BIS-LINUX.COM

ELCE 2012
Real-Time Linux in Industrial Appliances

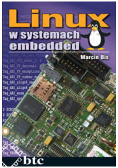
Martin Bis

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

November 2012

About me

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



Booming usage of Linux in embedded environments

- Linux (and Open Source) is being increasingly used in industrial control devices (previously reserved for classic RTOSes).
  - Hardware capable of running Linux become cheaper.
  - Linux supports lots of hardware, as well as communication protocols.
  - Security is easy to achieve (security != safety critical)
  - Code is easy to develop and reusable.
- More and more such applications must work in Real-Time.

Practical case

Weight dosing in injection-molding process (plastics industry).
Application based on TNKernel.

- `http://www.tnkernel.com/`
- compact and very fast real-time kernel for the embedded 32/16/8 bits microprocessors (Atmel AT91 chip in our case).
- Open-Source software
- Elegant code.
- Well tested (many previous success-stories).

But: - customer needs:

- a touchscreen
- logs pulled via FTP
- remote control
- „nicer icons”
- to connect a barcode scanner

Too difficult to implement (within a reasonable time).
Out of the box in Linux (+userspace) - but will it fit?

What Real-Time really means?

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.

And in practice

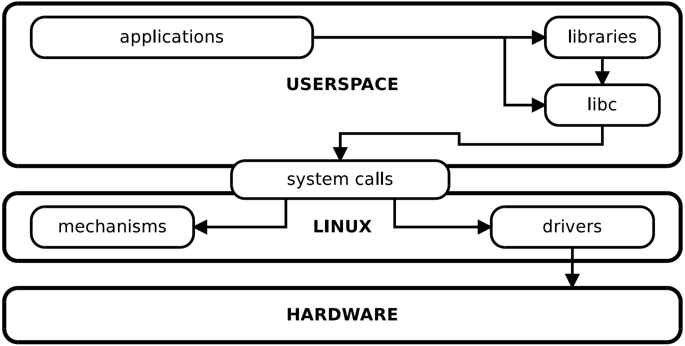
Practical definition:

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!

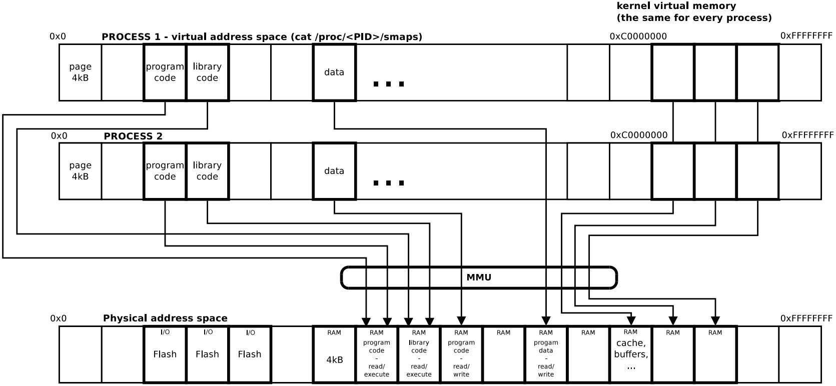
Real-Time vs. Real-Fast

- 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.



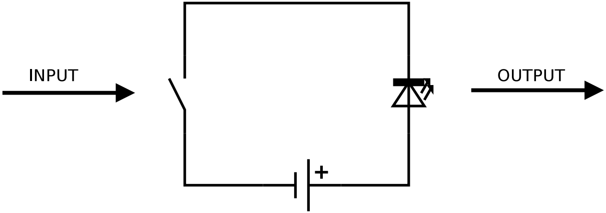
Real-Time vs. Real-Fast ...

Most layers and subsystems are complex:

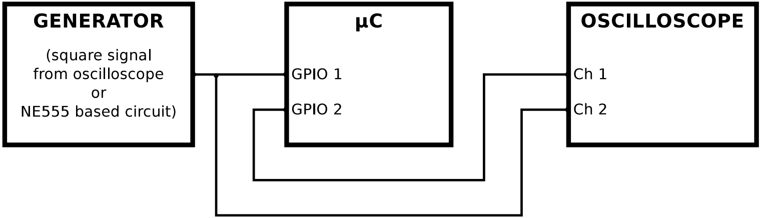


Latency tests

What are we testing?

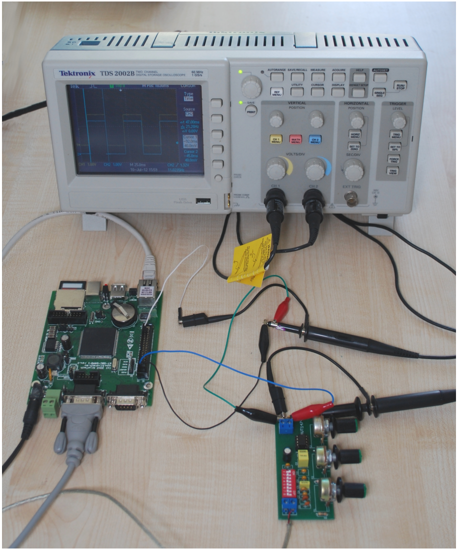


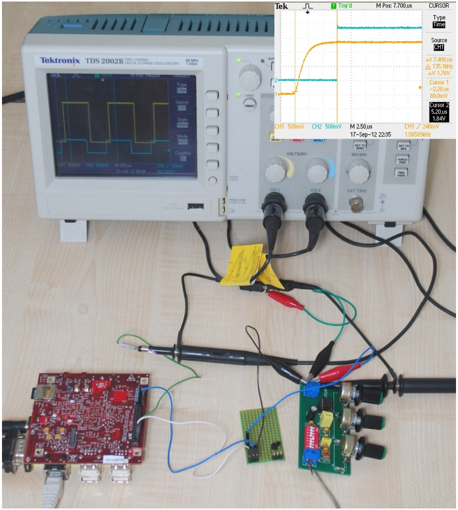
Reference circuit

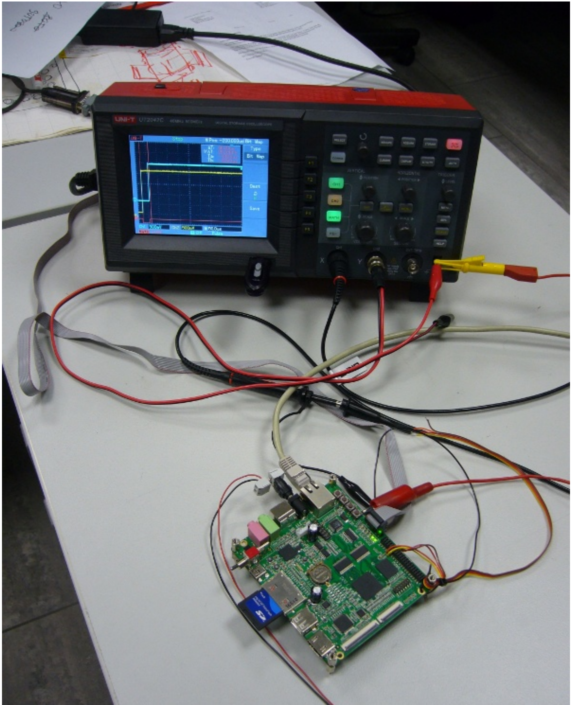


Input is triggered on falling and rising edge, output state changes according to input.

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

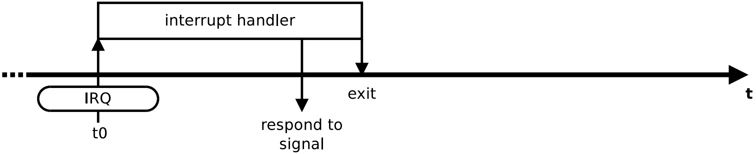

Tektronix
TDS 2002B


Tektronix
TDS 2002B
Tek
Trig'd
M Pos: 7.700μs
CURSOR
Type
Time
Source
CH1
Δt 7.400μs
1/Δt 135.1kHz
ΔV 1.76V
Cursor 1
−2.20μs
80.0mV
Cursor 2
5.20μs
1.84V
CH1 500mV
CH2 500mV
M 2.50μs
17−Sep−12 22:35
CH1 ⟋ 240mV
1.56589kHz

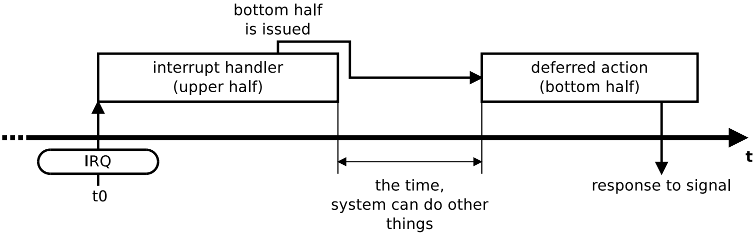

UNI-T
UT2042C
Type
Bit Map
Save

Driver design

01_inout.c

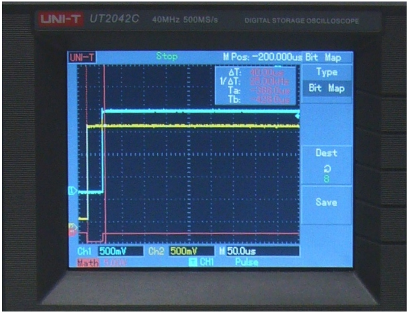


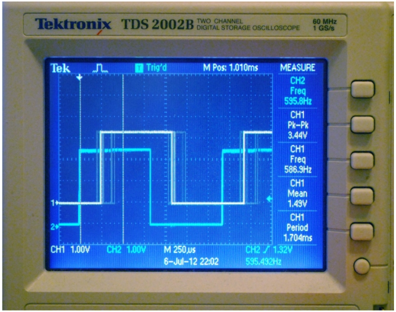
02_uinout.c


bottom half is issued
interrupt handler (upper half)
deferred action (bottom half)
IRQ
t0
the time, system can do other things
response to signal
t

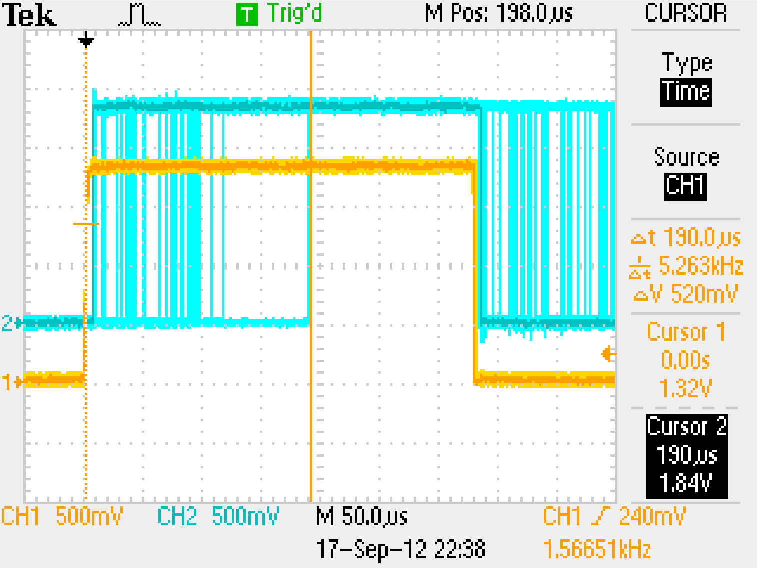
Code is on GitHub:

`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
ΔT:
1/ΔT:
Ta:
Tb:
Dest
8
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
1
2
CH1 1.00V
CH2 1.00V
M 250µs
6-Jul-12 22:02
CH2 ∫ 1.32V
595.492Hz

Results



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

CH1 500mV CH2 500mV M 50.0μs CH1 ⁄ 240mV

17-Sep-12 22:38

1.56651kHz

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:

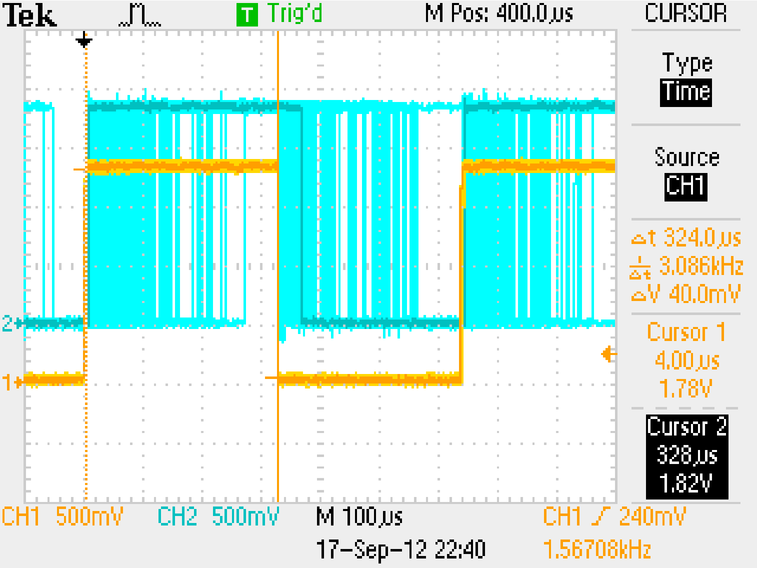
  ```
  cat /dev/zero | od -v
  ```

- 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

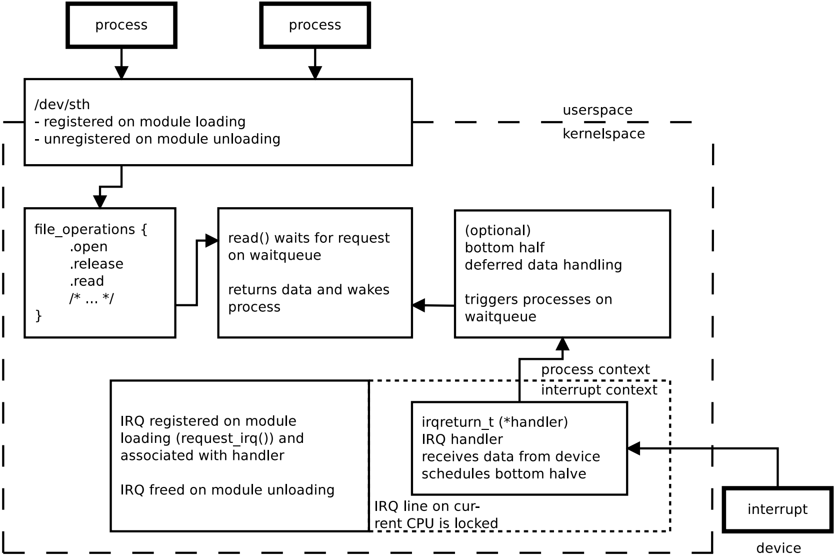


Linux:

Separates:

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

Interrupt-based I/O



process

process

/dev/sth
- registered on module loading
- unregistered on module unloading

userspace

kernelspace

file_operations {
        .open
        .release
        .read
        /* ... */
}

read() waits for request on waitqueue

returns data and wakes process

(optional)
bottom half
deferred data handling

triggers processes on waitqueue

process context

interrupt context

IRQ registered on module loading (request_irq()) and associated with handler

IRQ freed on module unloading

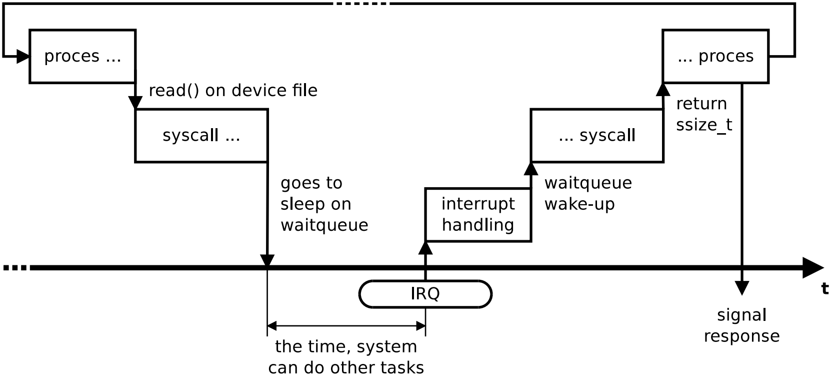
irqreturn_t (*handler)
IRQ handler
receives data from device
schedules bottom halve

IRQ line on current CPU is locked

interrupt

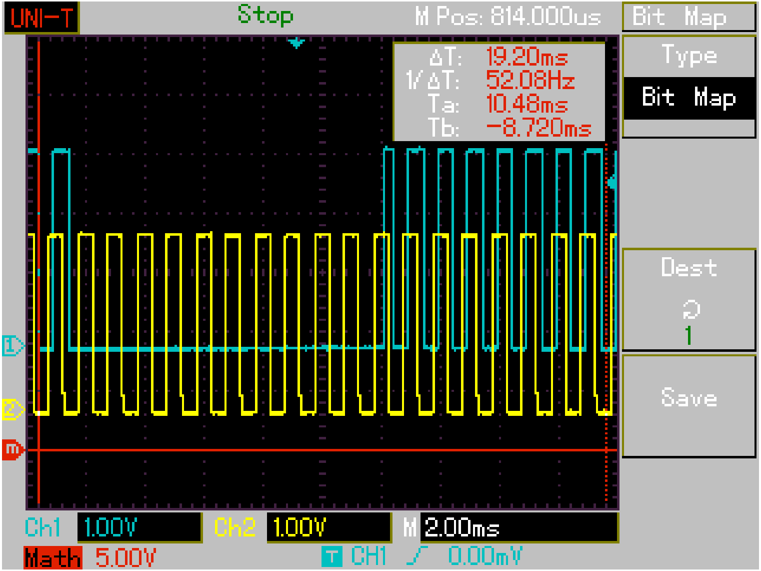
device

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



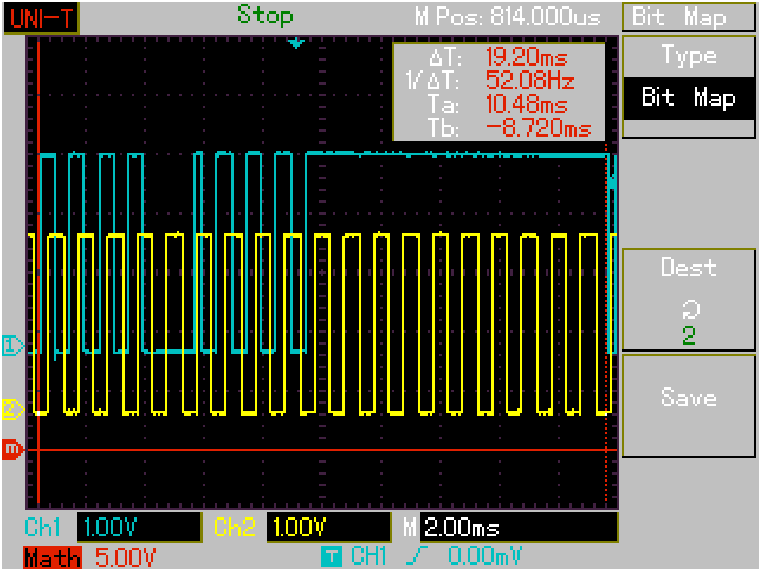
UNI-T Stop M Pos: 814.000us

ΔT: 19.20ms
1/ΔT: 52.08Hz
Ta: 10.48ms
Tb: -8.720ms

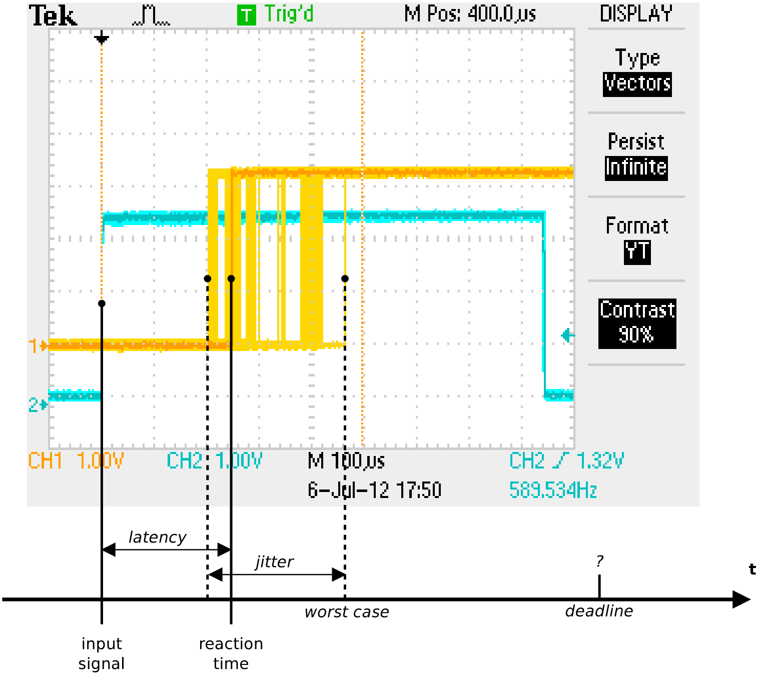
Ch1 1.00V Ch2 1.00V M 2.00ms

Math 5.00V T CH1 ⌐ 0.00mV

Userspace - results under load



Real-Time



Tek Trig'd M Pos: 400.0µs DISPLAY

Type Vectors

Persist Infinite

Format YT

Contrast 90%

CH1 1.00V CH2 1.00V M 100µs CH2 ⟋ 1.32V

6-Jul-12 17:50 589.534Hz

Concepts

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

Jitter

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

Predictability

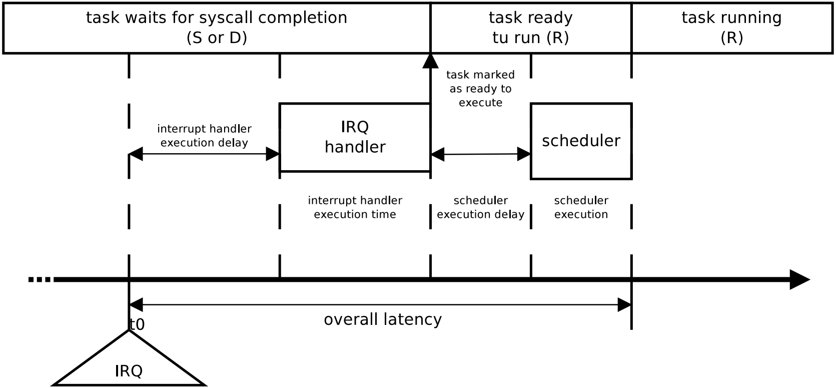
How much time, the action will take (eg. from IRQ occurred to handler finished executing).
O(1) algorithms should be used.

Worst Case

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

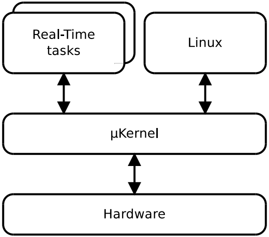
We have to know the latency in worst possible case.

Where is latency coming from?

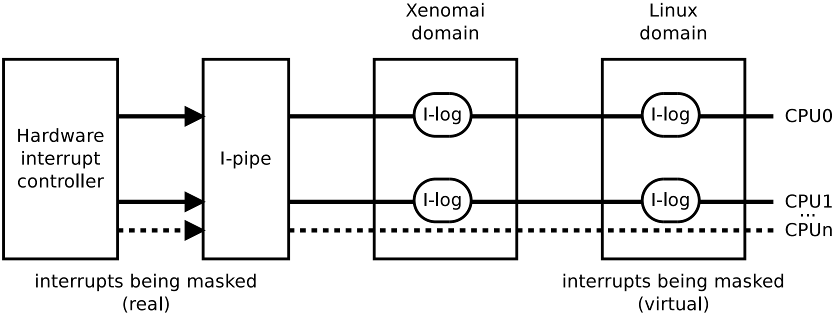


How to achieve Real-Time in Linux?

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/`



Adeos/I-Pipe

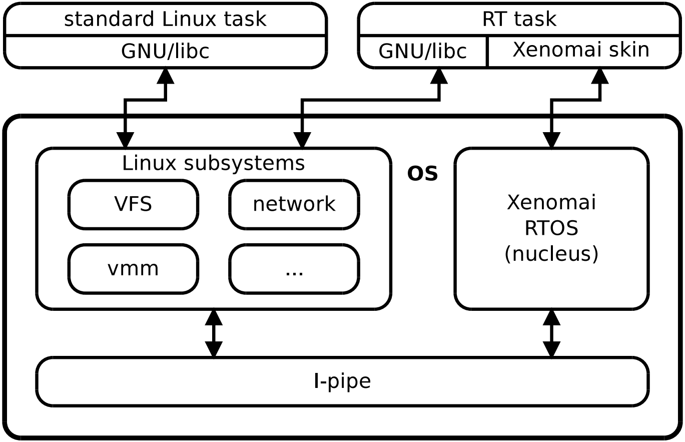

Xenomai domain
Linux domain
Hardware interrupt controller
I-pipe
I-log
I-log
I-log
I-log
CPU0
CPU1
...
CPUn
interrupts being masked (real)
interrupts being masked (virtual)

- 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



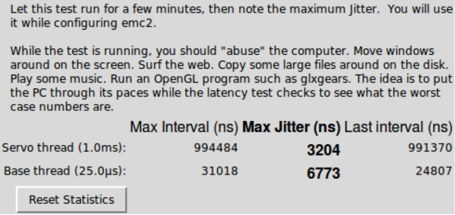
There are actually two kernels:

- Xenomai
- Linux

process (its scheduling) can migrate between them.

RTAI use-case: LinuxCNC

- RTAI demonstration running on PC
- Ubuntu (LiveCD) + RTAI + Applications
  - GUI
  - G language
  - Process visualisation
  - Software PLC (not only for learning)
  - Drivers for certain hardware ...
  - ...or timer-based stepping

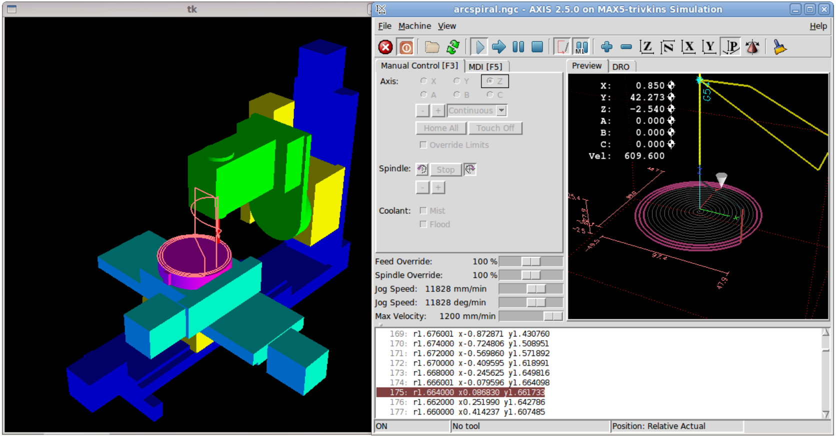


Let this test run for a few minutes, then note the maximum Jitter. You will use it while configuring emc2.

While the test is running, you should "abuse" the computer. Move windows around on the screen. Surf the web. Copy some large files around on the disk. Play some music. Run an OpenGL program such as glxgears. The idea is to put the PC through its paces while the latency test checks to see what the worst case numbers are.

	Max Interval (ns)	**Max Jitter (ns)**	Last interval (ns)
Servo thread (1.0ms):	994484	**3204**	991370
Base thread (25.0µs):	31018	**6773**	24807

LinuxCNC



Axis: ○ X ○ Y ● Z ○ A ○ B ○ C

Continuous

□ Override Limits

Coolant: □ Mist □ Flood

Feed Override: 100 %

Spindle Override: 100 %

Jog Speed: 11828 mm/min

Jog Speed: 11828 deg/min

Max Velocity: 1200 mm/min

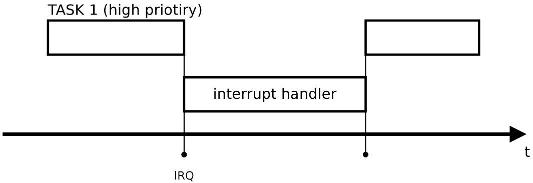
```
  X:    0.850
  Y:   42.273
  Z:   -2.540
  A:    0.000
  B:    0.000
  C:    0.000
Vel:  609.600
```

```
r1.676001 x-0.872871 y1.430760
r1.674000 x-0.724806 y1.508951
r1.672000 x-0.569860 y1.571892
r1.670000 x-0.409595 y1.618991
r1.668000 x-0.245625 y1.649816
r1.666001 x-0.079596 y1.664098
r1.664000 x0.086830 y1.661733
r1.662000 x0.251990 y1.642786
r1.660000 x0.414237 y1.607485
```

ON | No tool | Position: Relative Actual

RT PREEMPT

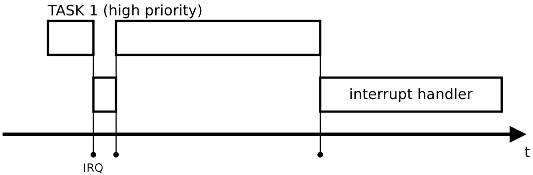
1. Standard kernel



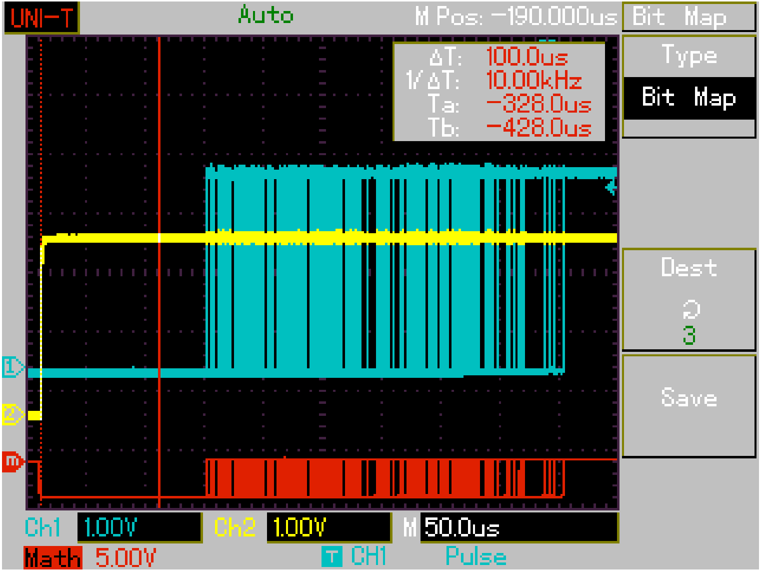
2. Interrupts as threads

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

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

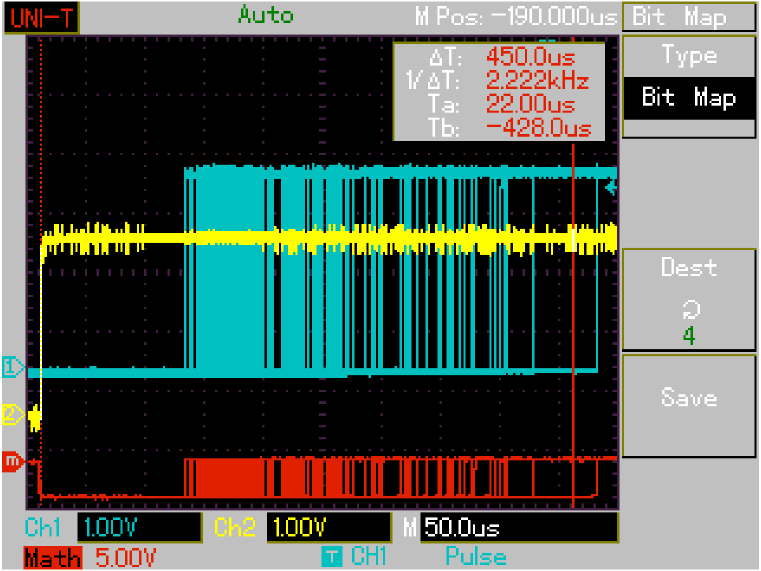


04 under load



ΔT: 100.0us
1/ΔT: 10.00kHz
Ta: -328.0us
Tb: -428.0us

04 under load



UNI-T Auto M Pos: -190.000us

ΔT: 450.0us
1/ΔT: 2.222kHz
Ta: 22.00us
Tb: -428.0us

Ch1 1.00V Ch2 1.00V M 50.0us
Math 5.00V T CH1 Pulse

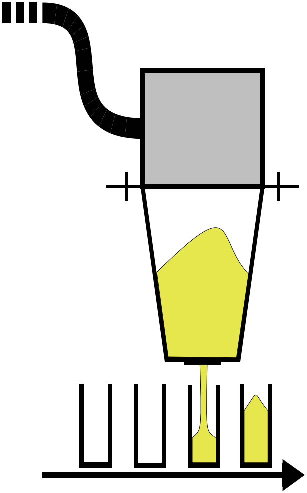
- Real-Time != Real Fast
  Maximum latency (Worst Case) is limited, but minimum latency is bigger.
- Kernel with RT-PREEMPT patch, does not make the whole system Real-Time
- Specially designed application and POSIX RT-API should be used:
  - Defined: `IEEE 1003.1b`. Linux supports it.
  - Scheduler
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

ICS

Industrial control system

Weight-dosing process - specification

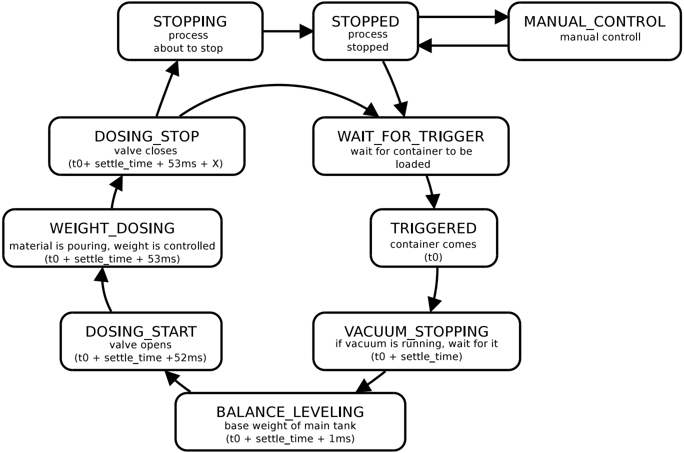
Loose material (or fluid) is loaded into containers.



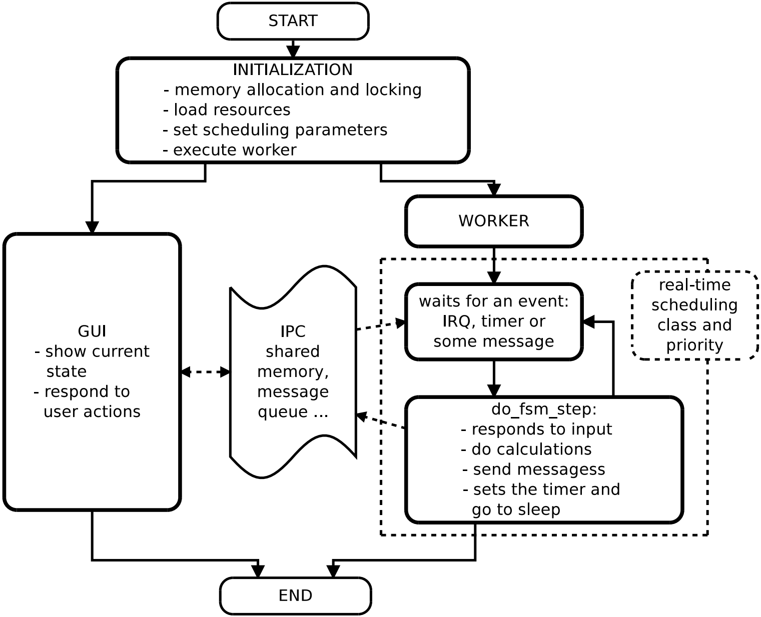
- the main tank is suspended on weight (tensometer)
- conveyor or robot provides containers, appearance of the container triggers interrupt
- the valve opens, and the material is poured into a container
- amount of material is measured by reading data from the weight
- main tank has a limited capacity, 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).



START

INITIALIZATION
- memory allocation and locking
- load resources
- set scheduling parameters
- execute worker

GUI
- show current state
- respond to user actions

IPC
shared memory, message queue ...

WORKER

waits for an event:
IRQ, timer or some message

do_fsm_step:
- responds to input
- do calculations
- send messagess
- sets the timer and go to sleep

real-time scheduling class and priority

END

System boot constraint: must be operational under 10s.

Results: 2.5s RT task, 8-9s GUI.

Weight-dosing process - implementation

Hardware

- PC for development, Debian GNU/Linux 6
- Custom AT91SAM9263 board for production, 2.6.33.7.2-rt30 `http://www.osadl.org/` *Latest Stable*
- KT-SBC-SAM9-1, BeagleBoard-xM for testing, 2.6.33 and 3.0

GUI part

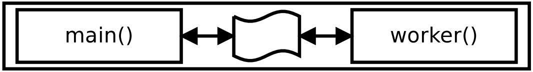
- written in QT/C++
- userspace components provided by: Buildroot
- ext4 on SD card as primary storage
- optimized bootlader

Real-Time process (PREEMPT-RT)

- implemented as separate process in C
- communicates with GUI using POSIX 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

Other cases...

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



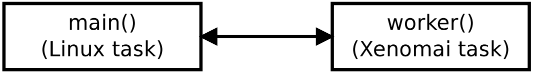
main()

worker()

Threads - share virtual memory, have different scheduling settings.

Welding machine - µs

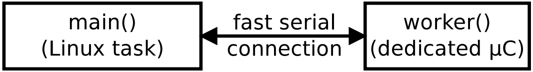
Medical laser controller - µs (or even less)



main()
(Linux task)

worker()
(Xenomai task)

Xenomai provides better latency and predictability.



main()
(Linux task)

fast serial
connection

worker()
(dedicated µC)

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)

Tips&Tricks

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.

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.

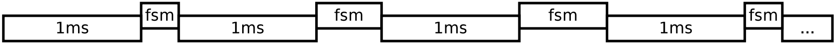
Prefault the stack (it can be shared within process we have forked from)

```
/* 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();
```

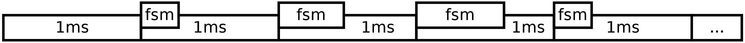
Use POSIX timer to do the fsm step (in a proper way)

```
#define NSEC_IN_SEC 1000000000l
#define INTERVAL 1000000l
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);
}
```



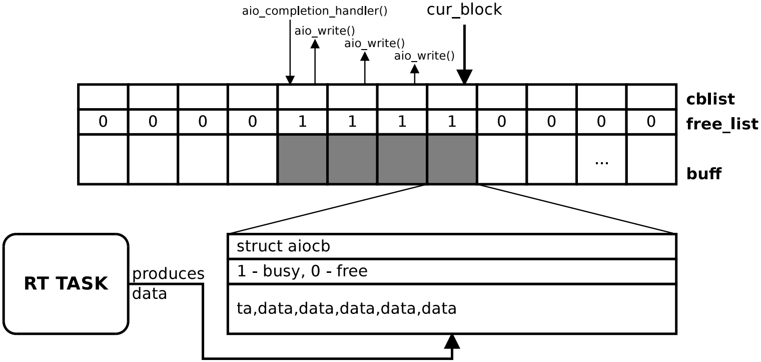
```
clock_nanosleep(CLOCK_MONOTONIC, 0, &interval, NULL);
```



```
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 *);
```



ZATRZYMANIE AWARYJNE. . .

. . . means 'Emergency Stop' in Polish.