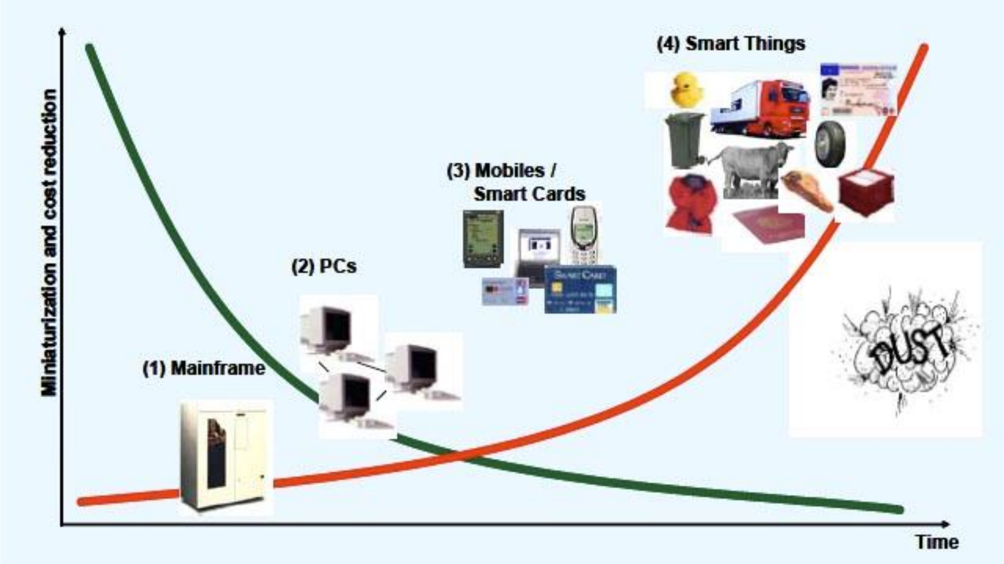
The friendly Operating System for the Internet of Things

TI 3 im WS 2013/2014
Oliver Hahm

An Operating System for what?

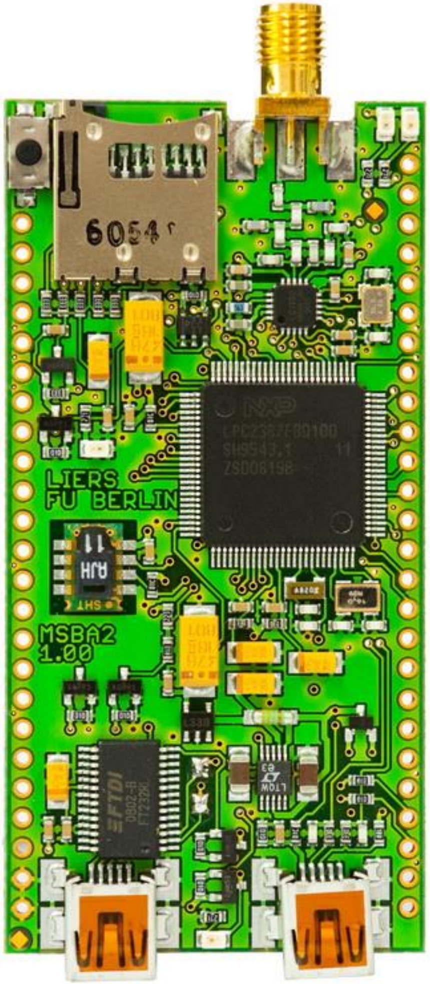
- The vision of IoT: „EveryTHING is connected.“
- Application scenarios:
  - Smart Metering
  - Building Automation
  - Smart City
  - Smart Grid
  - Structural health
  - Logistics

- Every day devices like fridges, coffee machines or watches need to communicate - with each other or to hosts in the Internet



Challenges in the IoT

- Heterogeneous hardware
  - Ranging from 8bit microcontrollers to quite powerful smartphones or routers
  - Various communication interfaces (mostly, but not limited to wireless networks)
- Slow CPU, often no FPU
- Little memory, often no MMU
- Limited energy resources
- Robustness and self-organization
- Real-Time requirements



Operating Systems for WSN and Real-Time Operating Systems

- Typical Real-Time Operating Systems:
  - FreeRTOS
  - QNX
  - RTLinux
- Not designed for energy-efficiency or constrained networks

- Traditional operating systems for WSN:
  - Contiki
  - TinyOS
- Concepts:
  - Event-driven design
  - Single-threaded
  - Specialized programming language

Hello World in TinyOS

```
///////////
#include <stdio.h>
#include <stdlib.h>

module HelloworldM {
provides {
    interface Hello;
}


}
implementation {
    command void Hello.sayhello()
{
        printf("hello world!");
    }
}
```

the hello interface: Hello.nc:

```
///////////
interface Hello{
  command void sayhello();
}
///////////
```

Hello World in Contiki

```
#include "contiki.h"
#include <stdio.h> /* For printf() */
/*------------------------------------------------------------------*/
PROCESS(hello_world_process, "Hello world process");
AUTOSTART_PROCESSES(&hello_world_process);
/*------------------------------------------------------------------*/
PROCESS_THREAD(hello_world_process, ev, data)
{
PROCESS_BEGIN();

printf("Hello, world\n");
PROCESS_END();
}
/*------------------------------------------------------------------*/
```

Hello World in RIOT

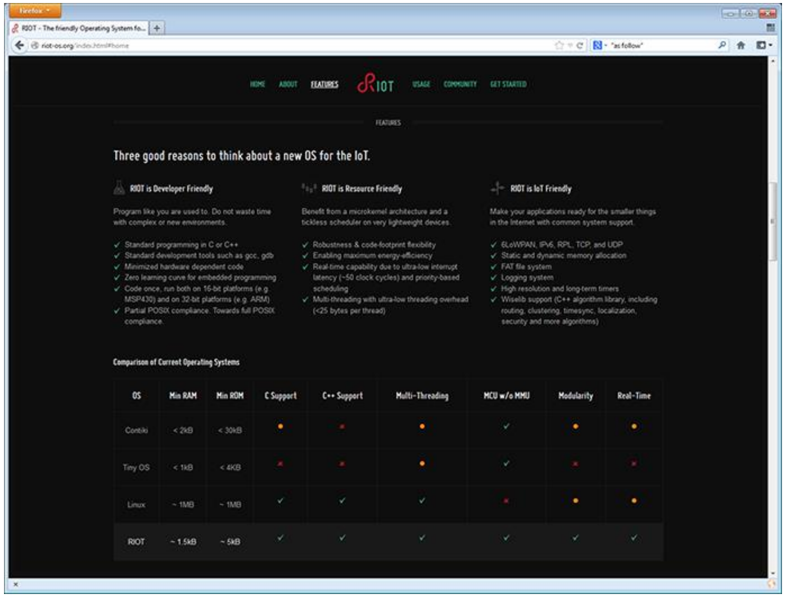
```
#include <stdio.h>

int main(void)
{
  printf("Hello World!\n");

  return 0;
}
```

RIOT: the friendly OS

- Microkernel (for robustness)
- Modular structure to deal with varying requirements
- Tickless scheduler
- Deterministic kernel behaviour
- Low latency interrupt handling
- POSIX like API
- Native port for testing and debugging



Three good reasons to think about a new OS for the IoT.

RIOT is Developer Friendly

Program like you are used to. Do not waste time with complex or new environments.

- ✓ Standard programming in C or C++
- ✓ Standard development tools such as gcc, gdb
- ✓ Minimized hardware dependent code
- ✓ Zero learning curve for embedded programming
- ✓ Code once, run both on 16-bit platforms (e.g. MSP430) and on 32-bit platforms (e.g. ARM)
- ✓ Partial POSIX compliance. Towards full POSIX compliance.

RIOT is Resource Friendly

Benefit from a microkernel architecture and a tickless scheduler on very lightweight devices.

- ✓ Robustness & code-footprint flexibility
- ✓ Enabling maximum energy-efficiency
- ✓ Real-time capability due to ultra-low interrupt latency (~50 clock cycles) and priority-based scheduling
- ✓ Multi-threading with ultra-low threading overhead (<25 bytes per thread)

RIOT is IoT Friendly

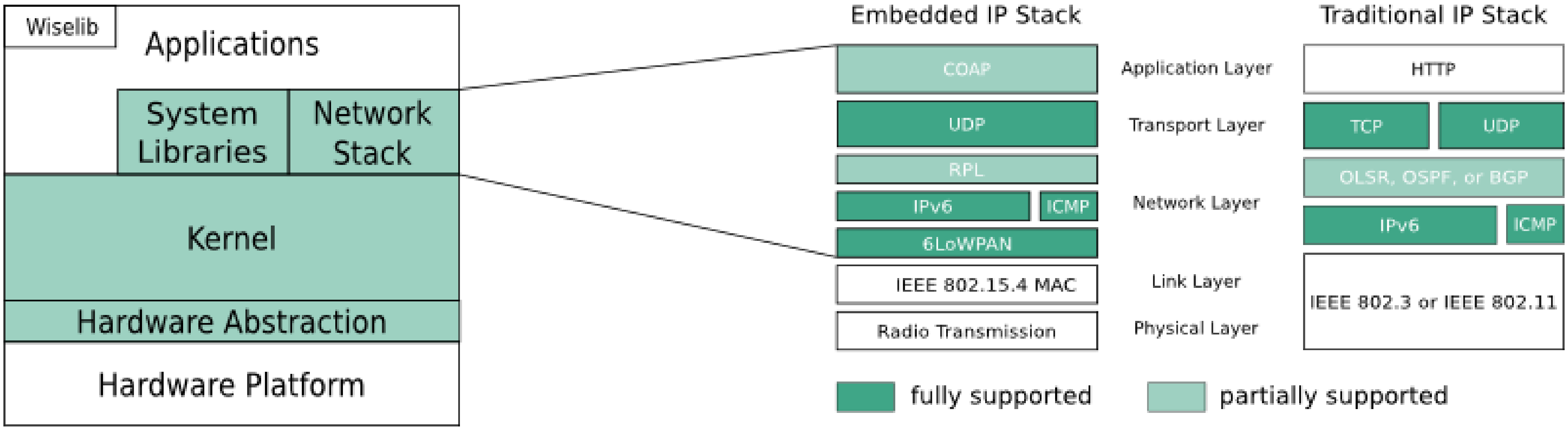
Make your applications ready for the smaller things in the Internet with common system support.

- ✓ 6LoWPAN, IPv6, RPL, TCP, and UDP
- ✓ Static and dynamic memory allocation
- ✓ FAT file system
- ✓ Logging system
- ✓ High resolution and long-term timers
- ✓ Wiselib support (C++ algorithm library, including routing, clustering, timesync, localization, security and more algorithms)

Comparison of Current Operating Systems

OS	Min RAM	Min ROM	C Support	C++ Support	Multi-Threading	MCU w/o MMU	Modularity	Real-Time
Contiki	< 2kB	< 30kB	●	✗	●	✓	●	●
Tiny OS	< 1kB	< 4KB	✗	✗	●	✓	✗	✗
Linux	~ 1MB	~ 1MB	✓	✓	✓	✗	●	●
RIOT	~ 1.5kB	~ 5kB	✓	✓	✓	✓	✓	✓

RIOT structure



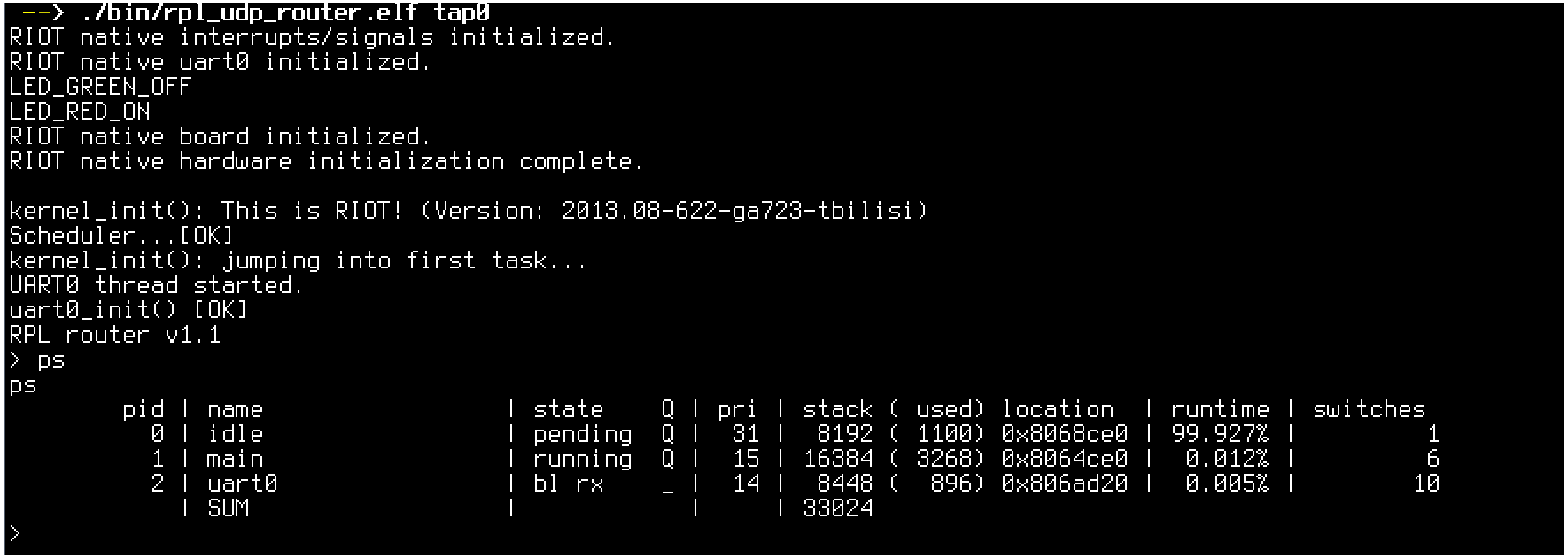
RIOT: hardware support

- CPUs:
  - ARM7
    - NXP LPC2387
    - Freescale MC1322
  - ARM Cortex
    - STM32f103 (Cortex M3)
    - STM32f407 (Cortex M4)
    - NXP LPC1768
  - MSP430
    - MSP430x16x
    - CC430

- Boards:
  - FUB Hardware
    - MSB-A2
    - PTTU
    - AVSExtrem
    - MSB-430(H)
  - TelosB
  - Redbee Econotag
  - WSN430 (Senslab)
  - TI eZ430-Chronos (Watch)
  - AgileFox (FIT testbed)
  - More to come, e.g. mbed hardware

RIOT: the native port

- Run RIOT as is on your Linux computer
- Emulates a network using virtual network devices
- Allows for enhanced debugging with gdb, valgrind, wireshark etc.



```
 --> ./bin/rpl_udp_router.elf tap0
RIOT native interrupts/signals initialized.
RIOT native uart0 initialized.
LED_GREEN_OFF
LED_RED_ON
RIOT native board initialized.
RIOT native hardware initialization complete.

kernel_init(): This is RIOT! (Version: 2013.08-622-ga723-tbilisi)
Scheduler...[OK]
kernel_init(): jumping into first task...
UART0 thread started.
uart0_init() [OK]
RPL router v1.1
> ps
ps
        pid | name                 | state    Q | pri | stack ( used) location  | runtime | switches
          0 | idle                 | pending  Q |  31 |  8192 ( 1100) 0x8068ce0 | 99.927% |         1
          1 | main                 | running  Q |  15 | 16384 ( 3268) 0x8064ce0 |  0.012% |         6
          2 | uart0                | bl rx    _ |  14 |  8448 (  896) 0x806ad20 |  0.005% |        10
            | SUM                  |            |     | 33024
>
```

RIOT: Microkernel

- minimalistic kernel

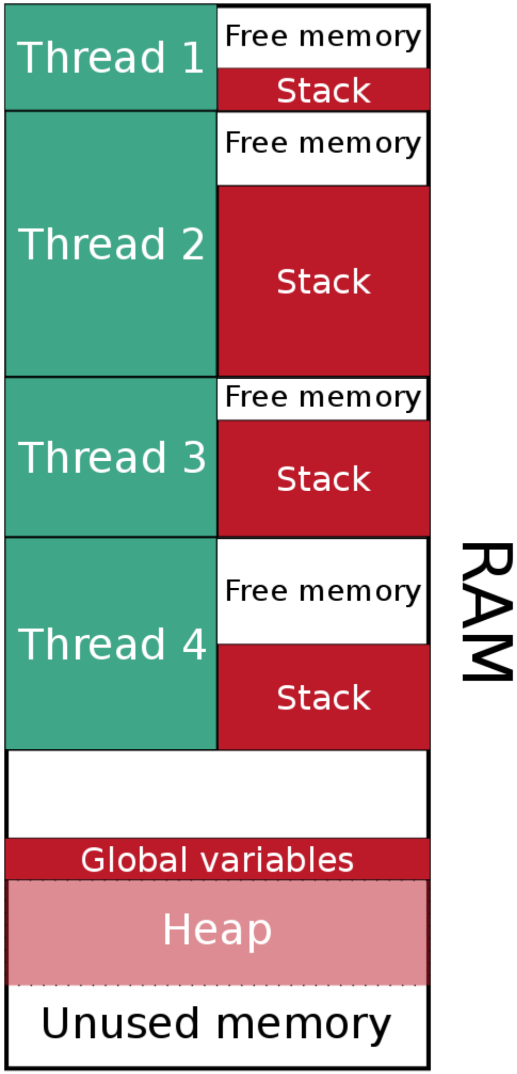
```
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C                               12            350            281           1017
C Header                        22            202            719            377
-------------------------------------------------------------------------------
SUM:                            34            552           1000           1394
-------------------------------------------------------------------------------
```

- Kernel functions:
  - Scheduler
  - IPC
  - Mutexes
  - Timer
- Modules and drivers communicate over IPC
- Deterministic runtime of all kernel functions
- Optimized context switching

RIOT: scheduler

- Tickless, i.e. no periodic timer event
  - ➢ more complex to implement, but most energy-efficient
- Run-to-complete, i.e. scheduler does not distribute equally to all threads
- Priority based
  - ➢ Priorities have to be chosen carefully to fulfill real-time requirements

RIOT: memory management

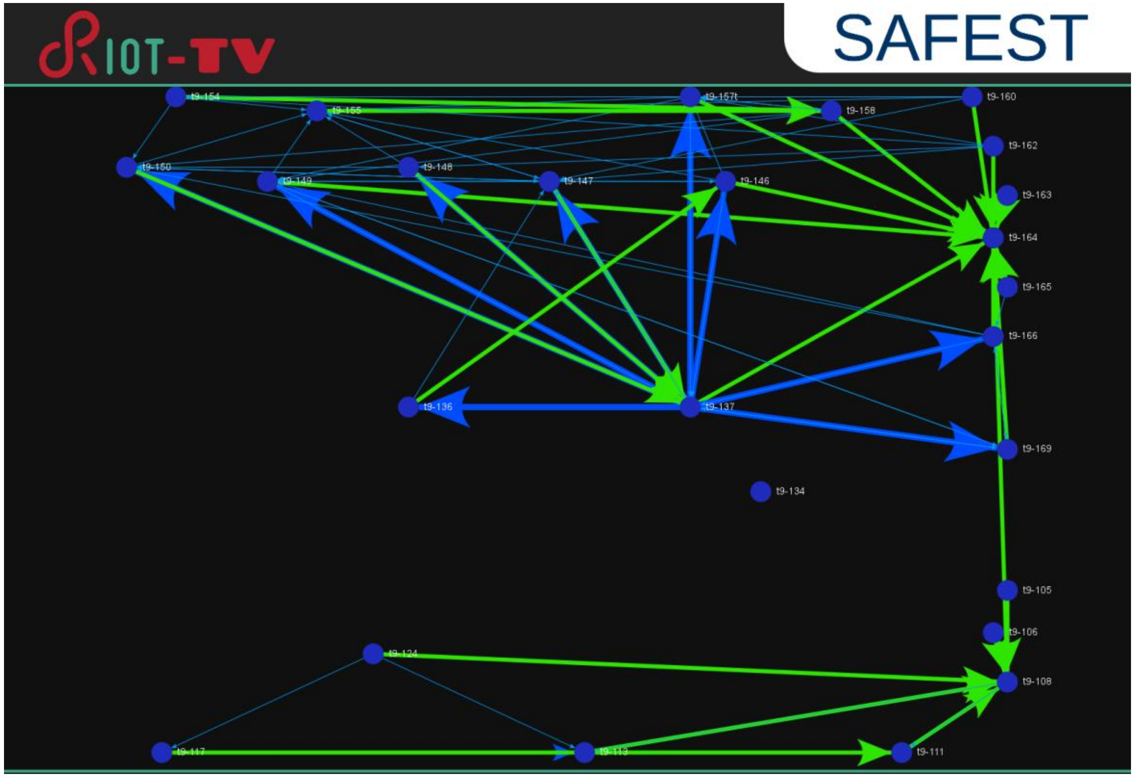


- Every thread has its own stack
- The stack also contains the tcb
- There‘s no memory protection

  => a stack overflow can destroy another stack

RIOT as a tool for research

- Network protocols like 6LoWPAN, RPL, CCN etc.
- Distributed operating systems
- Various testbeds and virtual networks with desvirt
- Measurement of energy consumption



Join the RIOT!

- About 35 forks on Github
  - https://github.com/RIOT-OS/RIOT
  - Start your own fork and contribute to RIOT!
- About 50 people on the developer mailing list
  - devel@riot-os.org
  - users@riot-os.org
- Developers from all around the world
- ~ 200 followers on Twitter