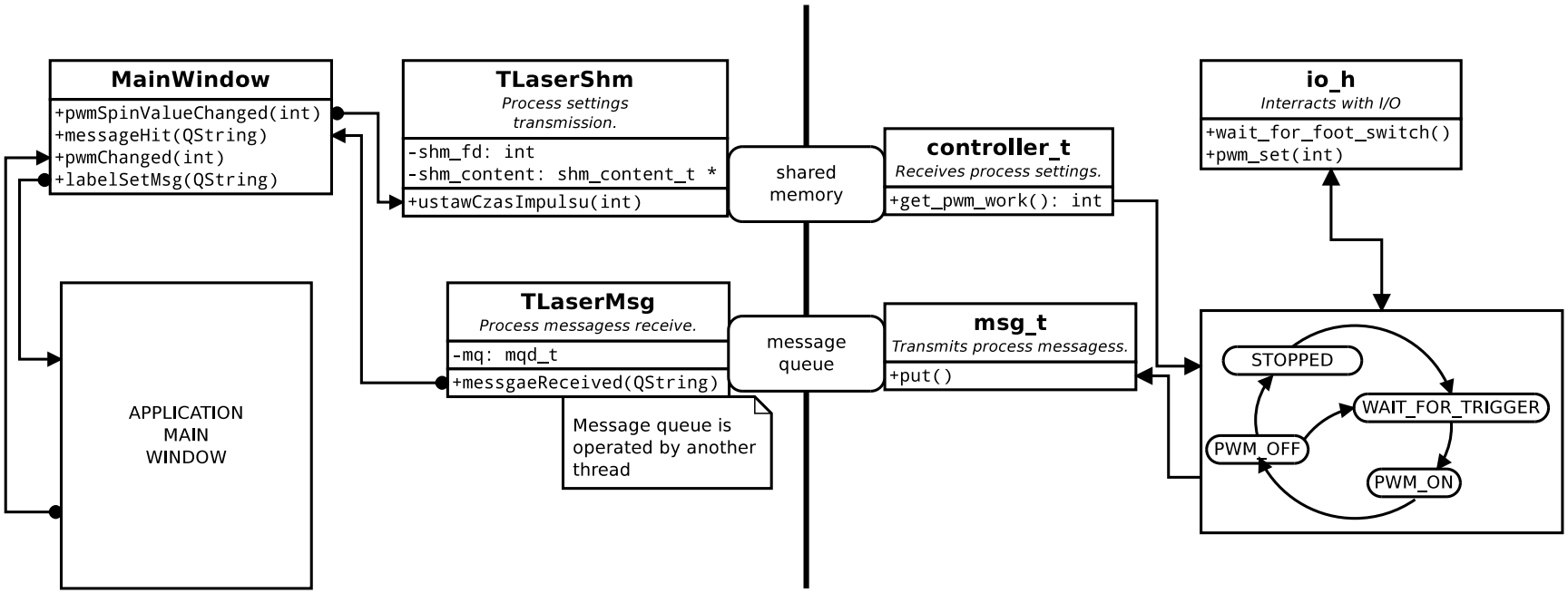
Actual model implemented with laser pointer.

- Pressing button trigers PWM for exacly 900ms
- Button both: virtual and physical

Laser


STOPPED
process stopped
WAIT_FOR_TRIGGER
wait for switch pressed
PWM_ON
pulses are being generated
PWM_OFF
stop generating pulses

Laser


MainWindow
+pwmSpinValueChanged(int)
+messageHit(QString)
+pwmChanged(int)
+labelSetMsg(QString)
APPLICATION
MAIN
WINDOW
TLaserShm
Process settings transmission.
-shm_fd: int
-shm_content: shm_content_t *
+ustawCzasImpulsu(int)
TLaserMsg
Process messageges receive.
-mq: mqd_t
+messgaeReceived(QString)
Message queue is operated by another thread
shared memory
message queue
controller_t
Receives process settings.
+get_pwm_work(): int
msg_t
Transmits process messageges.
+put()
io_h
Interracts with I/O
+wait_for_foot_switch()
+pwm_set(int)
STOPPED
WAIT_FOR_TRIGGER
PWM_OFF
PWM_ON

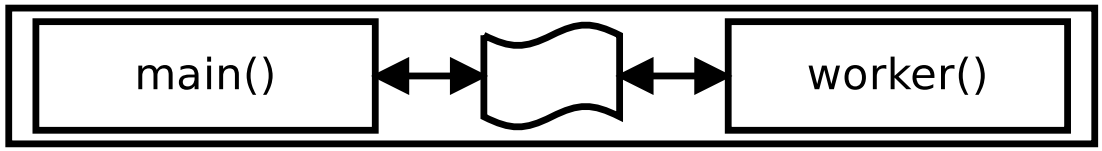
What we will need (hardware + tools)?

Plan

3 What we will need (hardware + tools)?
- Options
- MQX Lite
- Linux
- Linux + MQX

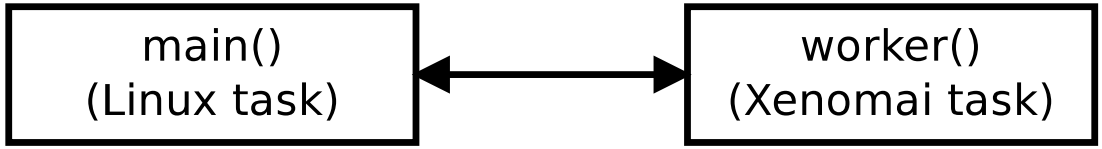
Using Linux...

Weight dosing - pneumatics are used (actuator latency is 15ms).

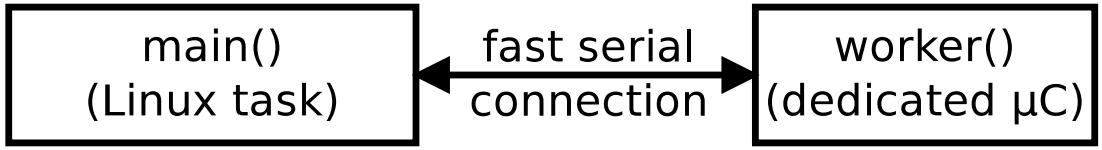


Threads - share virtual memory, have different scheduling settings.

Welding machine - µs
Medical laser controller - µs (or even less)

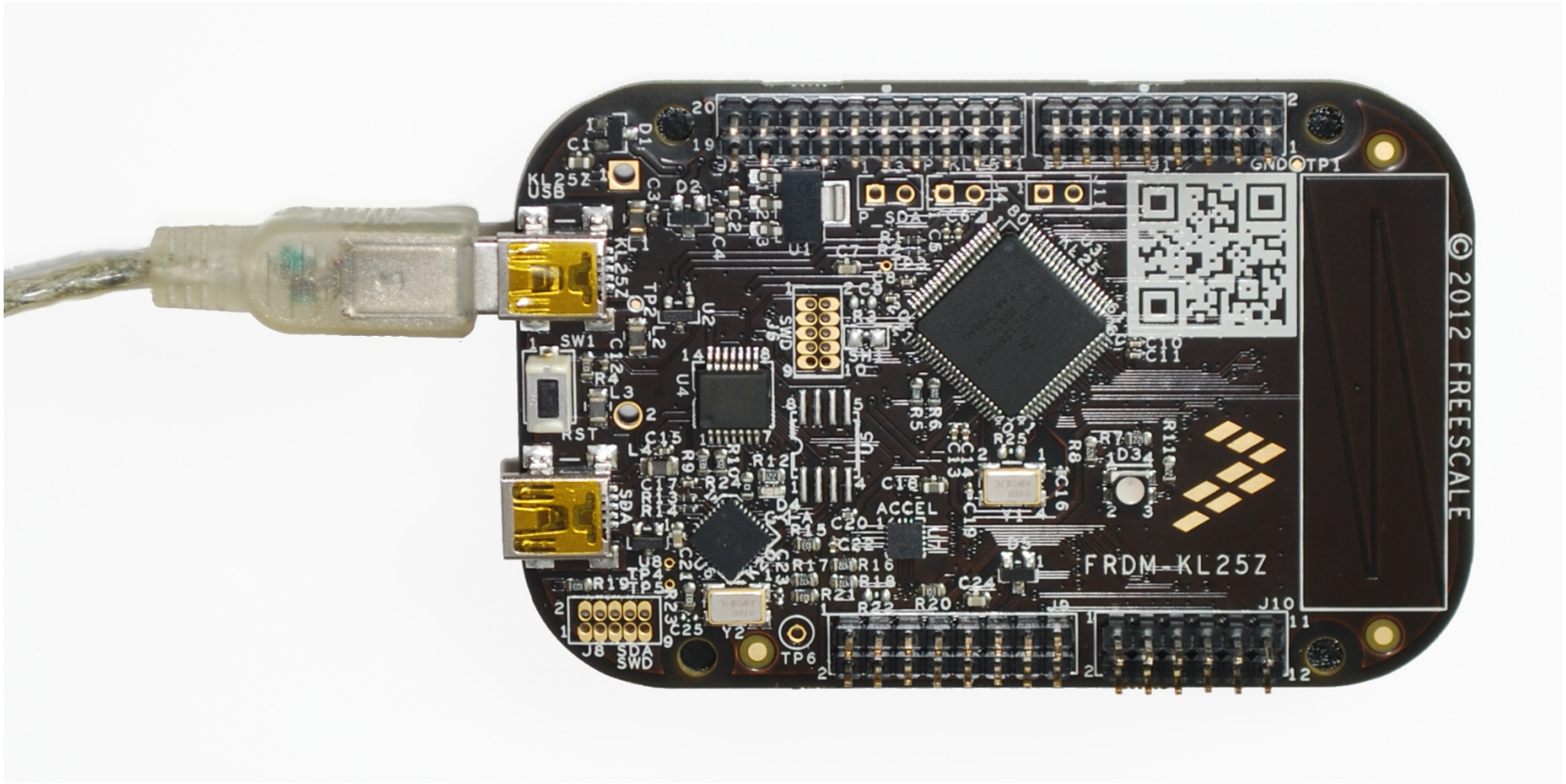


Xenomai provides better latency and predictability.

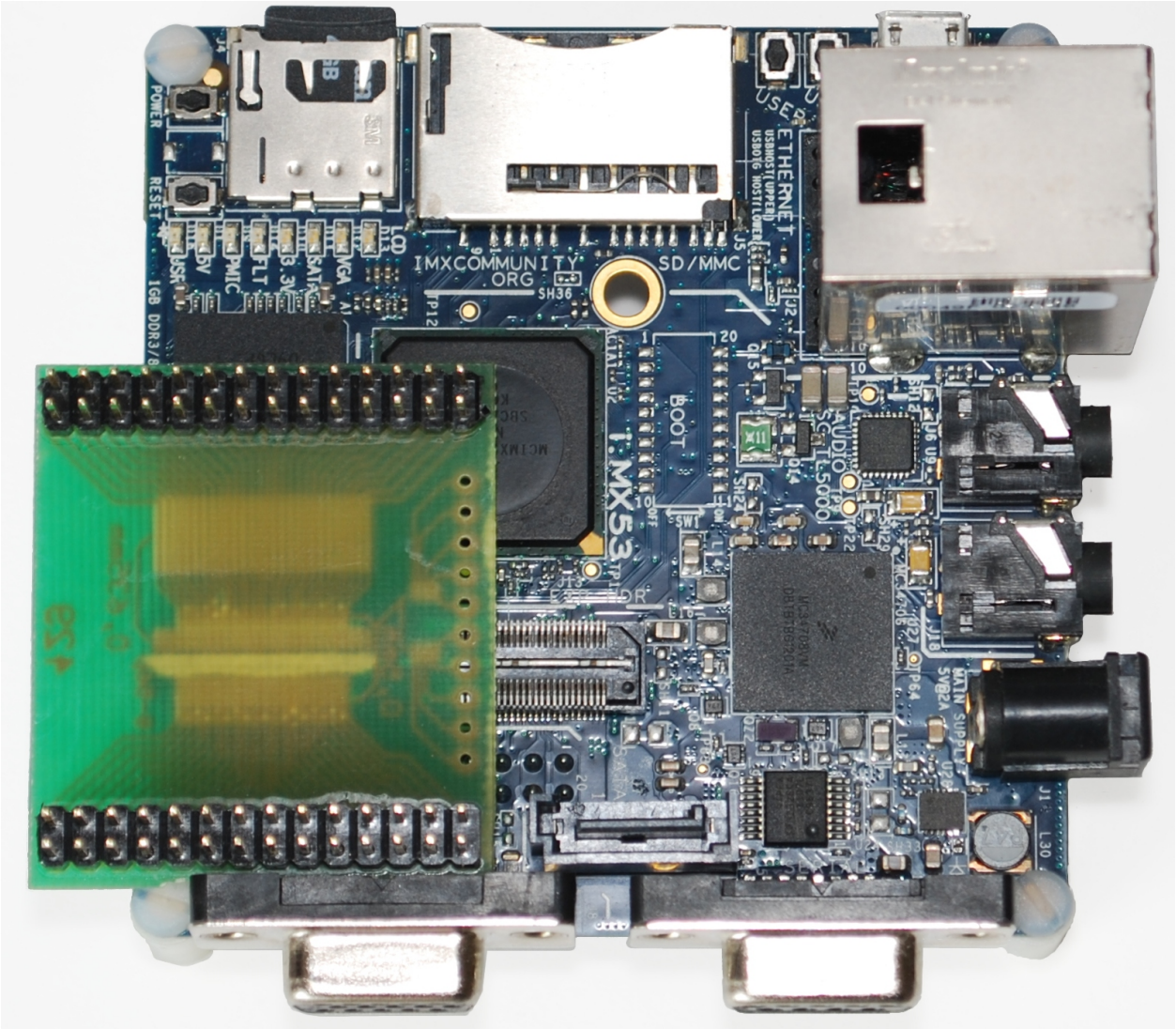


Special hardware can be utilized too:
- two processors: eg. additional Cortex-M for running worker task
- multicore systems: eg. Freescale Vybrid (Cortex-A5 + Cortex-M4)

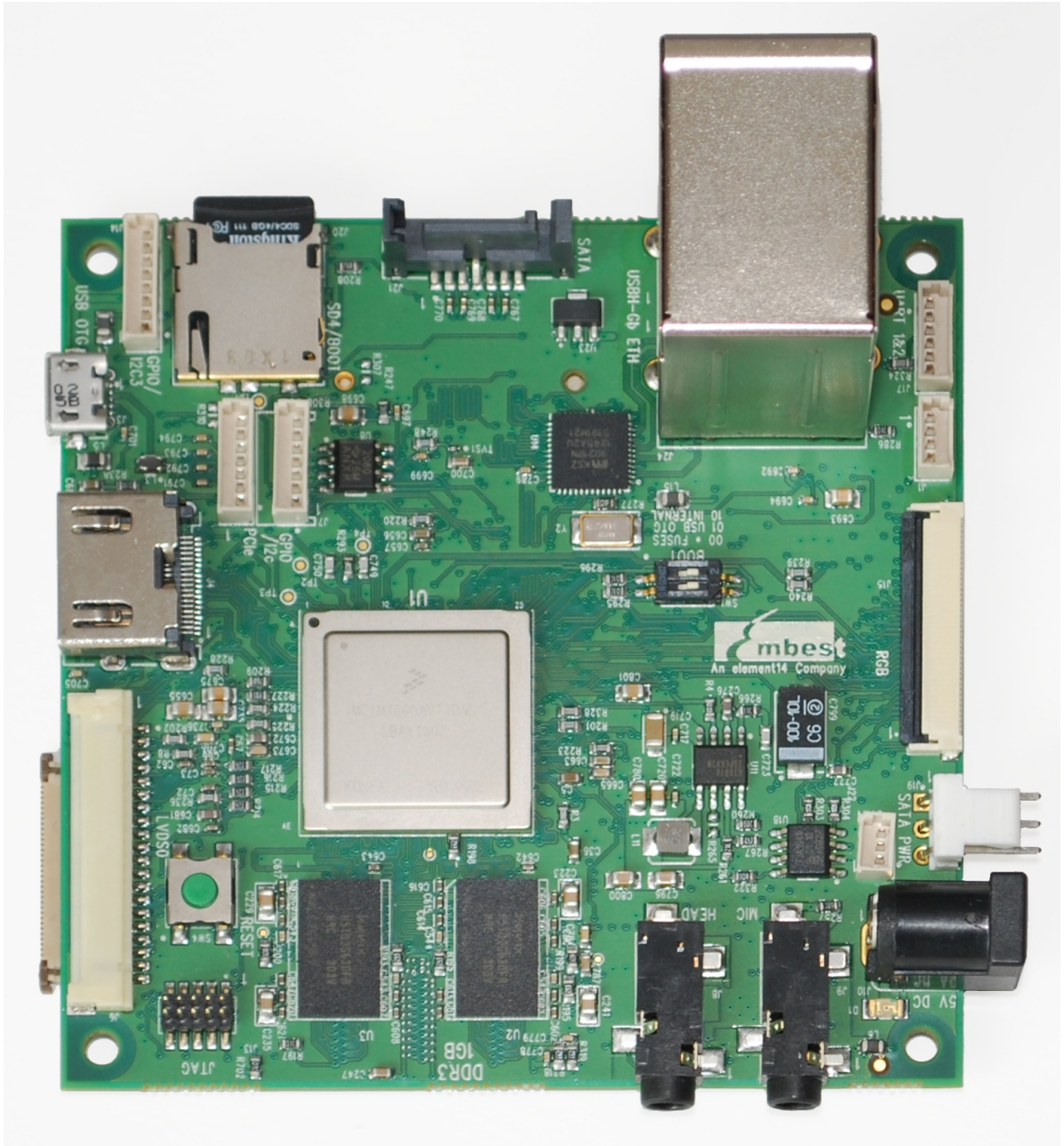
Why not?



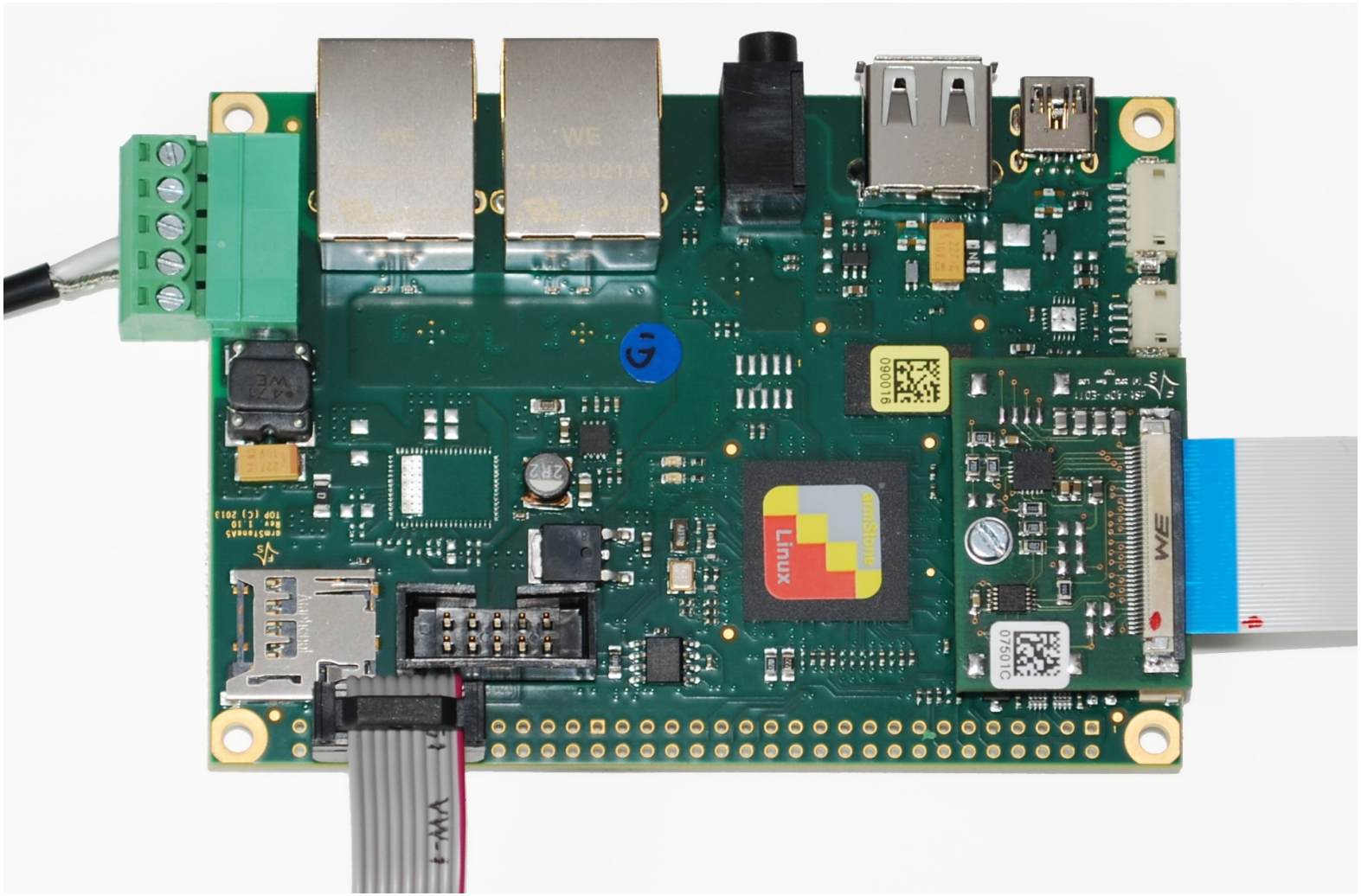
i.MX53


IMXCOMMUNITY
.ORG
SD/MMC
ETHERNET
AUDIO
SGTL5000
i.MX53
BOOT

i.MX6 Quad


SATA
USBH-Gb ETH
SD4/BOOT
USB OTG
I2C3
GPIO/
PCIe
GPIO
/I2c
mbest
An element14 Company
RGB
LVDS0
RESET
JTAG
DDR3
1GB
MIC
HEAD
5V DC
SATA PWR

Vybrid



Real-Time

Plan

BIS-LINUX

4 Real-Time
- Theory
- Latency tests
- Testing circuit
- Unmodified Linux kernel
- Real-Time - concepts
- Real-Time - measurements
- Not so good results ...
- Where is latency coming from?
- How to achieve Real-Time in Linux?

What Real-Time really means?

BIS-LINUX

Popular definition

Correctness of operation depends not only on whether performed without error, but also on the time (the upper limit) in which the operation completed.

BIS-LINUX

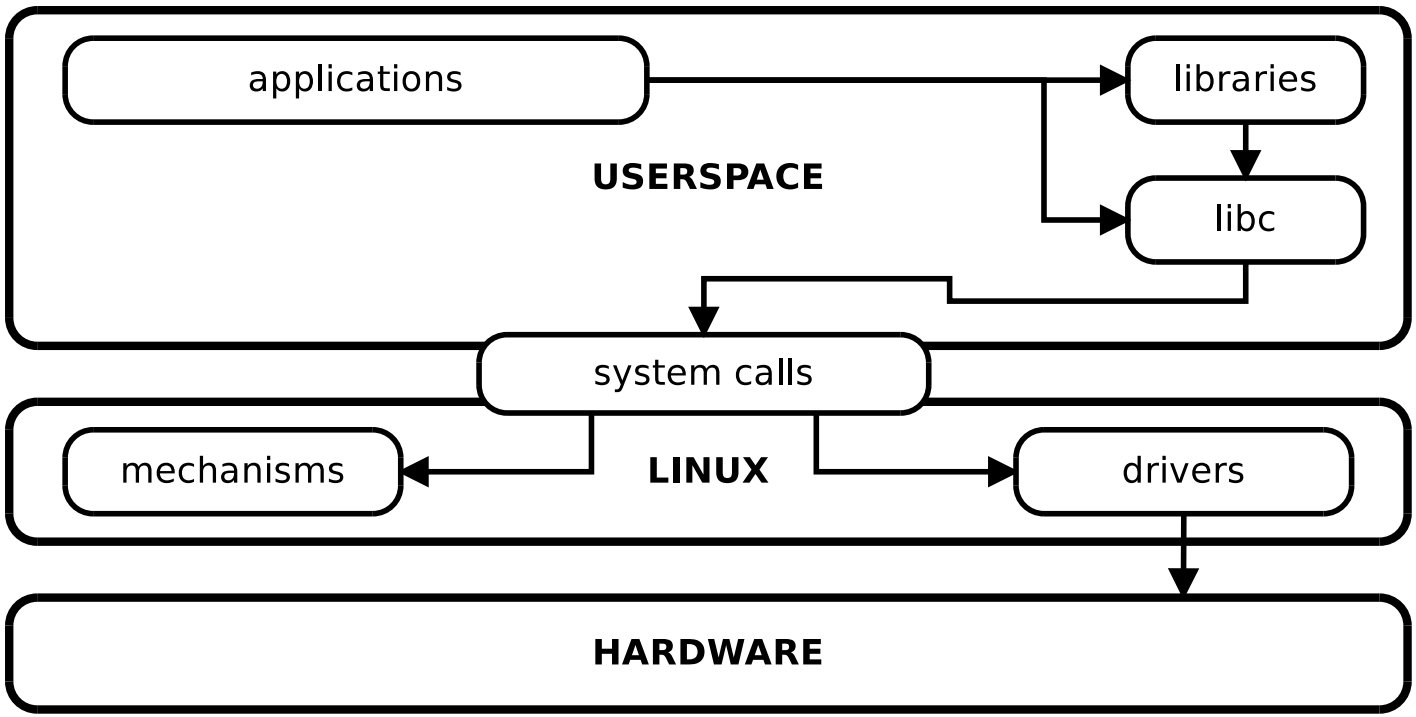
Practical definition

RT system is one in which can be proved that any required operation will be completed in a certain time.

- Mathematical proof would be perfect. Unfortunately systems are so complex, it is not possible.
- System is tested (TDD). If deadlines are met (under load) for all use-cases, system is Real-Time.
  Note: In some cases (eg. certification for safety-critical tasks), full-code coverage would be needed!

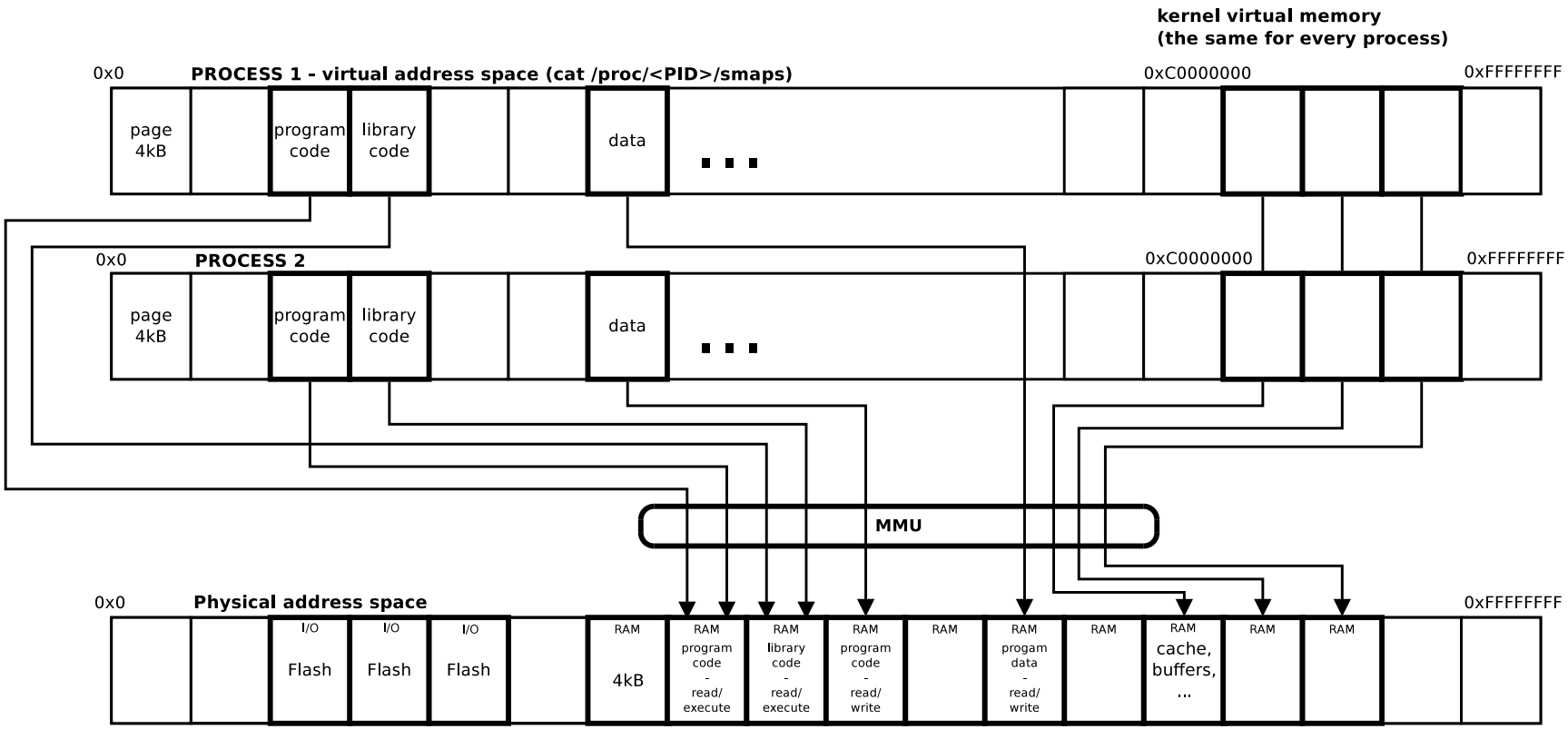
Real-Time vs. Real-Fast

- Linux kernel is designed to be „democratic”
  - resources are equally disposed
  - eg.: scheduler avoids process starvation
- Usually, determinism is not taken into account
- throughput is.

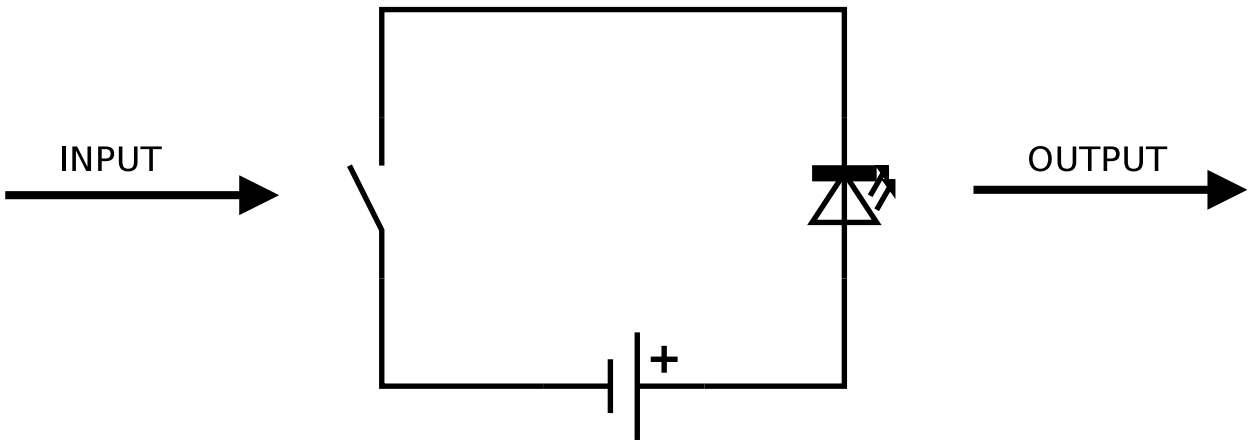


Real-Time vs. Real-Fast ...

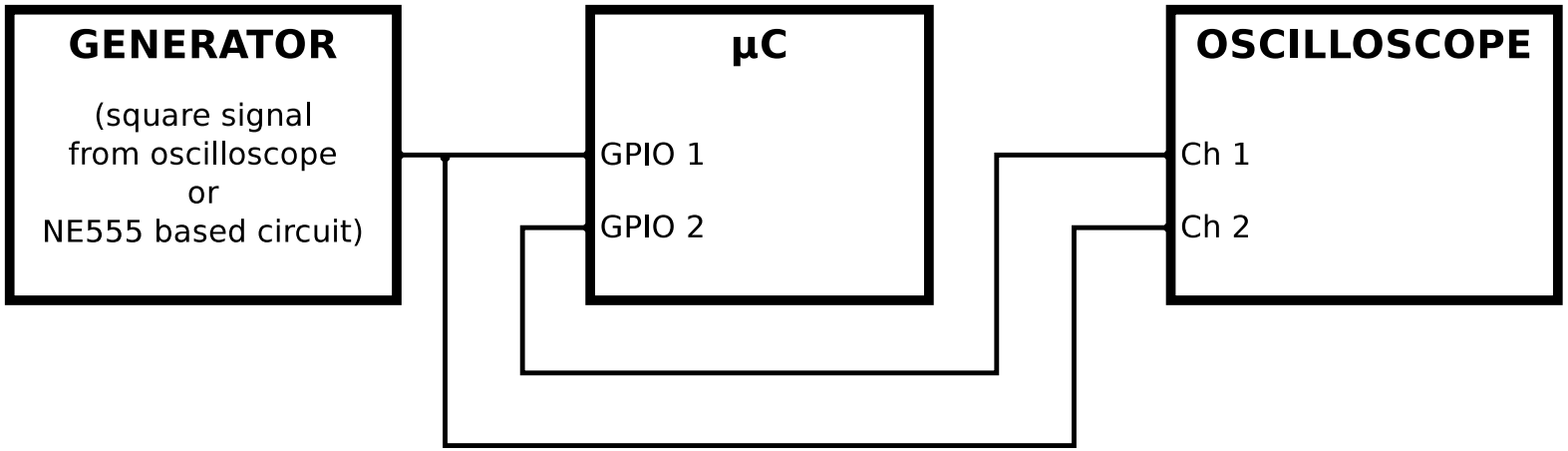
Most layers and subsystems are complex:



What are we testing?

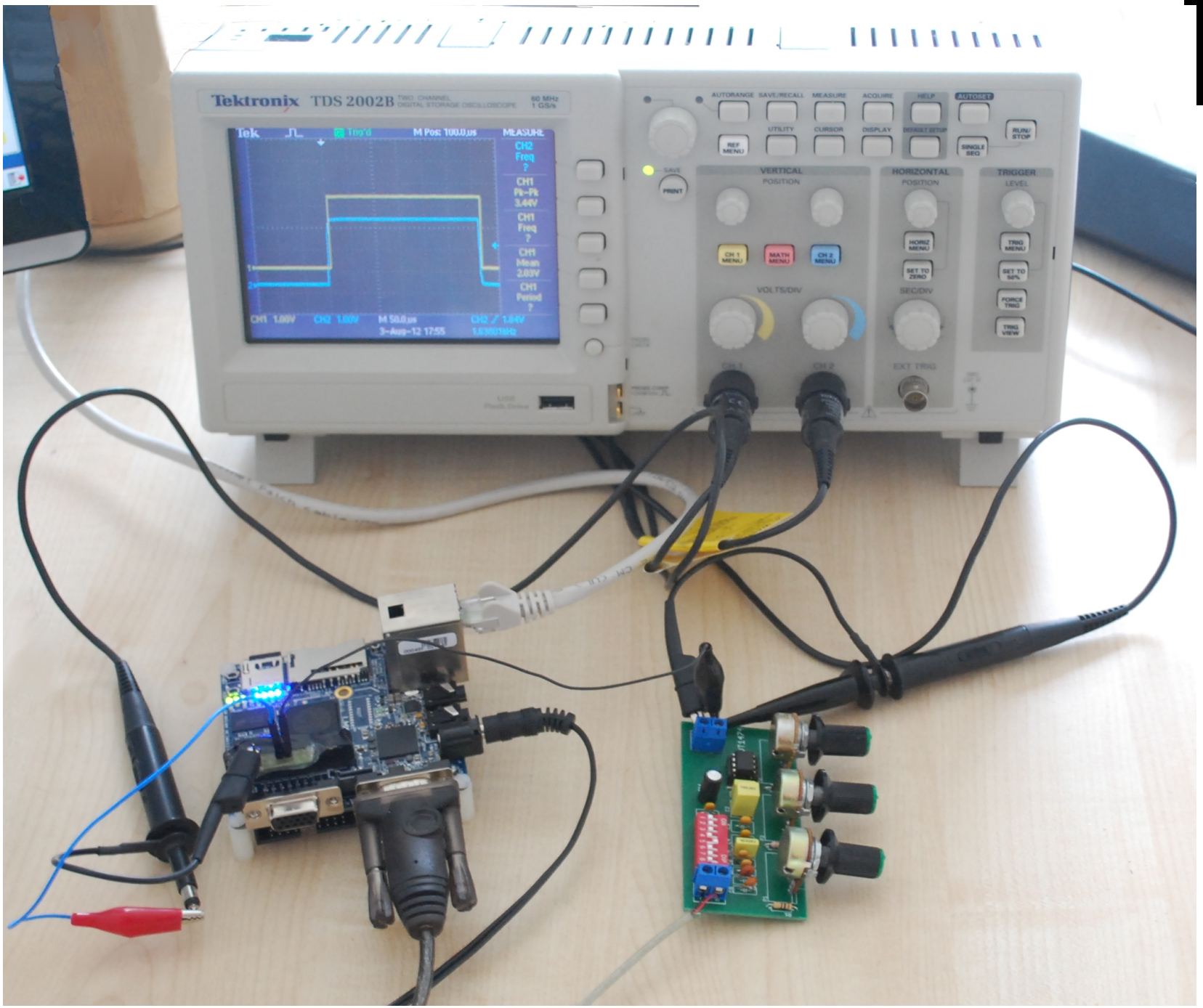


Reference circuit



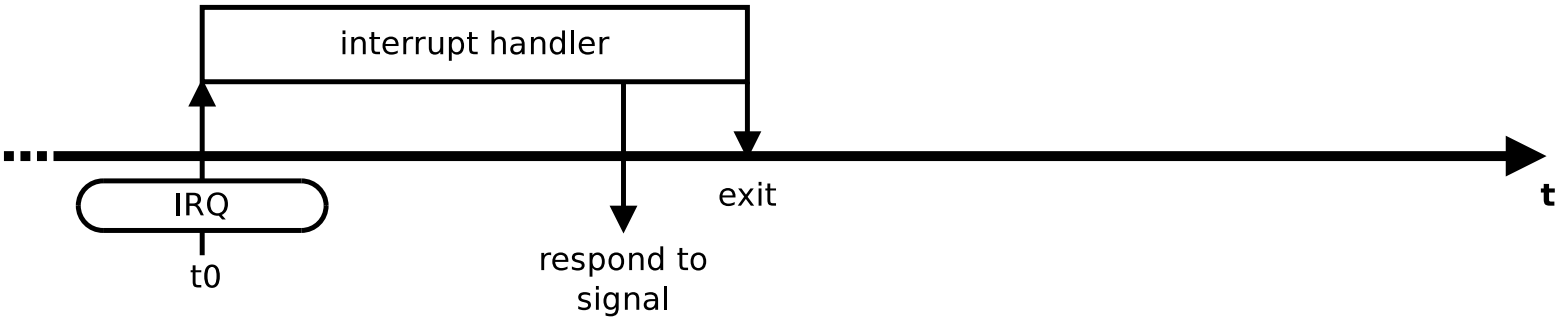
Input is triggered on falling and rising edge, output state changes according to input.

In this case, we are using GPIO pin-s.
Input can be other external or internal: timer, camera, network PHY, ADC etc.

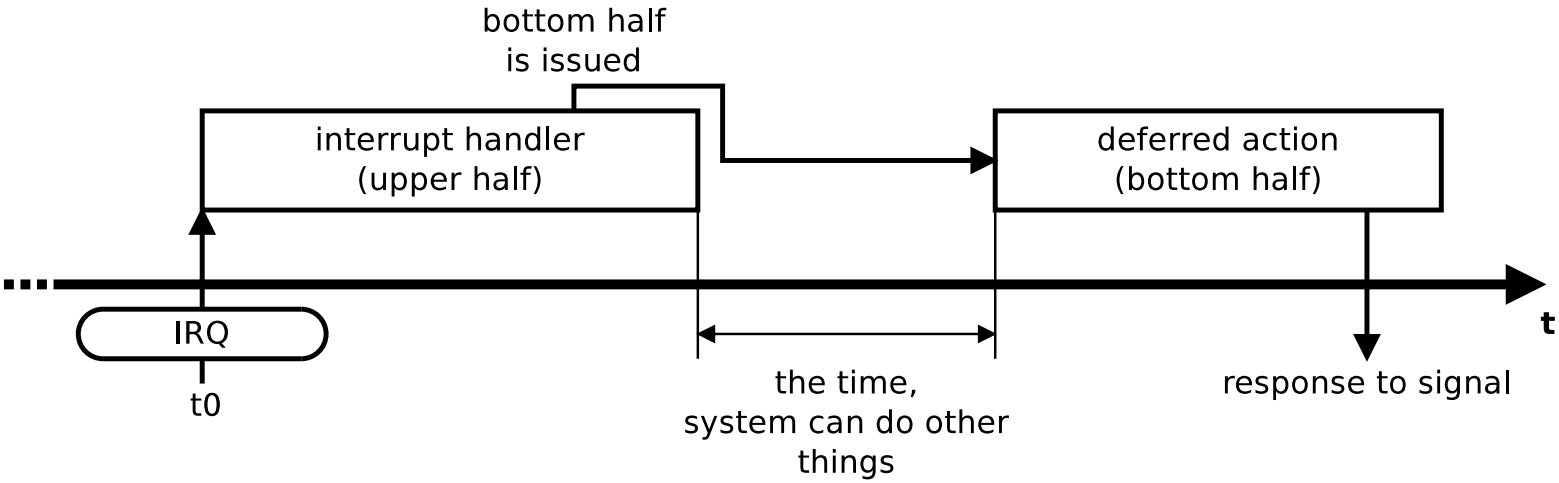

Tektronix TDS 2002B
VERTICAL
HORIZONTAL
TRIGGER

Driver design

01_inout.c

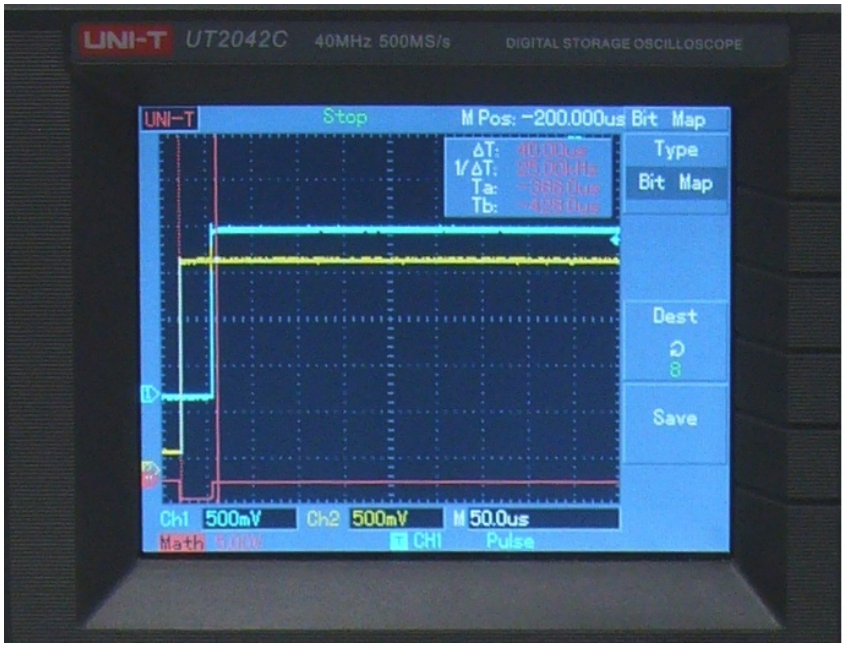


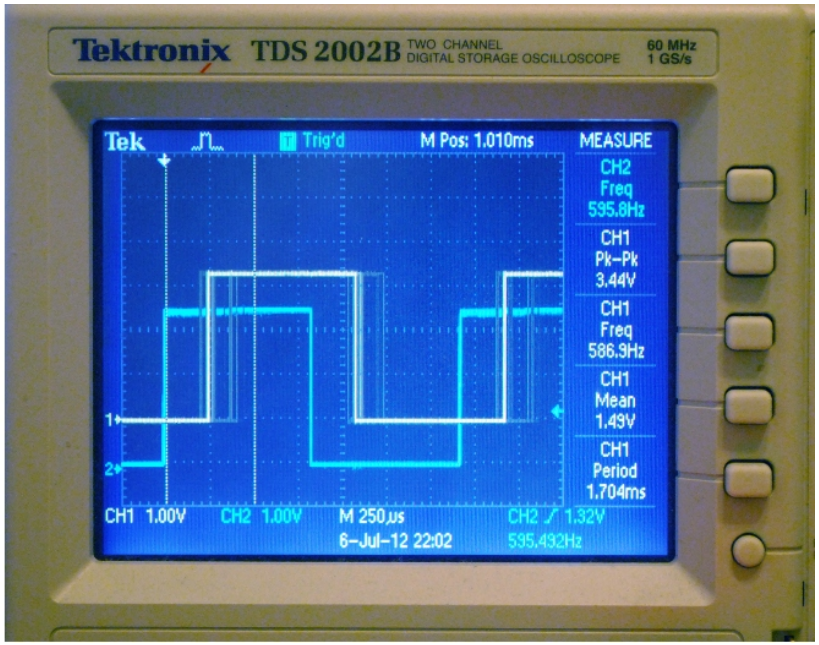
02_uinout.c



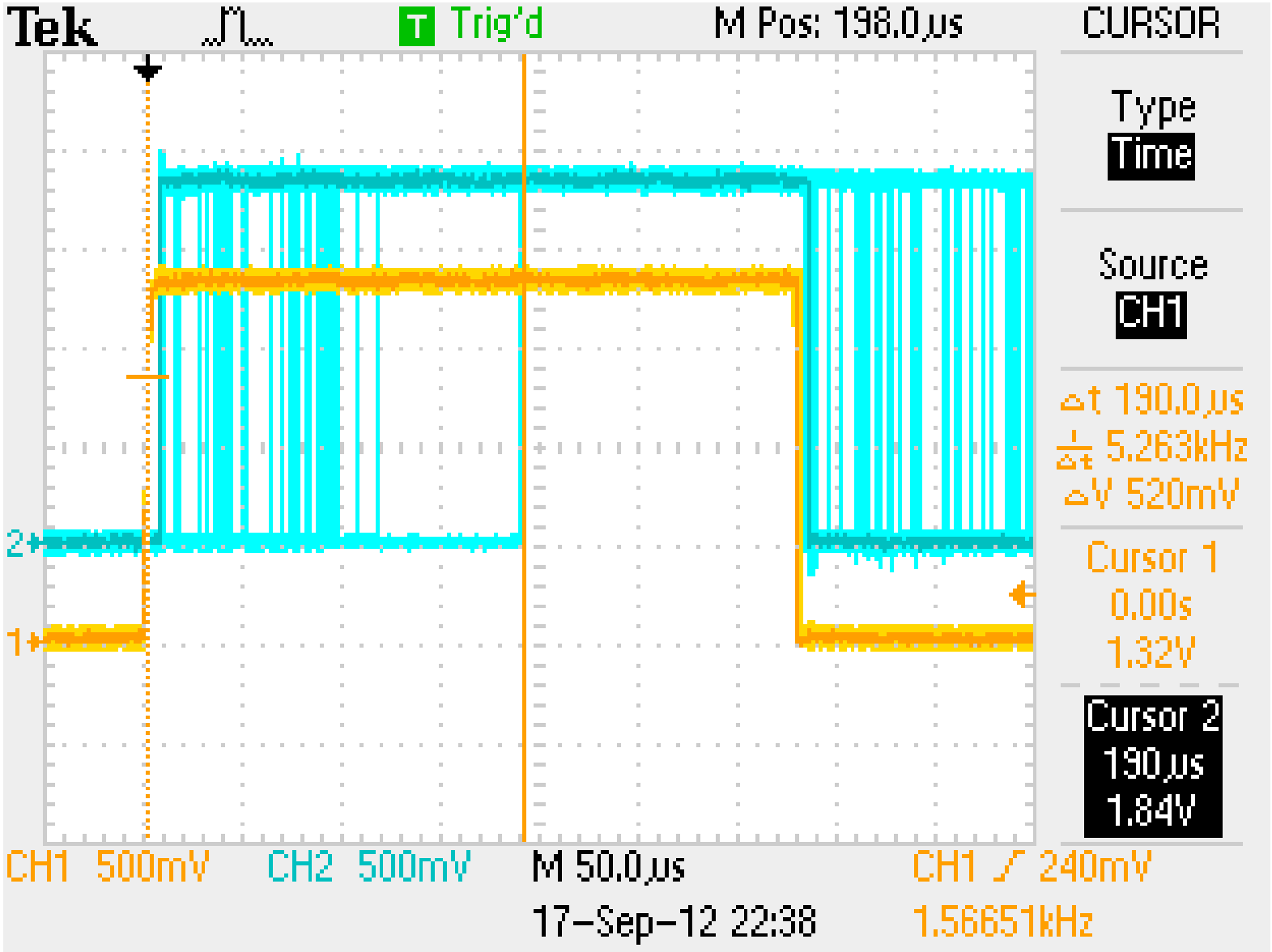
Code is on GitHub:

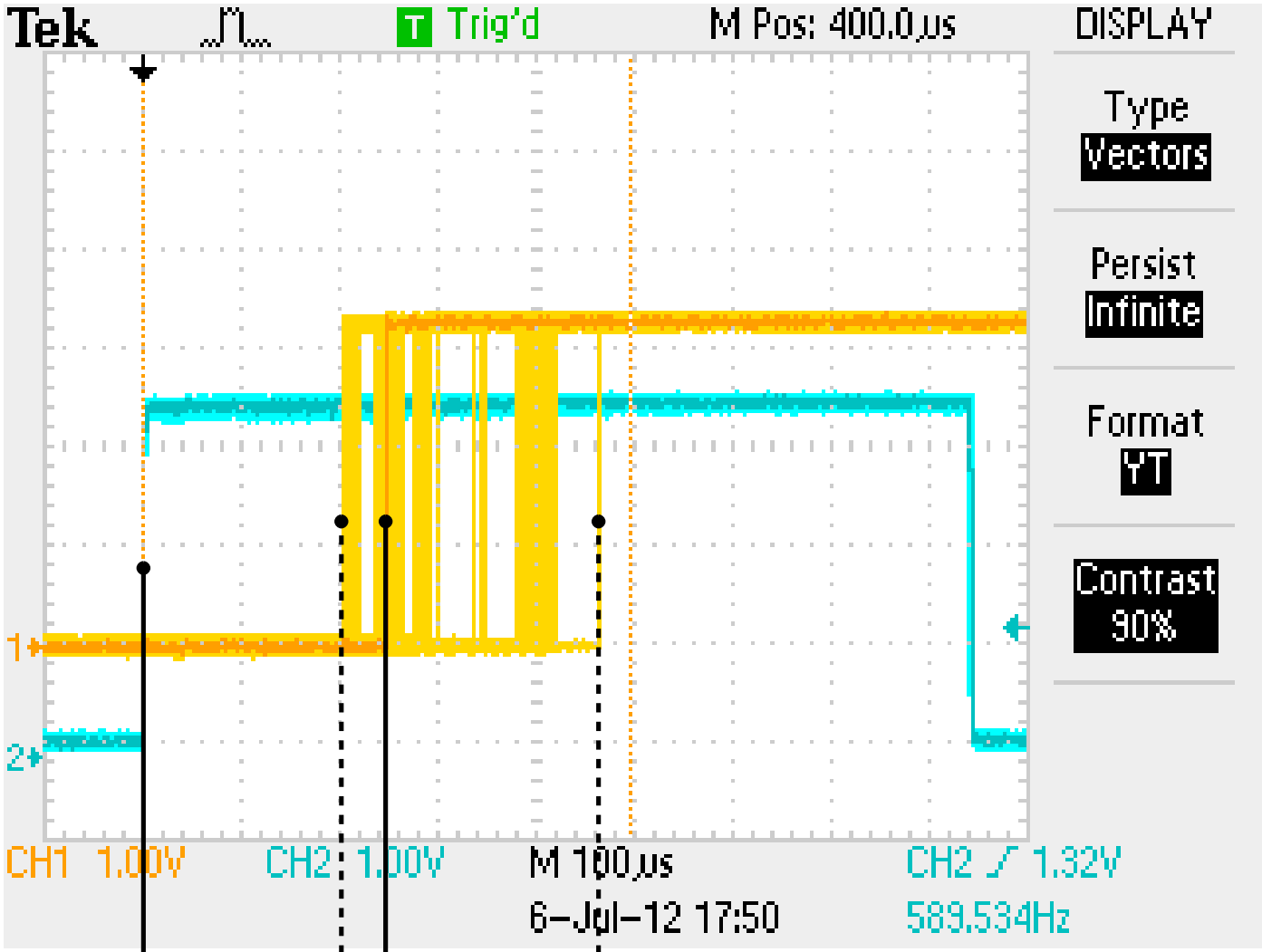
https://github.com/marcinbis/mb-rt-data.git

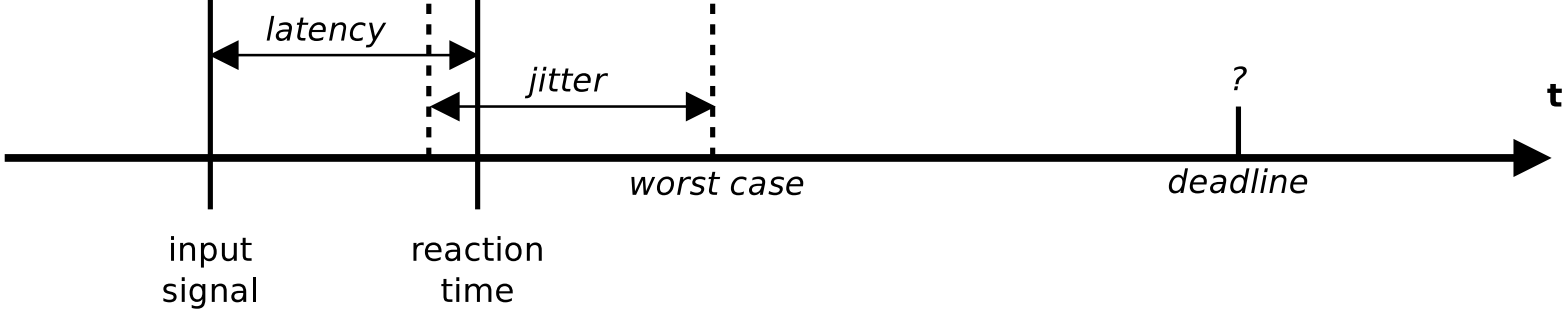

UNI-T UT2042C 40MHz 500MS/s DIGITAL STORAGE OSCILLOSCOPE
UNI-T
Stop
M Pos: -200.000us Bit Map
Type
Bit Map
Dest
Save
Ch1 500mV
Ch2 500mV
M 50.0us
Math
CH1
Pulse


Tektronix TDS 2002B TWO CHANNEL DIGITAL STORAGE OSCILLOSCOPE 60 MHz 1 GS/s
Tek
Trig'd
M Pos: 1.010ms
MEASURE
CH2 Freq 595.8Hz
CH1 Pk-Pk 3.44V
CH1 Freq 586.9Hz
CH1 Mean 1.49V
CH1 Period 1.704ms
CH1 1.00V
CH2 1.00V
M 250µs
CH2 1.32V
6-Jul-12 22:02
595.492Hz

Results


Tek
Trig'd
M Pos: 198.0µs
CURSOR
Type
Time
Source
CH1
Δt 190.0µs
1/Δt 5.263kHz
ΔV 520mV
Cursor 1
0.00s
1.32V
Cursor 2
190µs
1.84V
2
1
CH1 500mV
CH2 500mV
M 50.0µs
CH1 ∫ 240mV
17-Sep-12 22:38
1.56651kHz


Tek
Trig'd
M Pos: 400.0µs
DISPLAY
Type
Vectors
Persist
Infinite
Format
YT
Contrast
90%
1
2
CH1 1.00V
CH2 1.00V
M 100µs
CH2 1.32V
6-Jul-12 17:50
589.534Hz


latency
jitter
?
t
worst case
deadline
input signal
reaction time

BIS-LINUX

Deadline

Point in time, before which the action (system response) must occur.

- **Hard Real-Time** - deadline must be meet (fatal error if not).
- **Firm Real-Time** - deadline should be meet (system response is useless otherwise).
- **Soft Real-Time** - deadline should be meet, but nothing critical will happen if not (eg. decreased user experience, sample drop ...).

Latency

The time between the moment in which the action was to occur, and in which, in fact, occurred.

Concepts

BIS-LINUX

Jitter

Undesired deviation of latency. For various reasons, latency is not constant.
Too large jitter, renders system unusable for data acquisition.

Predictability

How much time, the action will take (eg. from IRQ occurred to handler finished executing).
O(1) algorithms should be used.

Worst Case

Due to imperfect nature of real-world systems, we are considering the Worst Case.

We have to know the latency in worst possible case.

Let's add some load

```
$ cat /proc/loadavg
5.02 3.76 2.04 2/47 432
```

- I/O on SD card:

  ```
  cat /dev/mmcblk0p1 > /dev/null
  ```

- sending ASCII data to serial console:

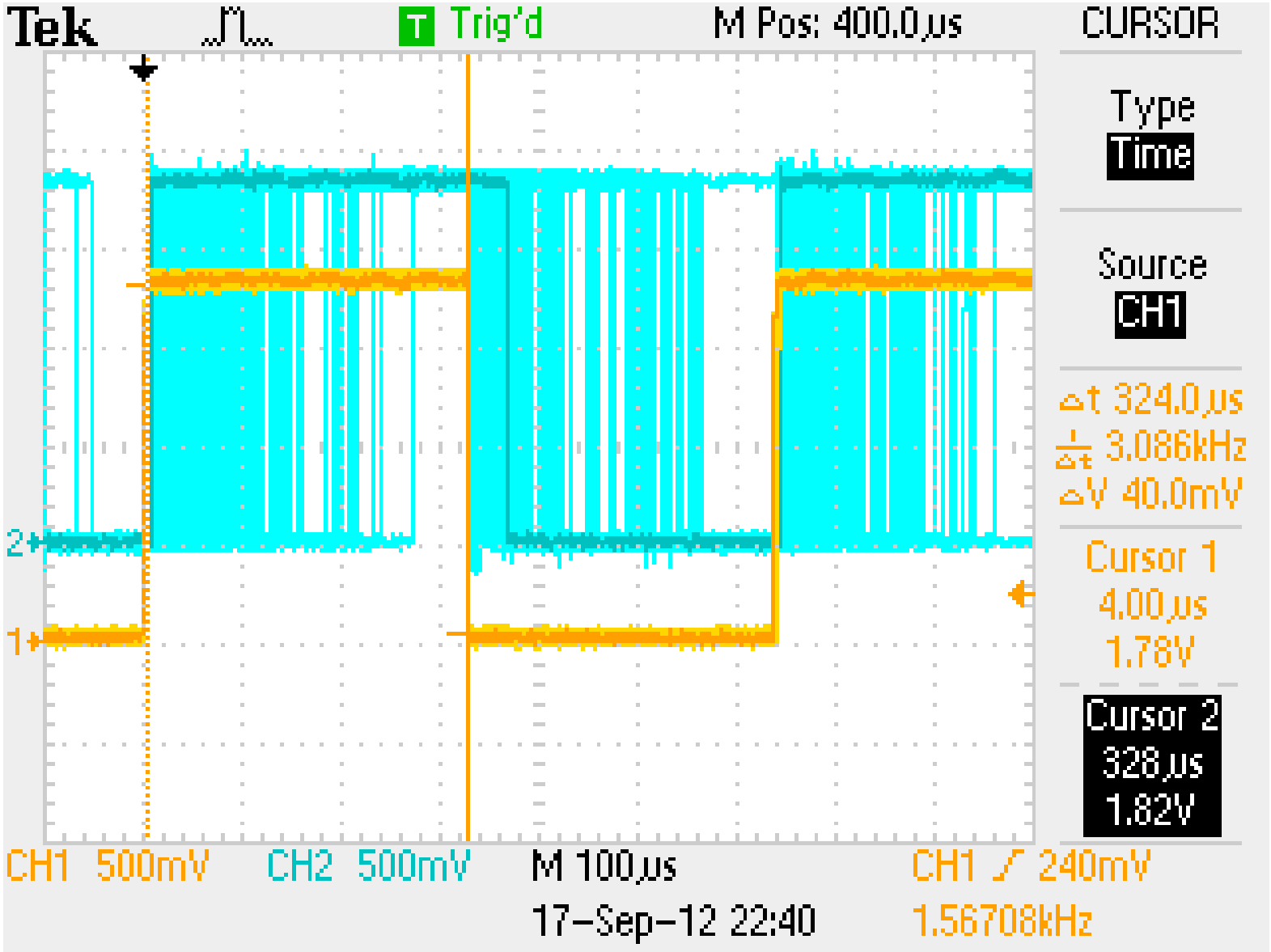
  ```
  cat /dev/zero | od -v
  ```

- send network packages:

  ```
  ping -f <ip address>
  ```

WARNING!: these tests just generate IRQ, they are not showing real-life load. Use real-case tests.

Results under load



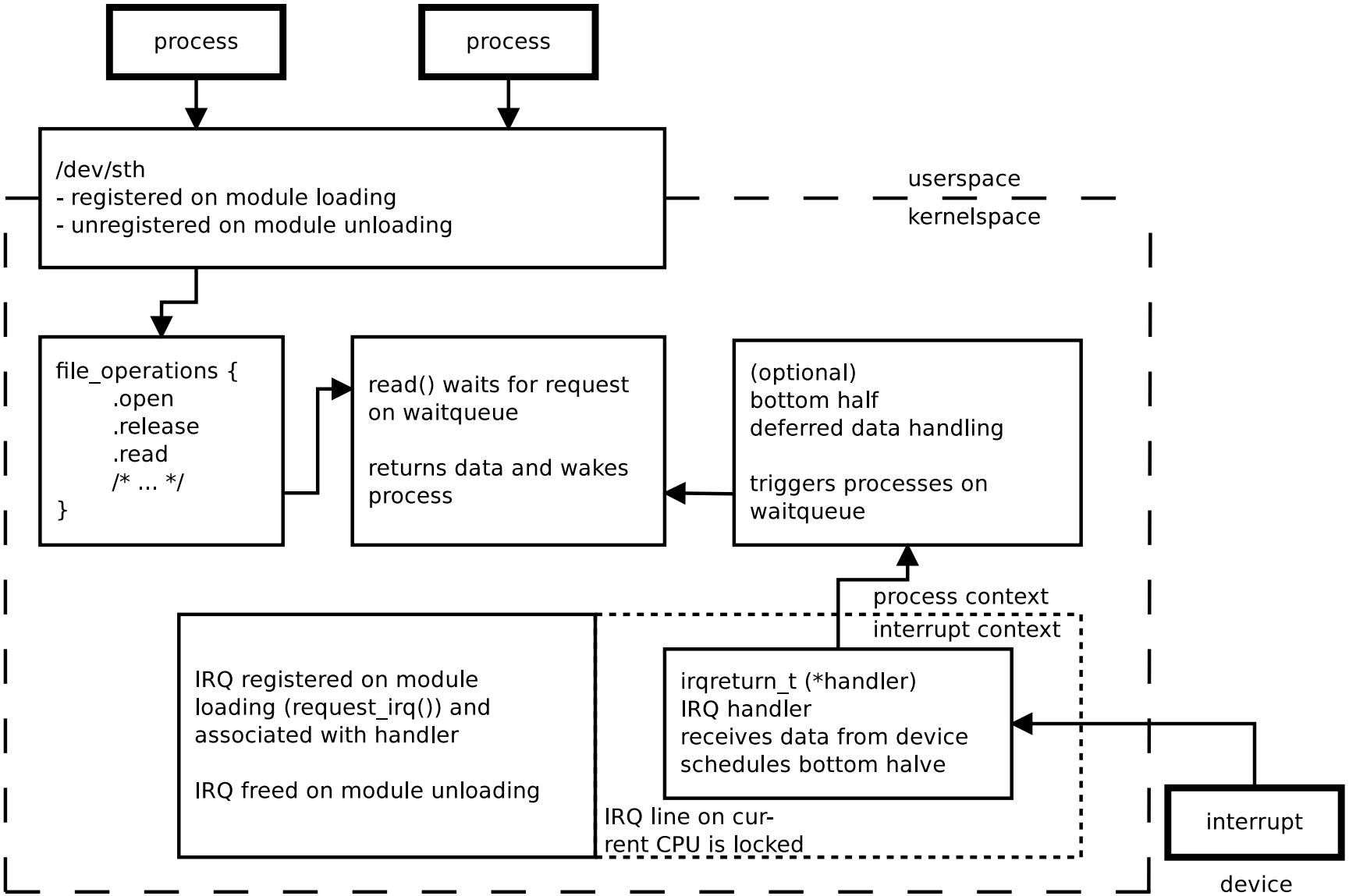
BIS-LINUX

Linux:

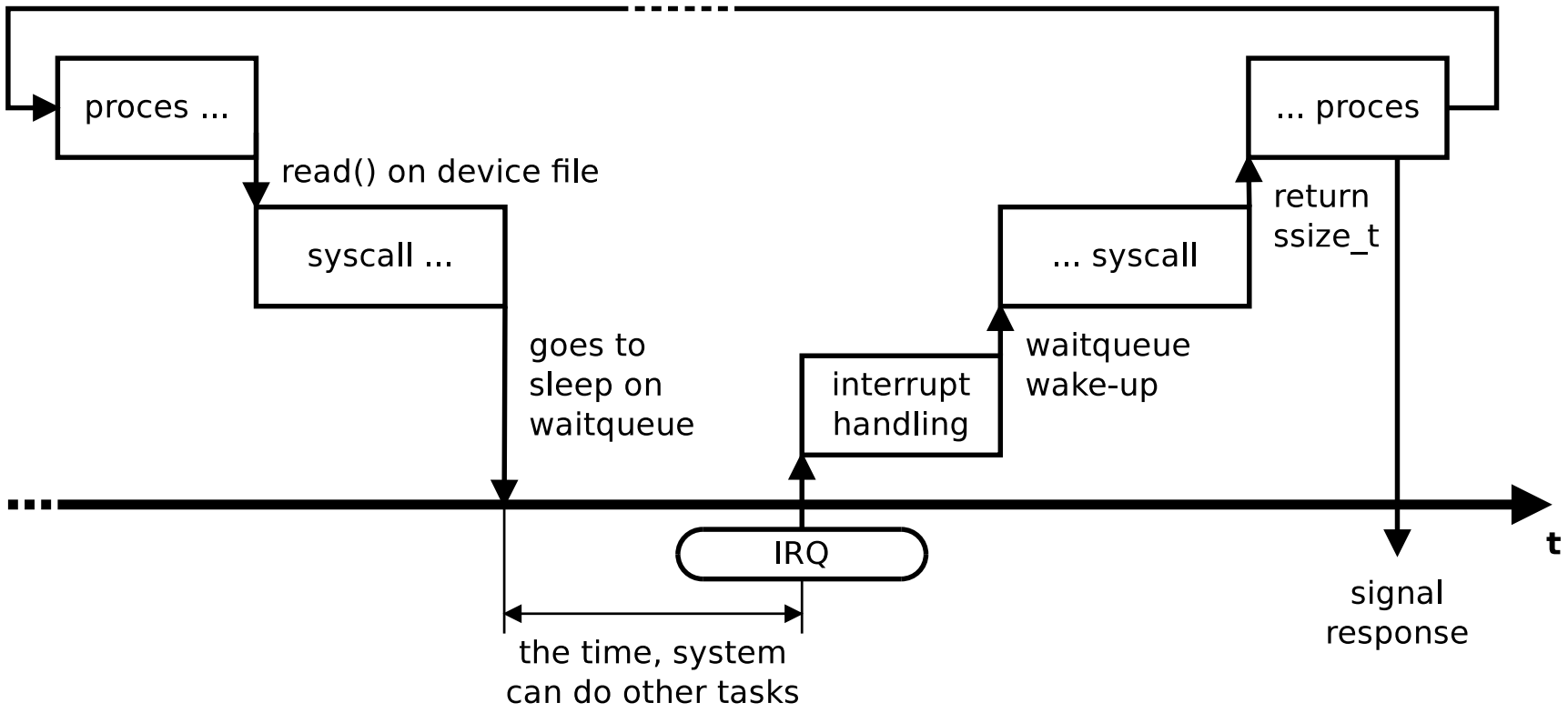
Separates:

- logic (in userspace)
- mechanisms (provided by kernel)

Interrupt-based I/O

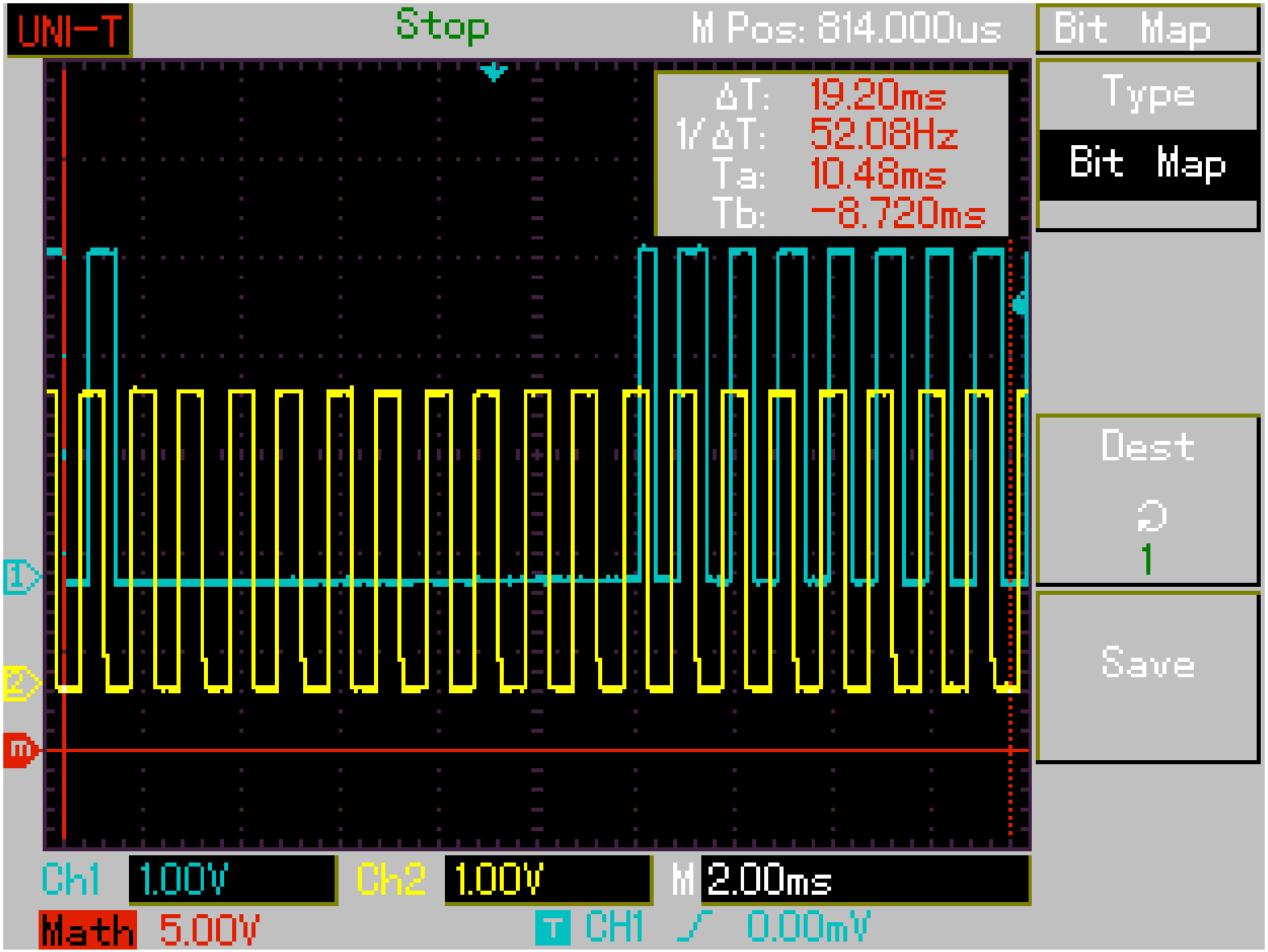


Interrupt-based I/O - another view

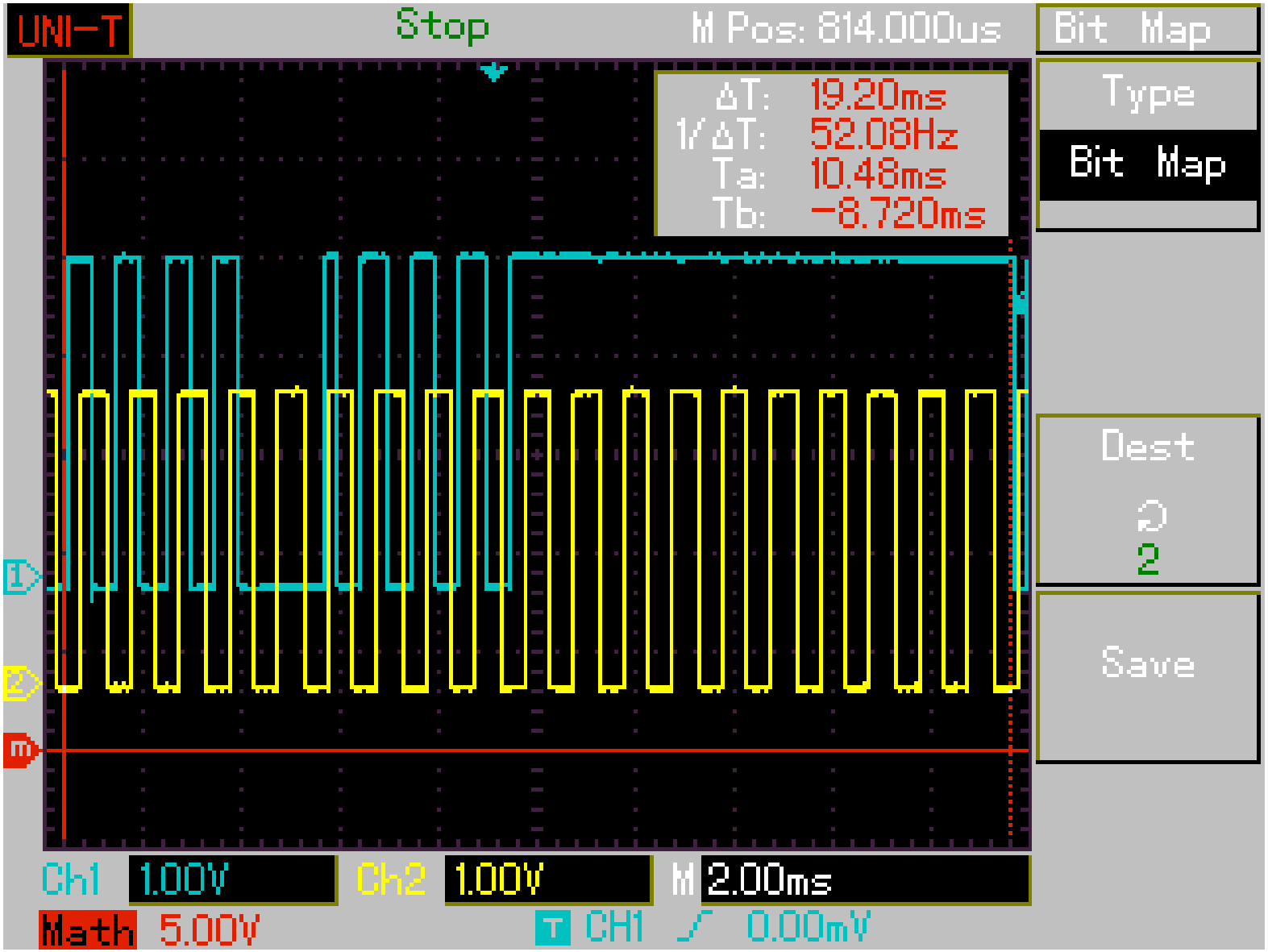


- `03_cinout.c`
- `04_real_cinout.c`
- In case of GPIO: `/sys/class/gpio/` can be used as well (`poll()`, `read()`, `write()`).

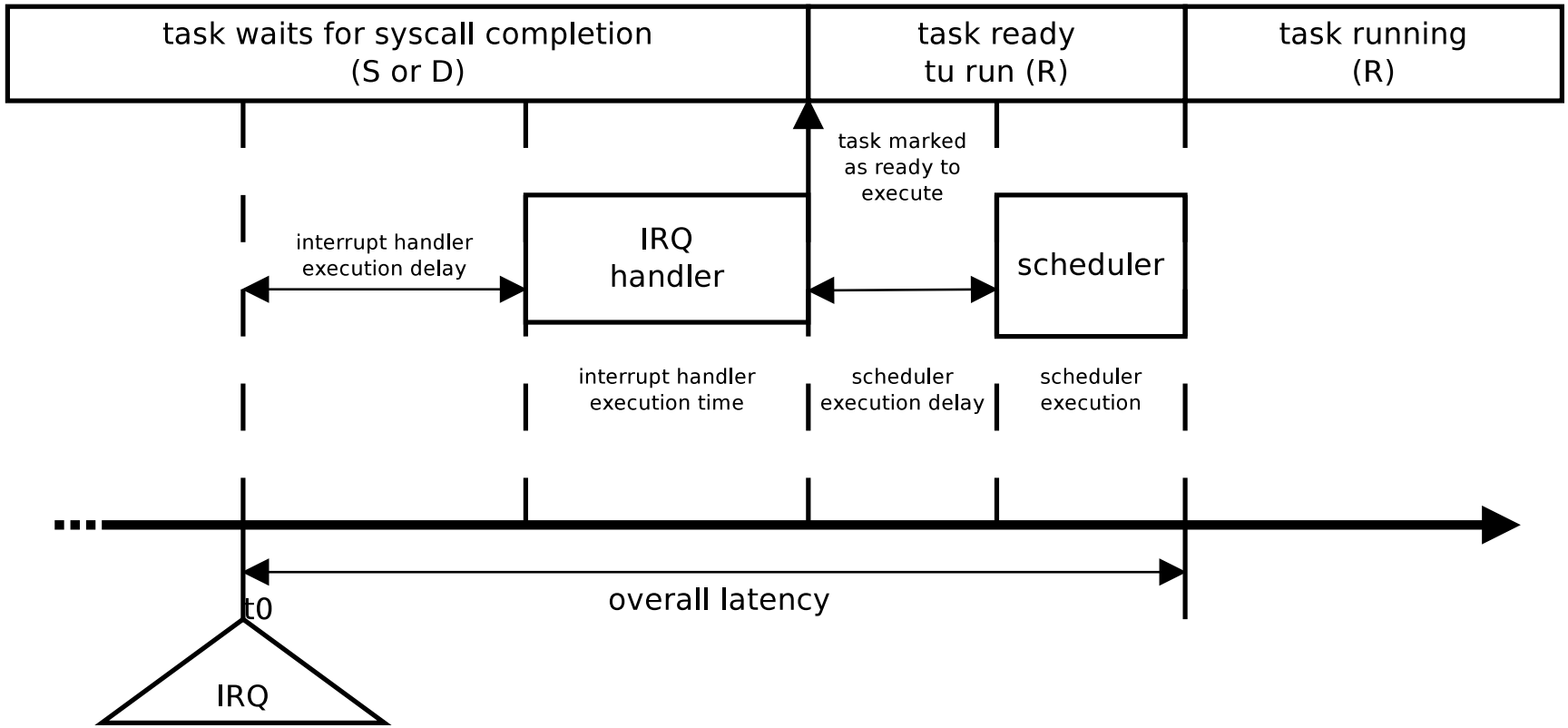
Userspace - results under load



Userspace - results under load

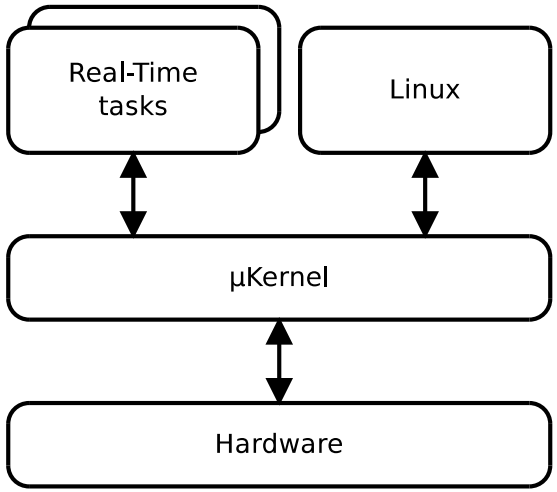


Where is latency coming from?



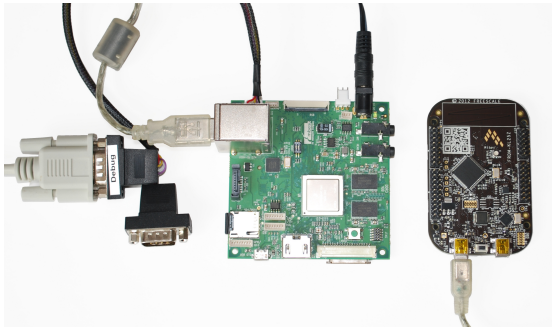
How to achieve Real-Time in Linux?

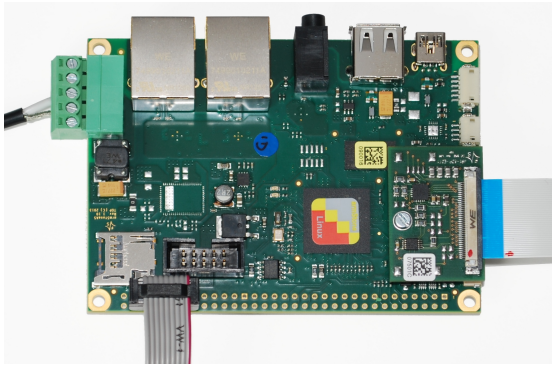
1. Micro-kernel approach:
   - RTLinux - http://en.wikipedia.org/wiki/RTLinux, there used to be open-source version: http://www.rtlinuxfree.com/.
   - Adeos/I-Pipe - http://home.gna.org/adeos/ - common base.
   - RTAI - https://www.rtai.org/ - minimum possible latency.
   - Xenomai - http://www.xenomai.org/ - provides various APIs.
2. In-kernel approach:
   - RT PREEMPT - https://rt.wiki.kernel.org http://www.kernel.org/pub/linux/kernel/projects/rt/



. . . or use hardware solutions

1. Dual processor/microcontroller approach
   - `Dual/Quad Core i.MX6` - devote one core to Real-Time task.
   - `Additional µC` - add (cheap) microcontroller and run RTOS on it (ore bare-metal app).
2. Heterogenous cores:
   - `Freescale Vybrid`



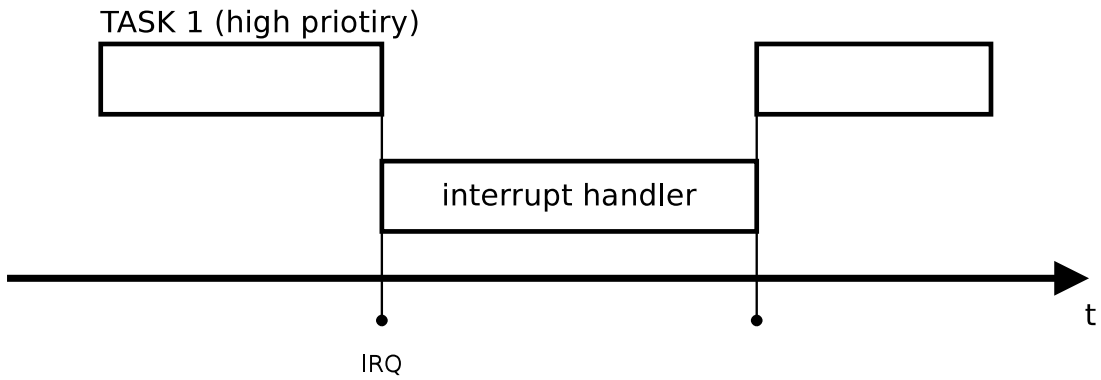


ICS using PREEMPT-RT

Plan

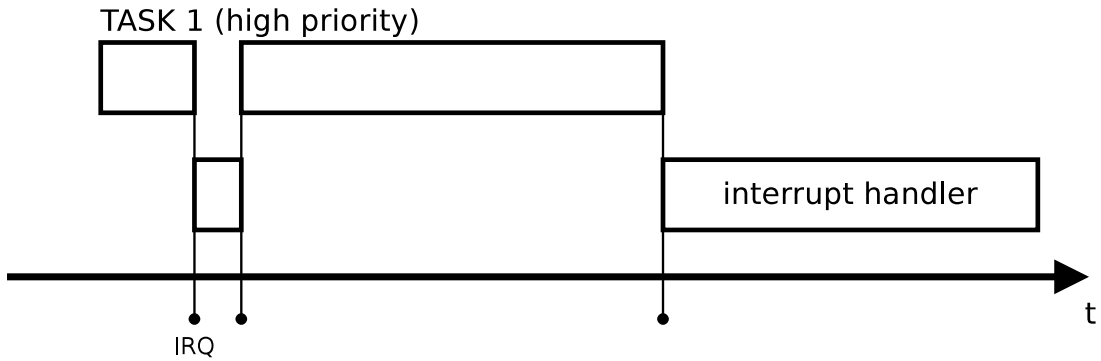
5 ICS using PREEMPT-RT
- PREEMPT-RT
- Implementation
- Internal latency measurements
- Demo

1 Standard kernel

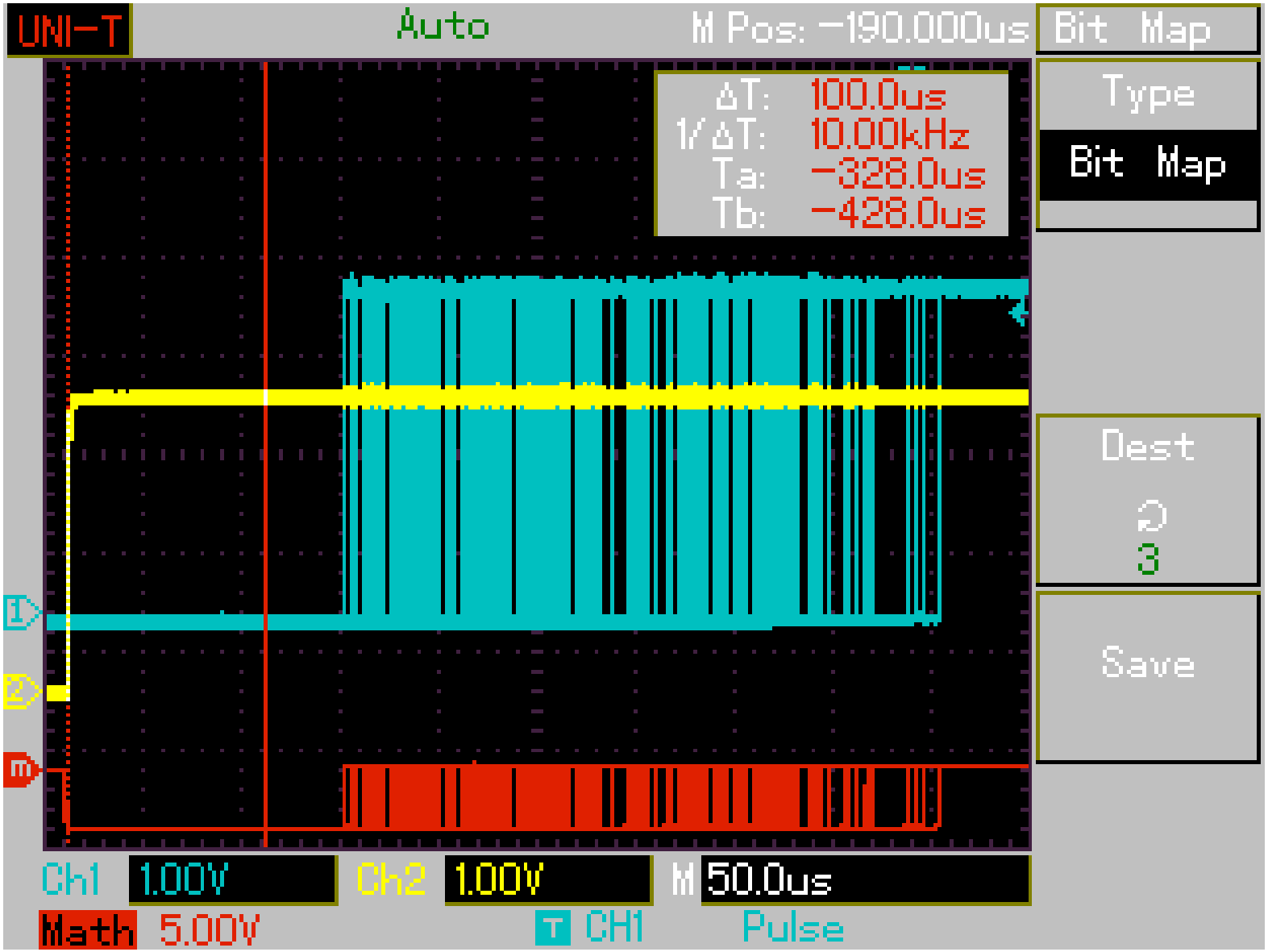


2 Interrupts as threads

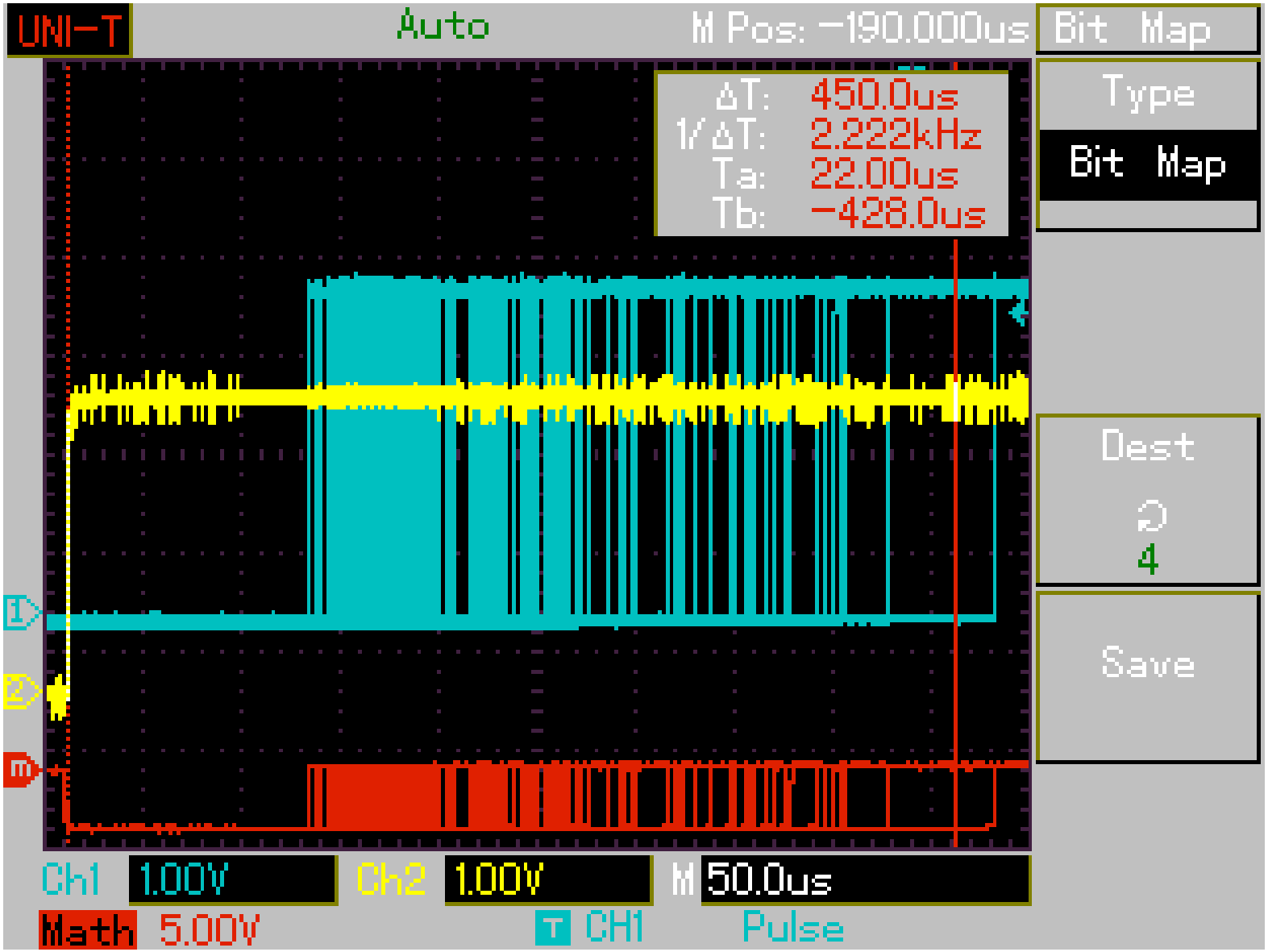
```
Kernel Features  --->
  Preemption Mode (Complete Preemption ())  --->
    (X) Complete Preemption (Real-Time)

-*- Thread Softirqs /* 2.6.33 */
-*- Thread Hardirqs
```



04 under load



04 under load



- Real-Time != Real Fast
  Maximum latency (Worst Case) is limited, but minimum latency is bigger.
- Kernel with RT-PREEMPT patch, does not make the whole system Real-Time
- Specially designed application and POSIX RT-API should be used:
  - Defined: `IEEE 1003.1b`. Linux supports it.
  - Scheduler
  - Memory locking
  - Shared memory
  - RT signals
  - Semaphores (priority inheritance)
  - Timers (esp. `CLOCK_MONOTONIC`)
  - AIO

Use appropriate programming language

- C - but make it object-oriented (for reference - Linux kernel: buses, drivers, classes etc.)
- C++ - would be nice too
  - cannot be utilized inside kernel or as Xenomai kernel process
  - can be executed as bare-metal µC or in userspace
- Utilize design patterns.

BIS-LINUX

Set the proper scheduler class and priority

```
struct sched_param sp;
sp.sched_priority = MY_PRIORITY;
ret = sched_setscheduler(0, SCHED_FIFO, &sp);
```

- Interrupts run in threads, default to: `SHCED_FIFO`/50.
- . . . do not forget to fine-tune them.
- `SCHED_DEADLINE` can be helpful too.

Lock all memory (mlock)

```
mlockall(MCL_CURRENT|MCL_FUTURE);
```

Try to cause page_fault (allocated memory, data from files)

```
buf = malloc(BUF_SIZE);
memset(buf, 0, BUF_SIZE);
```

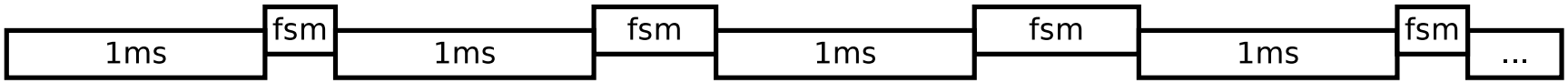
- Memory is locked, so it stays on place.

Prefault the stack (it can be shared within process we have forked from)

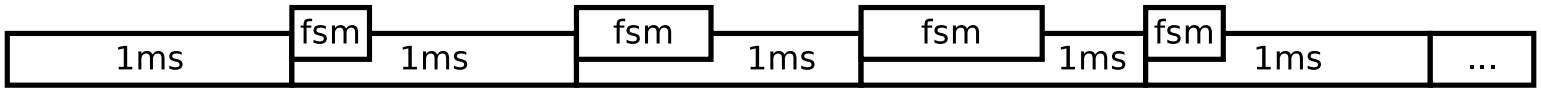
```
/* GCC will not inline this function */
__attribute__ ((noinline)) void stack_prefault(void)
{
    unsigned char tab[MAX_SAFE_STACK];
    /* GCC will omit optimizations */
    asm("");
    memset(tab, 0, MAX_SAFE_STACK);
}
/*...*/
stack_prefault();
```

Use POSIX timer to do the fsm step (in a proper way)

```
#define NSEC_IN_SEC 1000000000l
#define INTERVAL 10000000l
struct timespec timeout;

clock_gettime(CLOCK_MONOTONIC, &timeout);
while (1) {
    do_fsm_step(&some_data);
    timeout.tv_nsec += INTERVAL;
    if (timeout.tv_nsec >= NSEC_IN_SEC) {
        timeout.tv_nsec -= NSEC_IN_SEC;
        timeout.tv_sec++;
    }
    clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME,
                    &timeout, NULL);
}
```



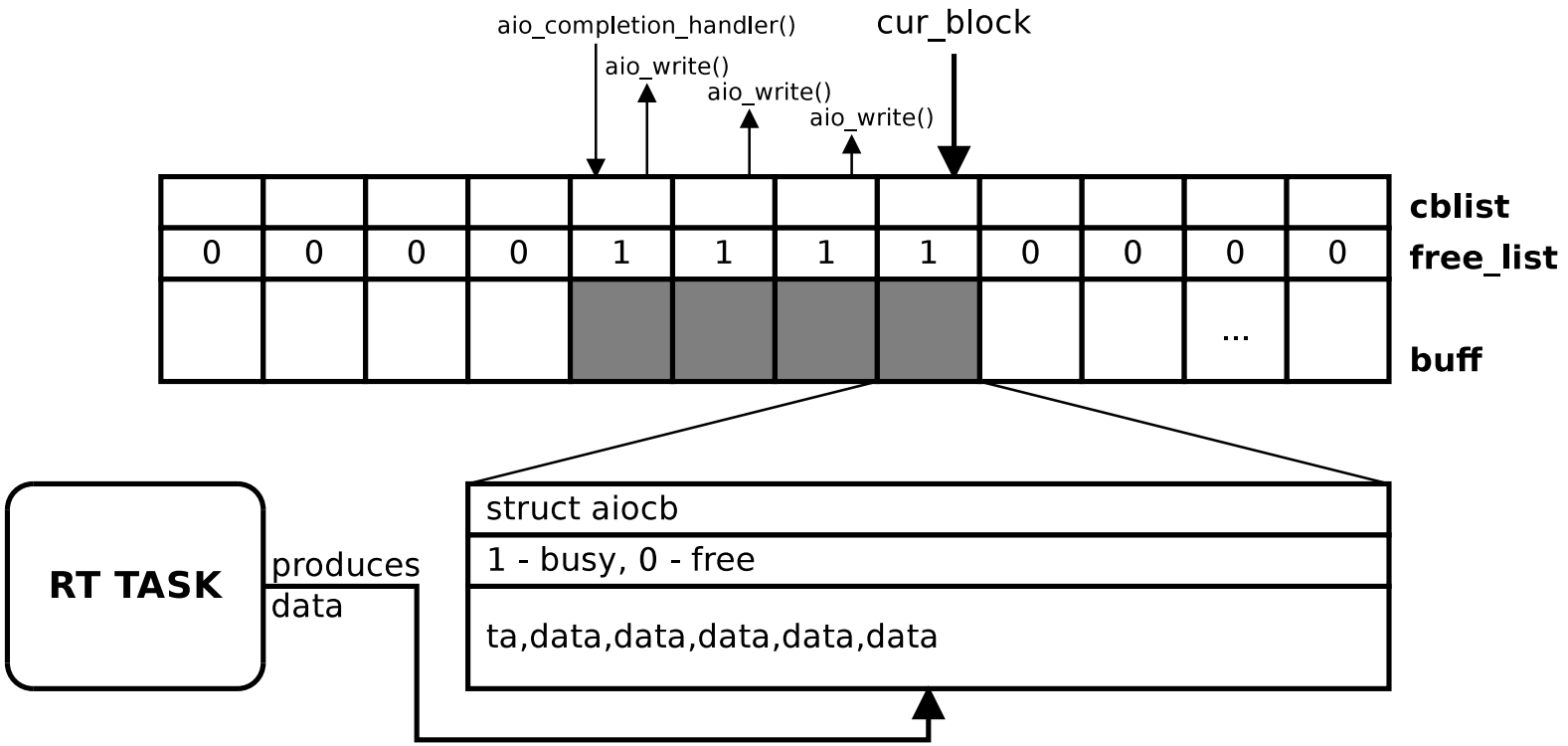
clock_nanosleep(CLOCK_MONOTONIC, 0, &interval, NULL);

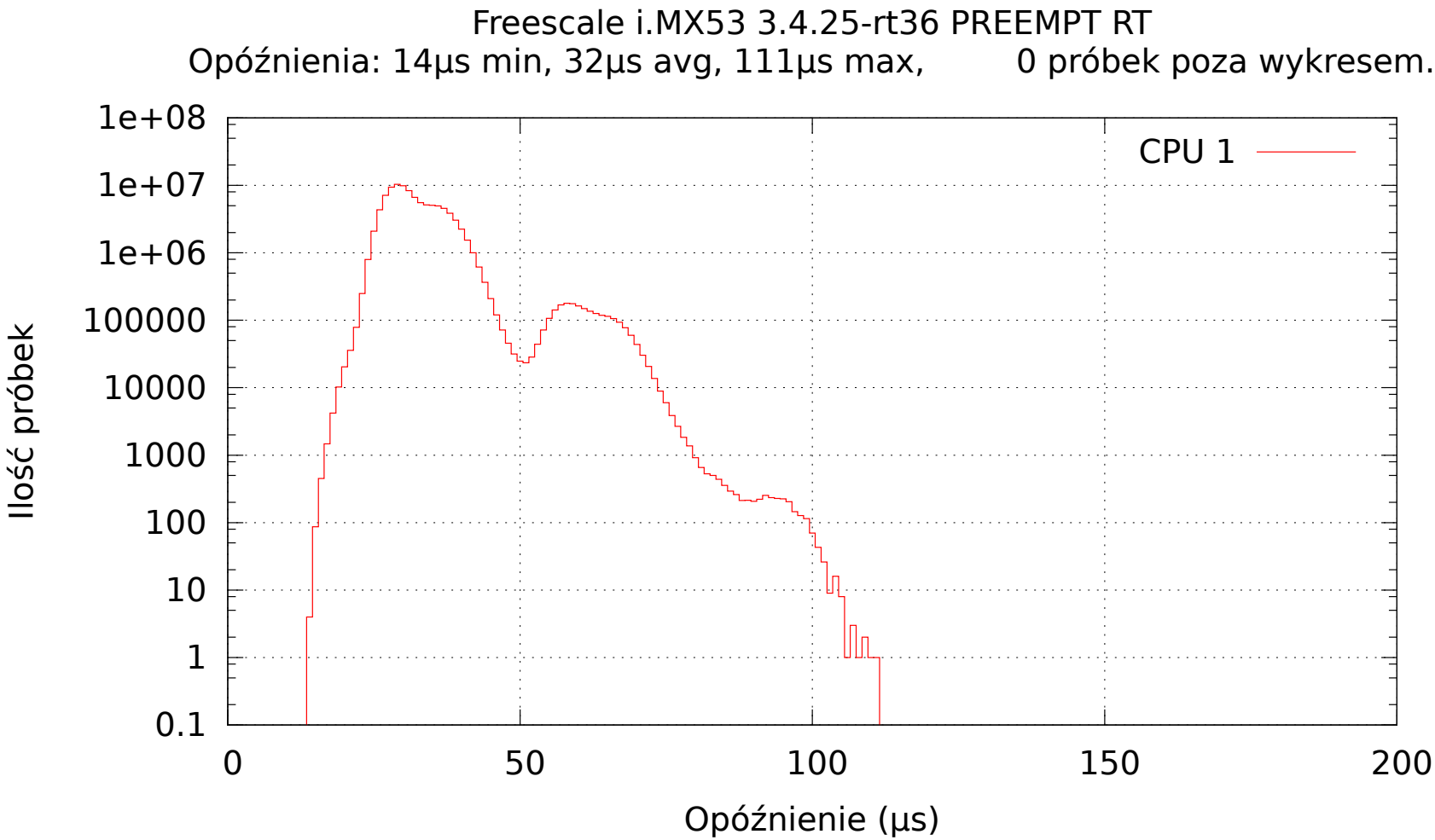


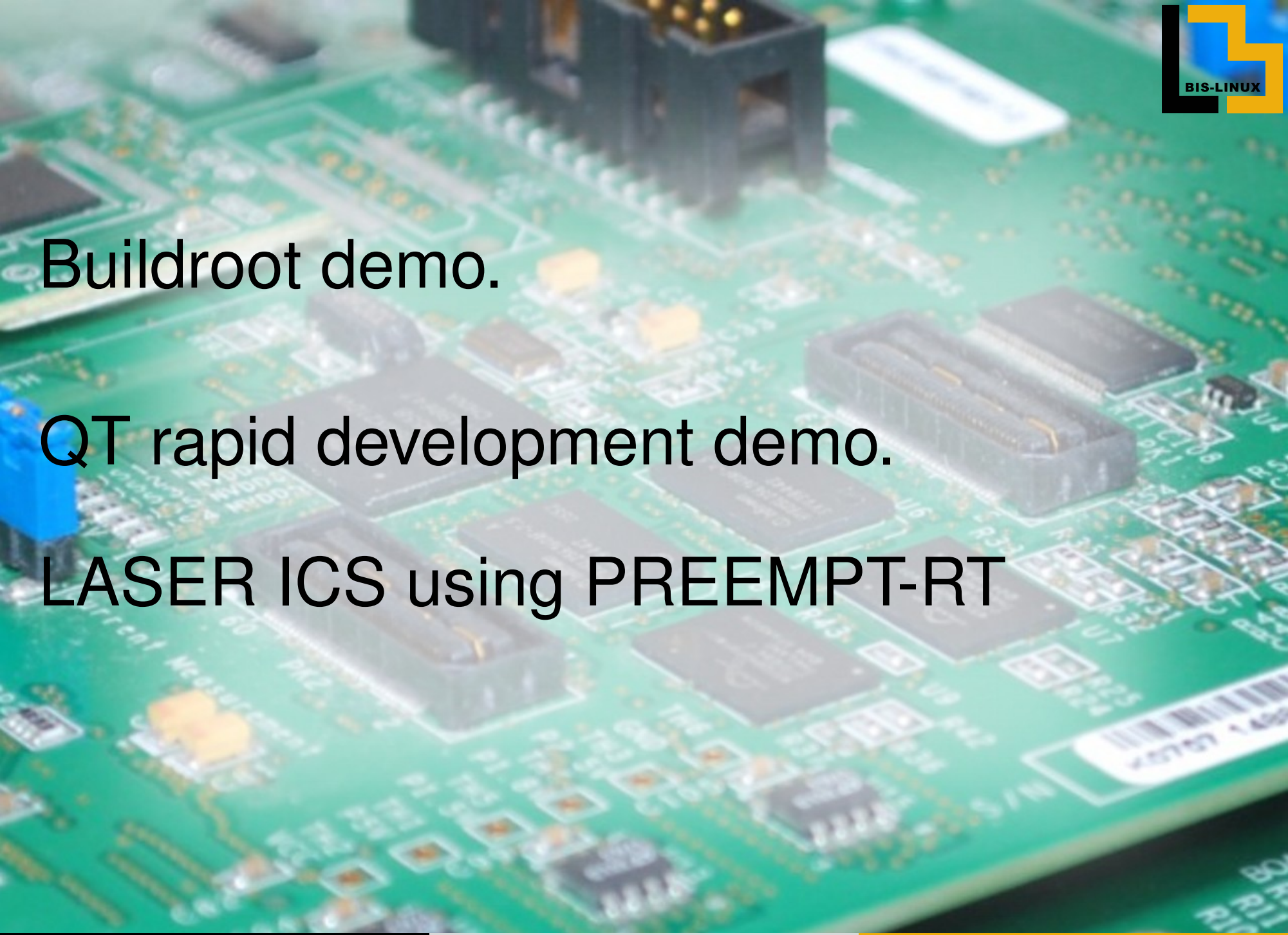
clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &timeout, NULL);

Utilize AIO to write or read data (eg. sensor data, production logs)

```
struct aiocb {
    int             aio_fildes //File descriptor.
    volatile void  *aio_buf    //Location of buffer.
    /* ... */
};
aio_write(struct aiocb *);
aio_return(struct aiocb *);
```


aio_completion_handler()
cur_block
aio_write()
aio_write()
aio_write()
cblist
0 0 0 0 1 1 1 1 0 0 0 0
free_list
...
buff
RT TASK
produces
data
struct aiocb
1 - busy, 0 - free
ta,data,data,data,data,data


Freescale i.MX53 3.4.25-rt36 PREEMPT RT
Opóźnienia: 14µs min, 32µs avg, 111µs max, 0 próbek poza wykresem.
CPU 1
Ilość próbek
1e+08
1e+07
1e+06
100000
10000
1000
100
10
1
0.1
0
50
100
150
200
Opóźnienie (µs)



Buildroot demo.

QT rapid development demo.

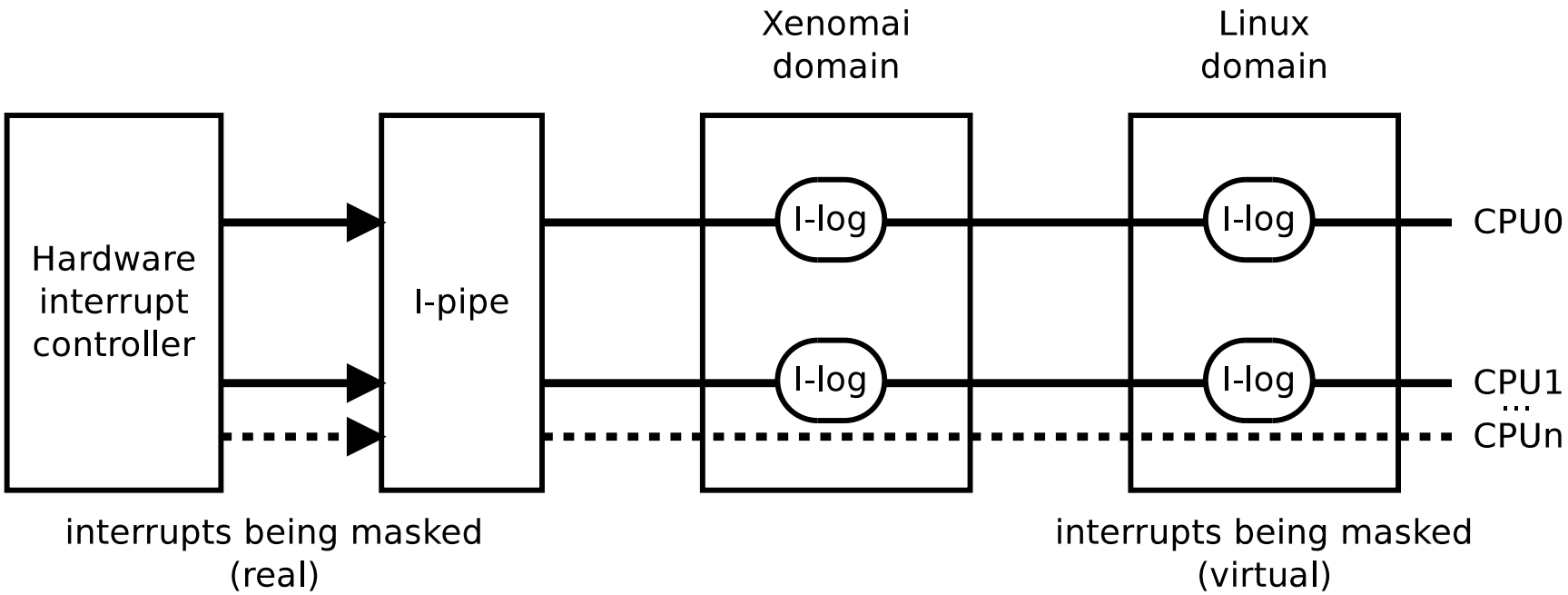
LASER ICS using PREEMPT-RT

ICS using Xenomai

Plan

6 ICS using Xenomai
- How does it work?
- Internal latency measurements
- Demo

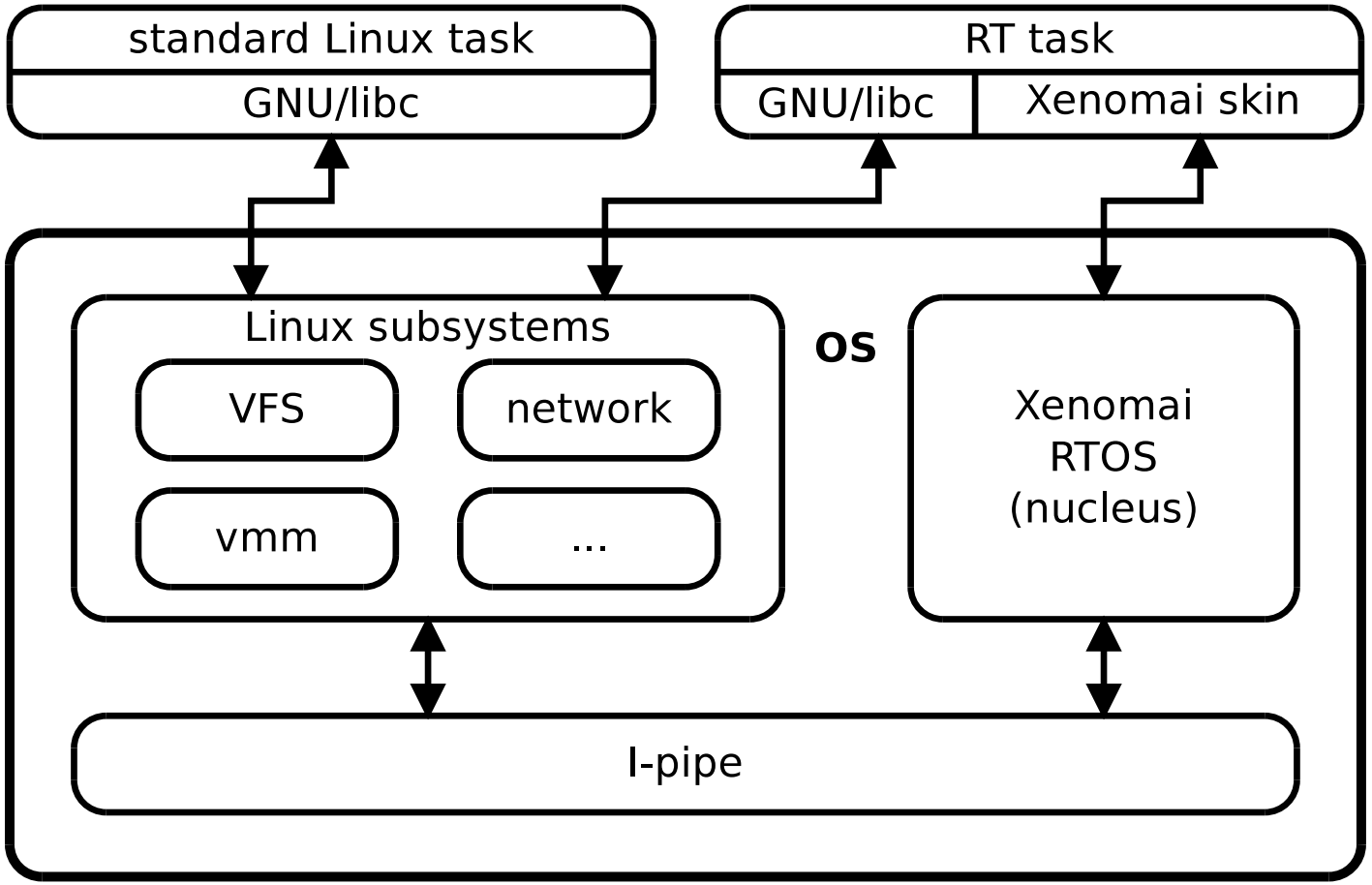
Adeos/I-Pipe



- I-Pipe take control over all hardware interrupts
- All system calls are passed through (I-Pipe)
- Events are dispatched to different I-pipe domains.

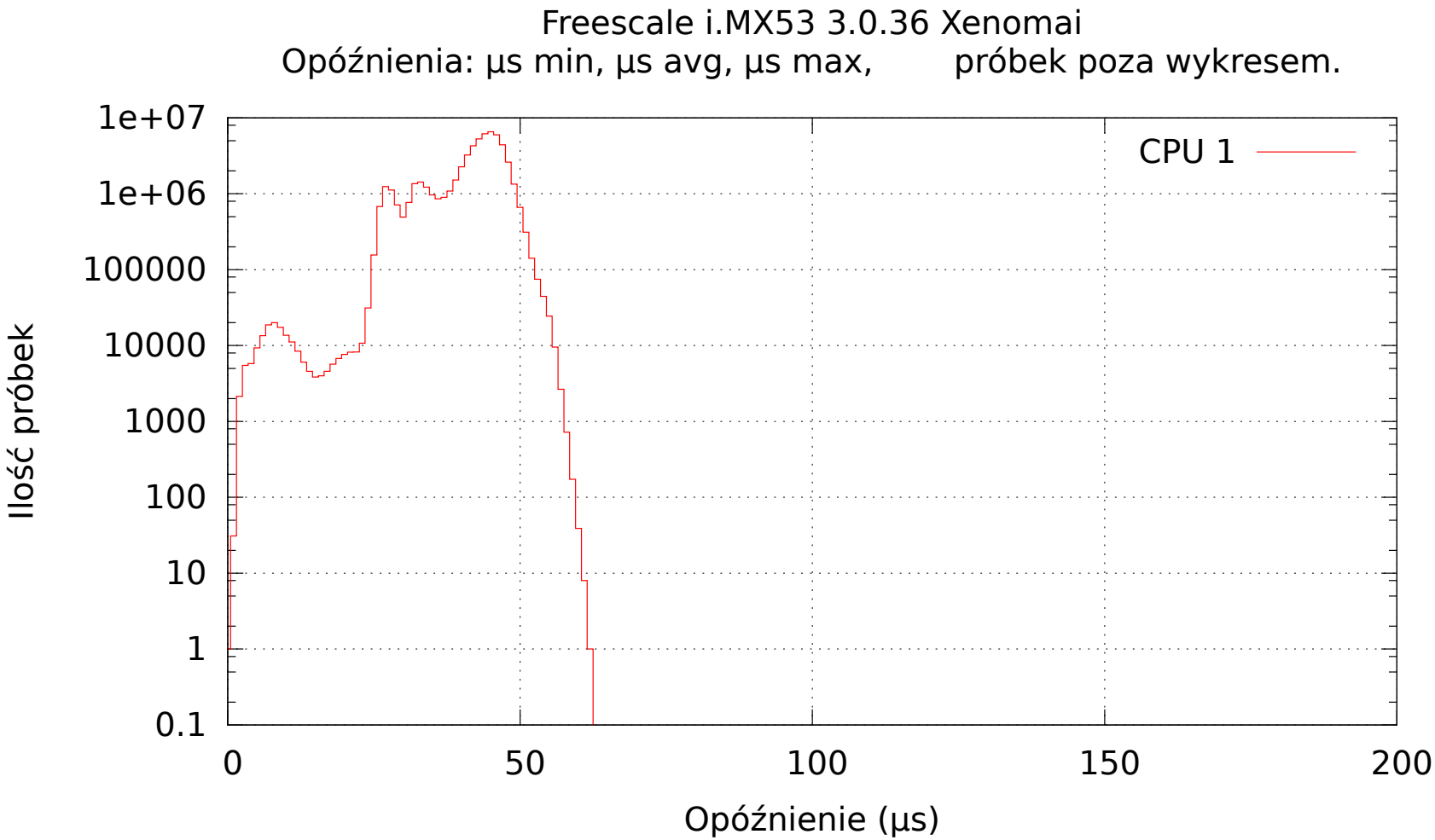
```
$ cat /proc/ipipe/Xenomai
       +----- Handling ([A]ccepted, [G]rabbed,
       |+---- Sticky          [W]ired, [D]iscarded)
       ||+--- Locked
       |||+-- Exclusive
       ||||+- Virtual
[IRQ]  |||||
  38:  W..X.
 418:  W...V
[Domain info]
id=0x58454e4f
priority=topmost

$ cat /proc/ipipe/Linux
   0:  A....
   1:  A....
...
priority=100
```

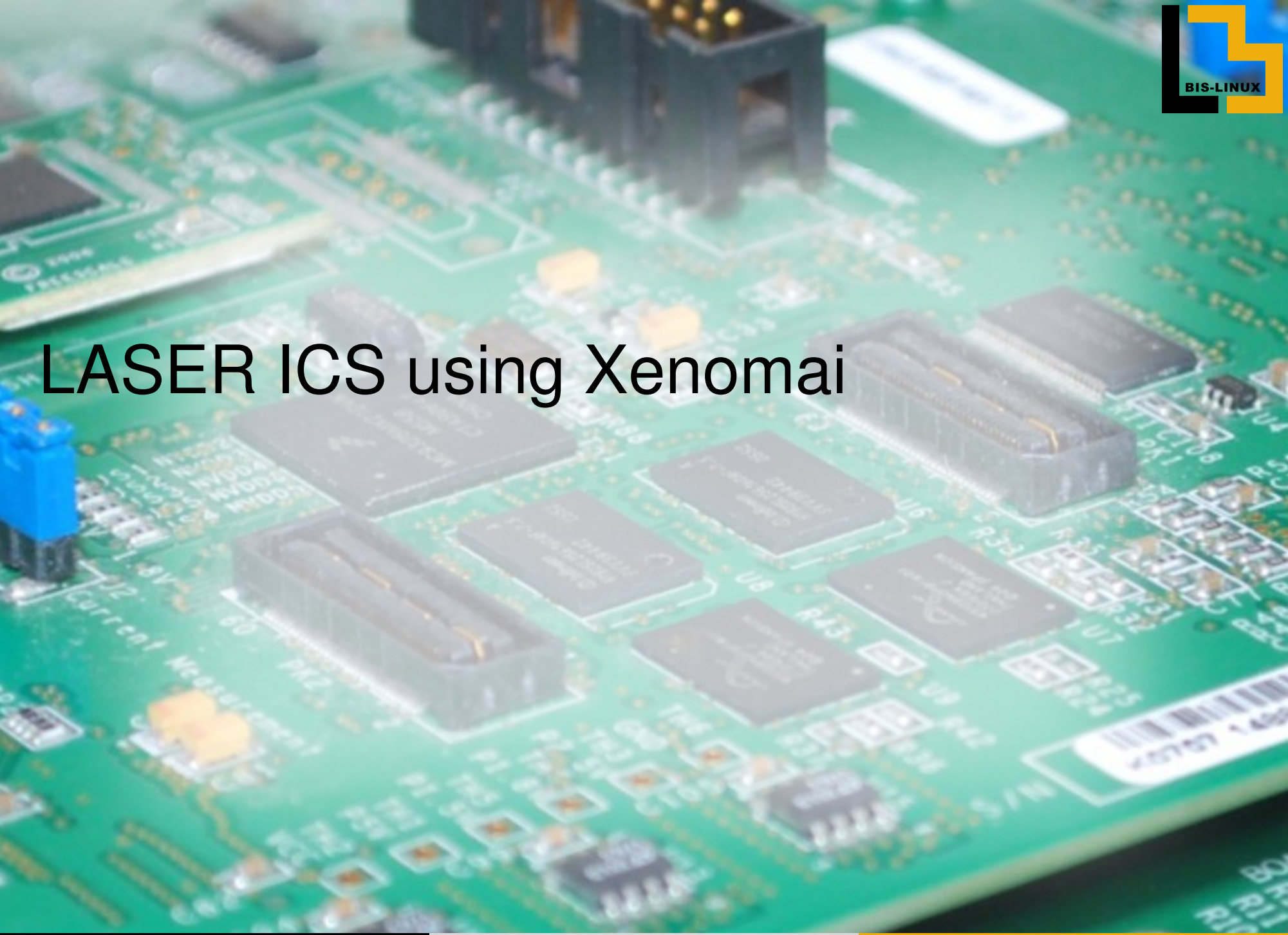
Xenomai



There are actually two kernels. Process can migrate between them:

- Xenomai
- Linux


Freescale i.MX53 3.0.36 Xenomai
Opóźnienia: µs min, µs avg, µs max, próbek poza wykresem.
CPU 1
Ilość próbek
1e+07
1e+06
100000
10000
1000
100
10
1
0.1
0
50
100
150
200
Opóźnienie (µs)



LASER ICS using Xenomai

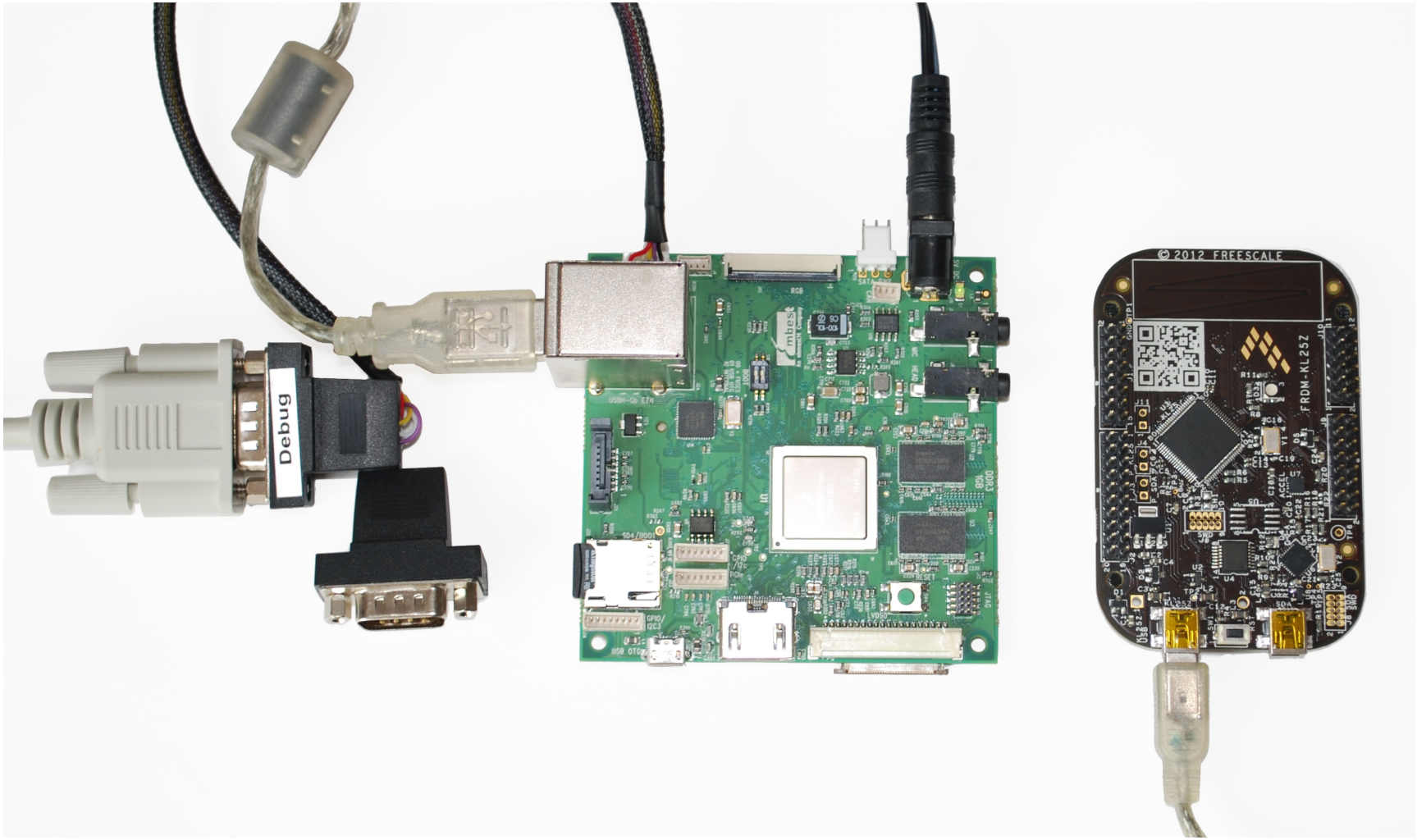
ICS using multiple processors (cores)

Plan

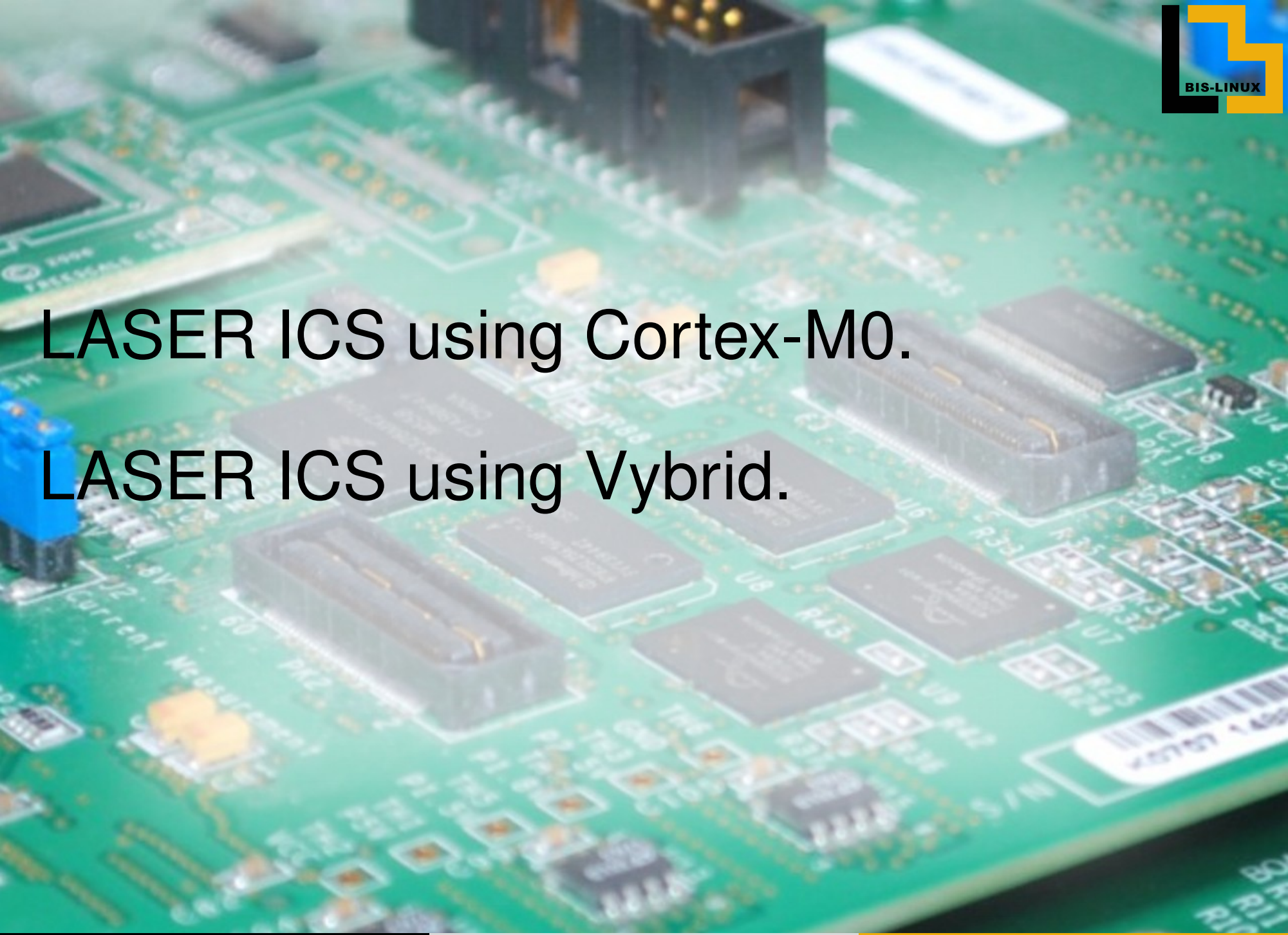
7 ICS using multiple processors (cores)
Demo

- Run MQX-Lite on Cortex-M0 to do the real-time stuff. Run Linux on more powerfull processor.
- **Vybrid**: Linux runs on Cortex-A5, MQX on Cortex-M4

Vybrid




WE
7499010211A
WE
7499010211A



LASER ICS using Cortex-M0.

LASER ICS using Vybrid.

Thank You!

http://bis-linux.com/dwf_waw2013

This presentation, source code and additional materials.

http://bis.org.pl

My books.

http://bis-linux.com/

Embedded Linux and Real-Time - training and support.

Contact

marcin@bis.org.pl

Questions?