CS-541
Wireless Sensor Networks

Lecture 8: Introduction to WSN programming and Hands-on Session

Spring Semester 2017-2018

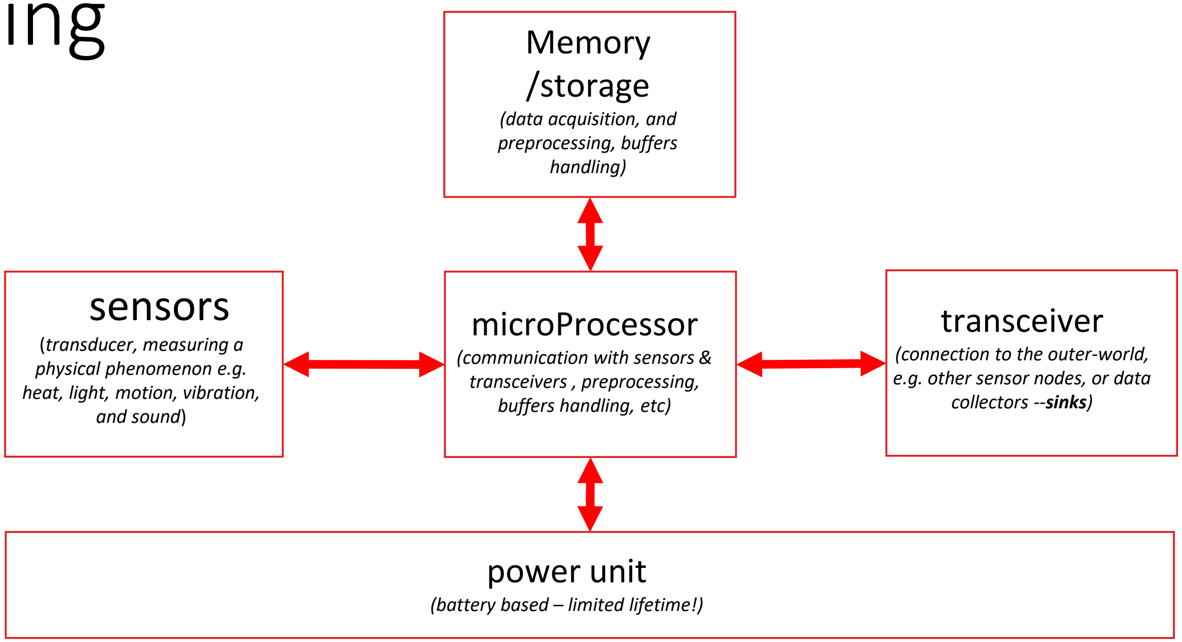
Prof Panagiotis Tsakalides, Dr Athanasia Panousopoulou, Dr Gregory Tsagkatakis

Objectives

- Programming aspects for WSN
- Hands on sessions (Wednesday & Friday B306)

WSN Programming

- HW Platform



Memory /storage
(data acquisition, and preprocessing, buffers handling)

sensors
(*transducer, measuring a physical phenomenon e.g. heat, light, motion, vibration, and sound*)

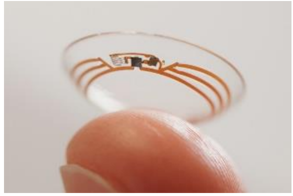
microProcessor
(communication with sensors & transceivers , preprocessing, buffers handling, etc)

transceiver
*(connection to the outer-world, e.g. other sensor nodes, or data collectors --**sinks**)*

power unit
(battery based – limited lifetime!)

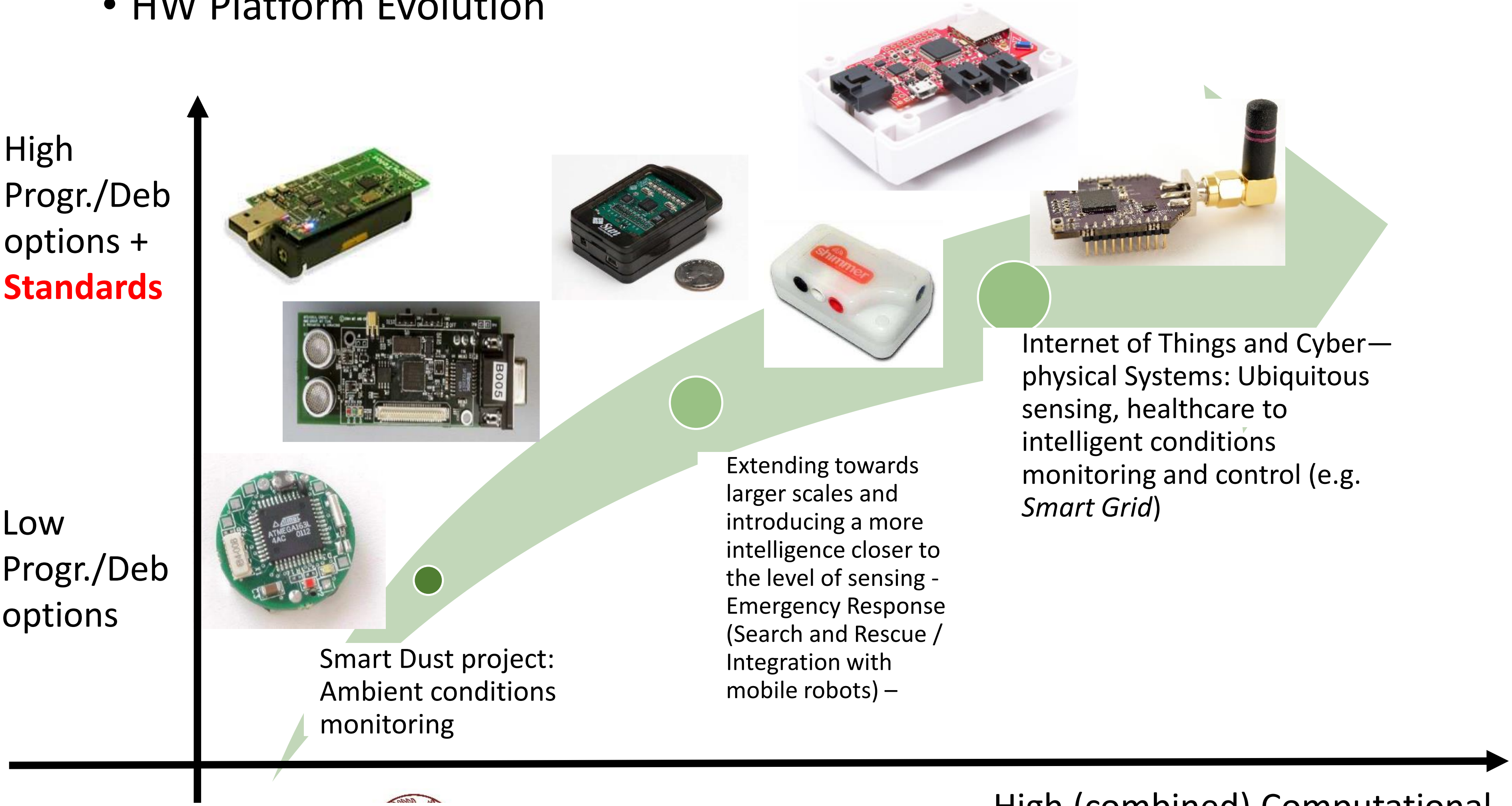
Integration of SW components

Some of them are dedicated to WSN (e.g. sensors, transceivers) and some of them are not e.g. μProcessors, RAM/ROM components



WSN Programming

- HW Platform Evolution



High Progr./Deb options + **Standards**

Low Progr./Deb options

Smart Dust project: Ambient conditions monitoring

Extending towards larger scales and introducing a more intelligence closer to the level of sensing - Emergency Response (Search and Rescue / Integration with mobile robots) –

Internet of Things and Cyber—physical Systems: Ubiquitous sensing, healthcare to intelligent conditions monitoring and control (e.g. *Smart Grid*)

Low Computational Complexity

High (combined) Computational Complexity

Family	Memory	On-board Sensors	Expandability	Notes & Application areas
TELOSB	10KB RAM, 48KB Flash	Temperature, Humidity, Light	10 GIOs, USB programming interface	Open platform. Environmental and health structural monitoring. PoC research projects Open source software support – Active.
Shimmer	10 KB RAM, 48 KB Flash, 2GB µSD	3-axis accelerometer, Tilt & vibration	Expandability for Accelerometers and ECG, EMG. USB mother board.	Research platform with commercial support. Excellent support (open source tools & customized applications). Healthcare and Sports projects (wearable computing) Active and expanding. Rechargeable battery (up to 8hours in fully functional mode)
Zolertia Z1	8K RAM, 92KB Flash	3-axis accelerometer, temperature	52-pin expansion board. Open source community support & commercial support (excellent Wiki)	All WSN-related. One of the latest platforms. Allows the option for a dipole antenna.
XM1000	8K RAM, 116 Flash, 1MB External Flash	Temperature, Humidity, Light	10 GIOs, USB programming interface	from a family of open platforms…. SMA connection (dipole antenna)… All WSN-related, perhaps not for healthcare (bulky size and design). Can last up to 3 weeks on low data rate (per minute).

WSN Programming

Types of programming:

End users:

- Sensor network as a pool of data
- Interact with the network via queries
- Programming language: High-level abstraction, expressive & structured for efficient execution on the distributed platform
- Shielded away from details of how sensors are organized and how nodes communicate.

Application developers:

- Provide the end users with the capabilities of data acquisition, processing, and storage.
- Have to deal with all kinds of uncertainty caused by network, hardware, and real-world imperfections (e.g., noisy, events can happen at the same time, communication and computation take time, communications may be unreliable, battery life)
- Appropriate programming abstractions??

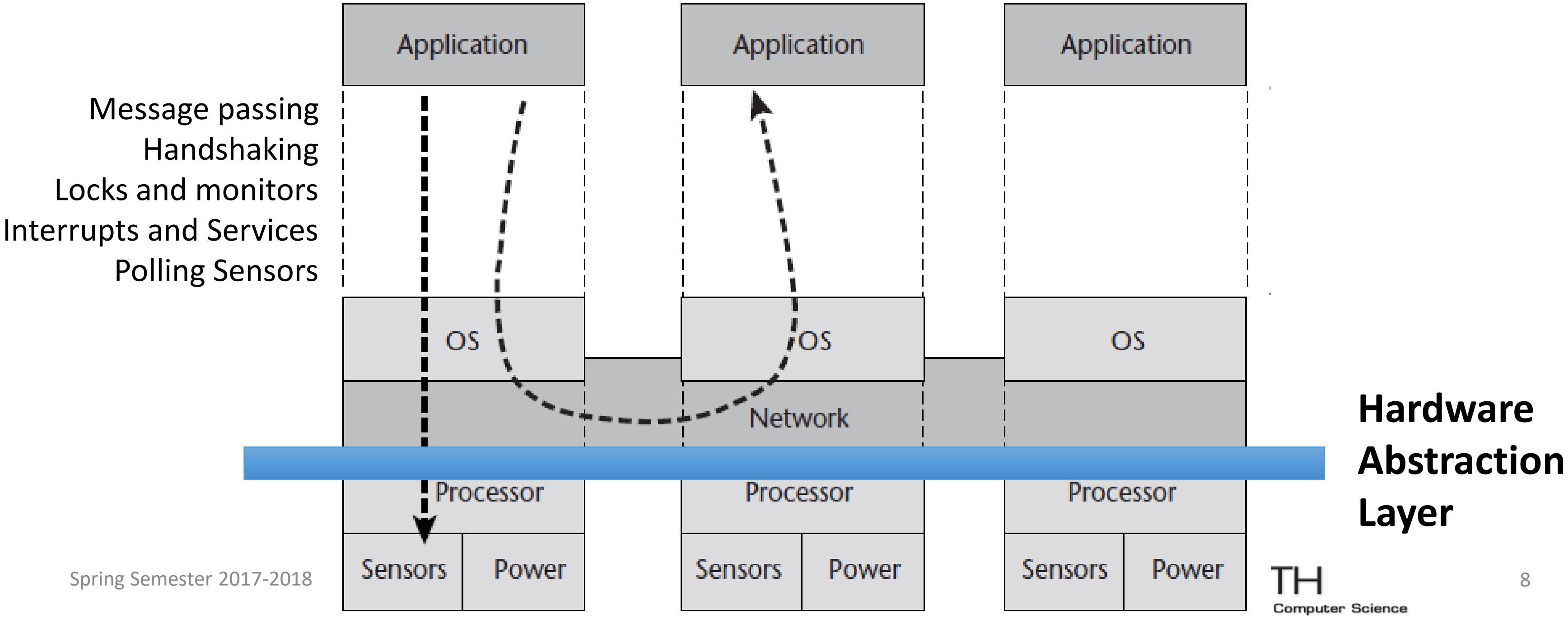
WSN Programming

Application Developers….

- Objective: control its peripheral devices, sample data from the sensors, actuate on demand (or on command) and communicate with the rest of the nodes.
- Challenges:
    - Deal with message passing, event synchronization, interrupt handing, and sensor reading?
    - How to have **access/control** to what each mote is actually doing? Or being able to accurately **emulate** its behavior?
    - How to allow rapid prototyping of common applications and network standards across different types of hardware? **Hardware abstraction**
    - How to do that easily (by using for example widely adopted programming languages and tools?) How to reduce the learning curve?
    - How to fine grain resource management in terms of memory and energy?
    - How to deal with the inherent concurrent / event-drive nature of the applications?

WSN Programming

	Access / Control	Hardware Abstraction	Ease to use
µProcessor Programming	Control / Emulation @ the level of µProcessor	NO	NOT FOR typical WSN practitioner / developer
Real-time Operating Systems	Depends on how well it is designed and what are the supporting tools it provides	YES	Depends on how well it is designed…



WSN Programming

OS:

Abstracts the hardware platform

A set of services for applications: file management, memory allocation, task scheduling, peripheral device drivers, networking.

OS for embedded systems:

The above under the constraint of limited resources:

Different trade-offs when providing these services – application dependent: no file management requirement, if file system is not needed.

No dynamic memory allocation, if memory management can be simplified.

No prioritization among tasks if not critical

OS for WSN:

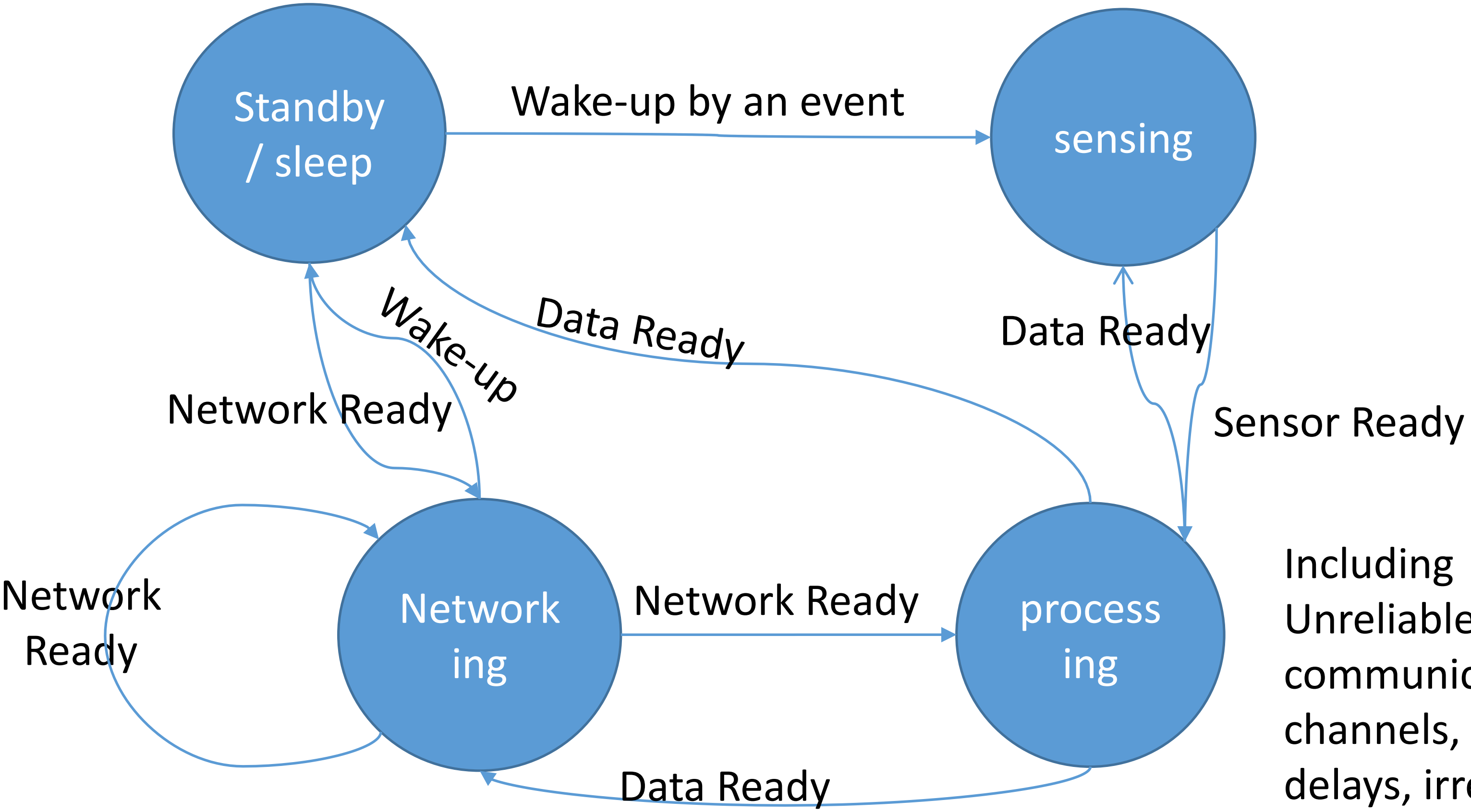
- The above under the challenge of scalability, need to support distributed applications, real-time response to stimuli from the physical environment

Reduce code size
Improve time response
Reduce energy consumption

- Microkernel: modularizing the operating system so that only the necessary parts are deployed with the application.
- RT sched.: allocates resources to more urgent tasks so that they can be finished early.
- Event-driven execution allows the system to fall into low-power sleep mode when no interesting events need to be processed.

WSN Programming

Typical State Machine of a Sensor Node



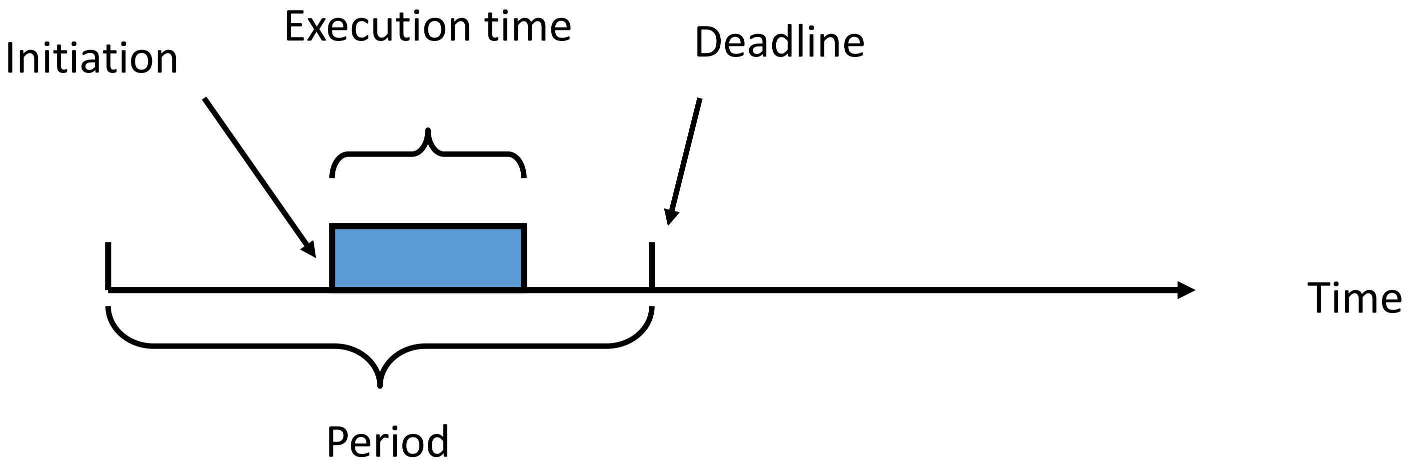
Including Unreliable communication channels, long delays, irregular arrival of messages, simultaneous events

WSN Programming

- Task Model

- a non-preemptive scheduler - a task is executed to the end, may not be interrupted by another task
- a preemptive scheduler - a task of higher priority may interrupt a task of low priority

Each task a triplet: (execution time, period, deadline)



OS For WSN: How to successfully manage the execution of tasks related to the sensors and the protocol stack? (Interrupts, Services, & Polling)

WSN Programming

Types of Architectures for OS For WSN

Monolithic

- Services provided by an OS are implemented separately and each service provides an interface for other services.
- A single system image & smaller OS memory footprint.

++Low module interaction costs are low.
--The system is hard to understand and modify, unreliable, and difficult to maintain.

Modular

- Event driven @ kernel and optional threading facilities to individual processes.
- A lightweight event scheduler that dispatches events to running processes.
- Process execution is **triggered by events** dispatched by the kernel to the processes or by a polling mechanism.
- Any scheduled event will run to completion, however, event handlers can use internal mechanisms for preemption.
- Polling mechanism: high-priority events that are scheduled in between each asynchronous event. When a poll is scheduled, all processes that implement a poll handler are called in order of their priority.

Layered

Implements services in the form of layers.
++manageability, easy to understand, and reliability.
--not a very flexible architecture from an OS design perspective.

WSN Programming

MANTIS
MultimodAl NeTworks of In-situ Sensors

TinyOS

Contiki
The Open Source OS for the Internet of Things

Nano-RK: A Wireless Sensor Networking Real-Time Operating System

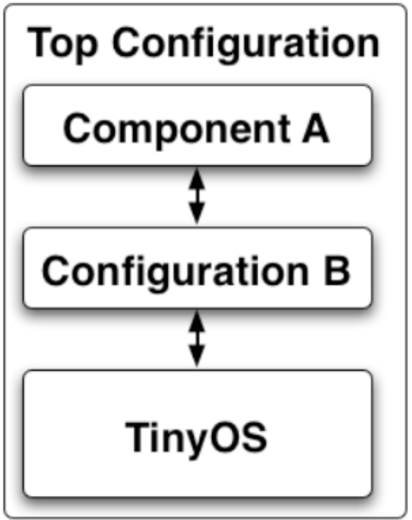

LiteOS: A Unix-like Operating System for Embedded Controllers and Sensor Networks

WSN Programming

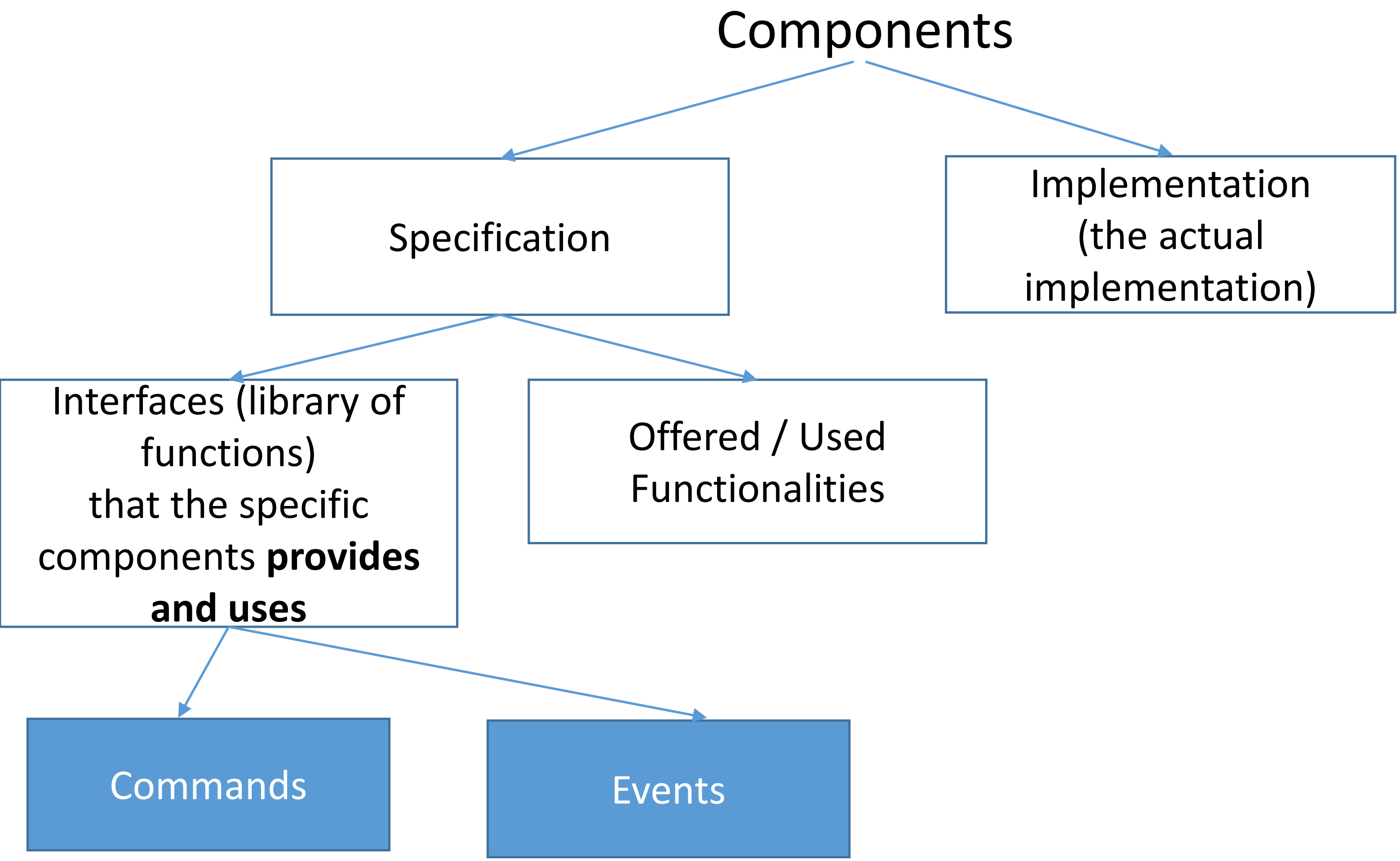
- Component-based architecture, implementing one single stack
- Event-based, non-blocking design that allows intra-mote concurrency
- Written in NesC
  - Structured, component-based C-like programming language

Programming Model:

- ***Components:*** encapsulate state and processing – use or provide interfaces
- ***Interfaces*** list commands and events
- ***Configurations*** wire components together for creating applications


Top Configuration
Component A
Configuration B
TinyOS

WSN Programming



Components

Specification

Implementation (the actual implementation)

Interfaces (library of functions) that the specific components **provides and uses**

Offered / Used Functionalities

Commands

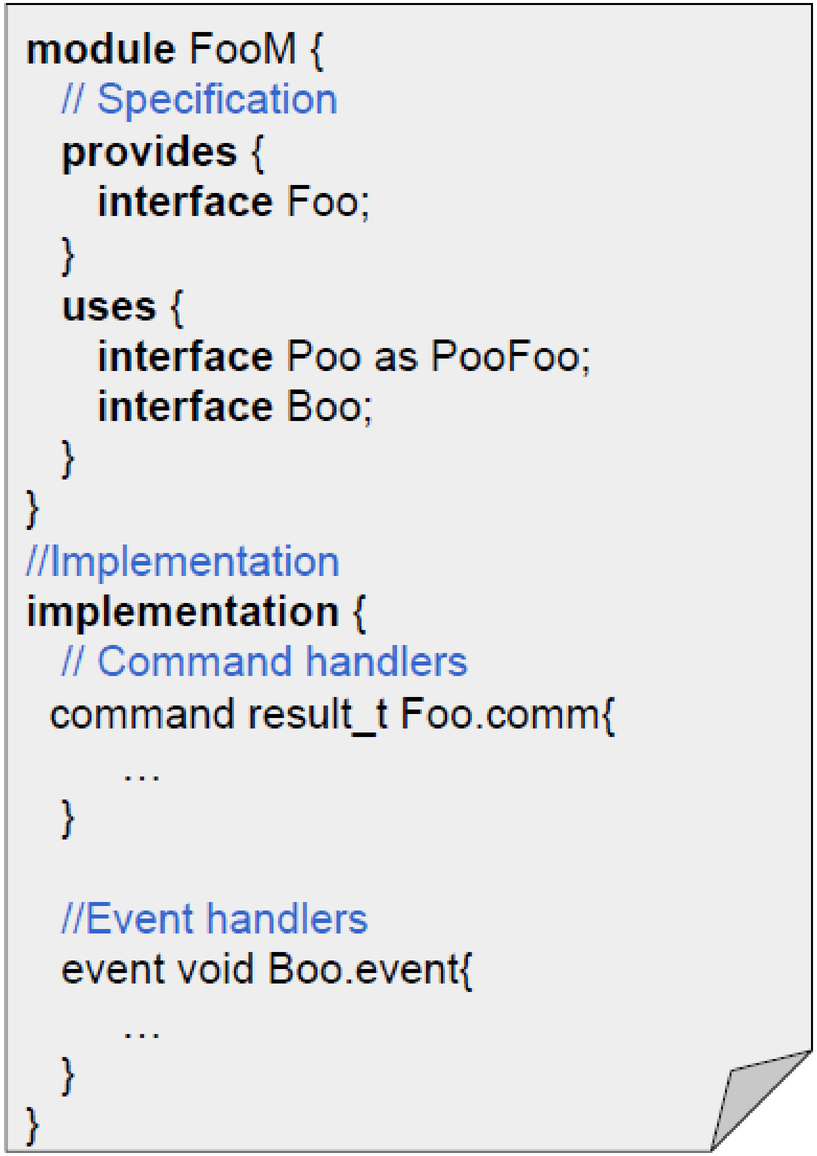
Events

Provider Component must define & implement
User component can use **as is**

Provider Component **must define** and **may** implement
User component must implement

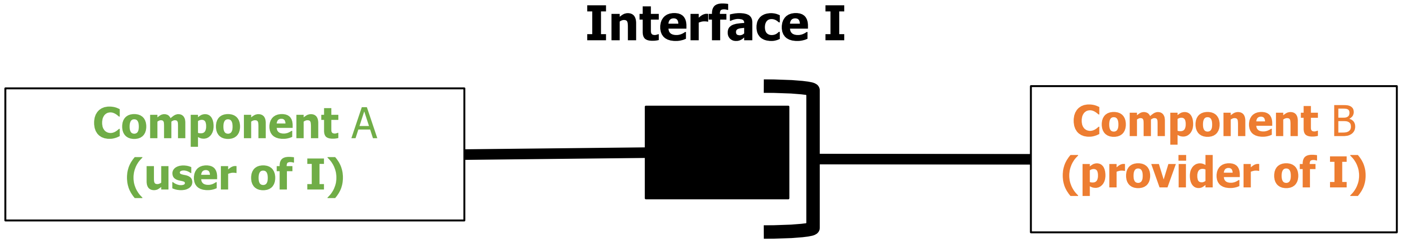
- Two kinds of components
  - Modules
    - specify and implement interfaces (**commands & events**)
    - A new set of library modules that can be used in a range of applications
  - Configurations (wiring)
    - connecting interfaces used by components to interfaces provided by others
    - build applications out of existing implementations.

Applications: define a top level configuration that wires together different components



```
module FooM {
   // Specification
   provides {
      interface Foo;
   }
   uses {
      interface Poo as PooFoo;
      interface Boo;
   }
}
//Implementation
implementation {
   // Command handlers
  command result_t Foo.comm{
        ...
   }

   //Event handlers
   event void Boo.event{
        ...
   }
}
```

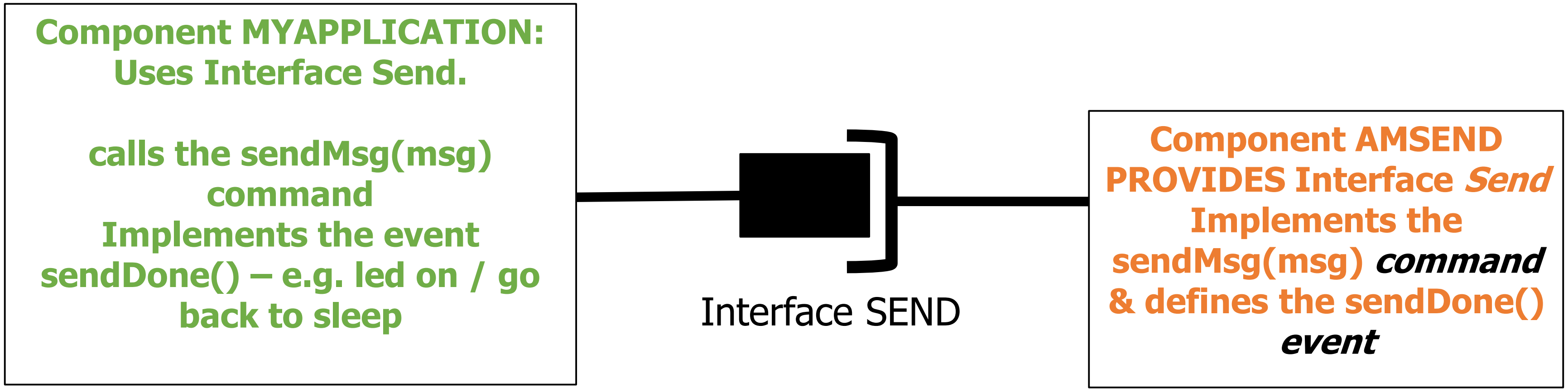
WSN Programming



Interface I

Component A
(user of I)

Component B
(provider of I)



Component MYAPPLICATION:
Uses Interface Send.

calls the sendMsg(msg) command
Implements the event sendDone() – e.g. led on / go back to sleep

Interface SEND

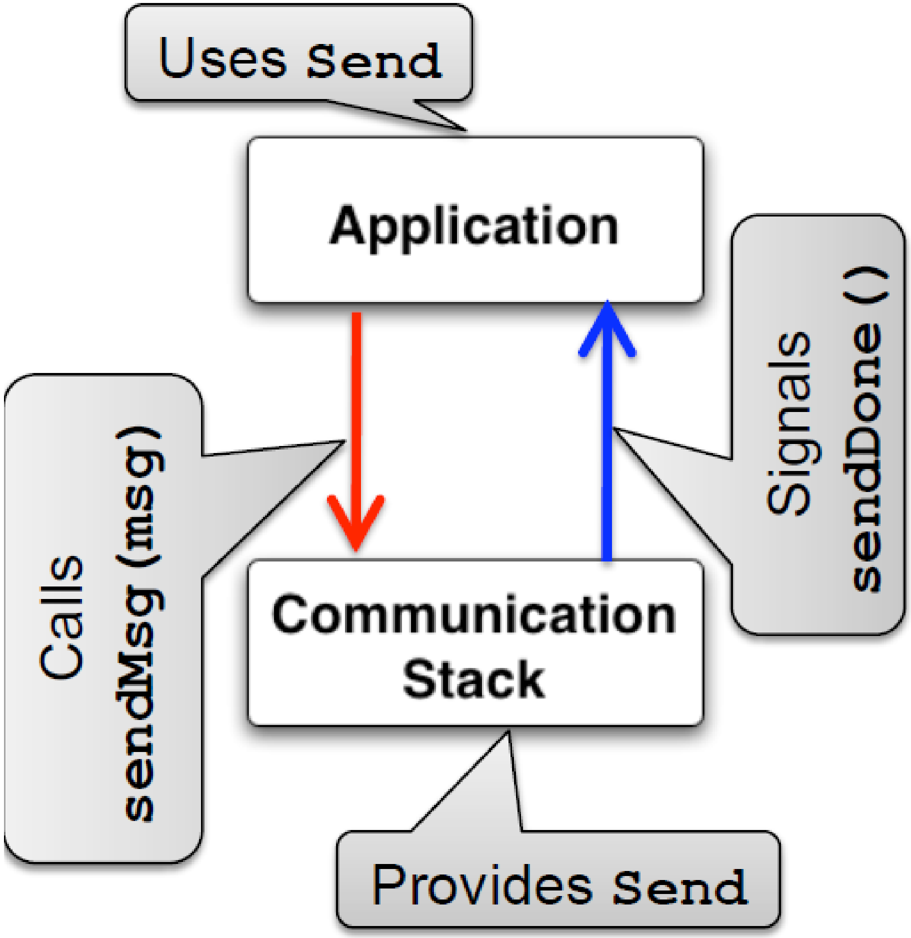
Component AMSEND PROVIDES Interface *Send*
Implements the sendMsg(msg) *command* & defines the sendDone() *event*

WSN Programming

Function calls	**Commands**	**Events**
Using	Call Command	Implement Event Handler
Providing	Implement Command Body	Signal Event

Split-phase execution
(Return values arrive asynchronously through events)

Computation mechanisms -> Tasks:
Typically spawned by events
By default non-preemptive but can be pre-empted by asynchronous events (usually kept small)
FIFO scheduling



WSN Programming

- A simple example:

Program a sensor node to blink a led every 250ms…..

A timer module

A led module

A module that combines them together.

```
configuration BlinkAppC
{
} //specification
implementation
{
  components MainC, BlinkC, LedsC;
  components new TimerMilliC() as Timer0;




  BlinkC -> MainC.Boot;

  BlinkC.Timer0 -> Timer0;


  BlinkC.Leds -> LedsC;
}
```

```
#include "Timer.h"
module BlinkC
{
  uses interface Timer<TMilli> as Timer0;


  uses interface Leds;
  uses interface Boot;
}
implementation
{
  event void Boot.booted()
  {
    call Timer0.startPeriodic( 250 );


  }

  event void Timer0.fired()
  {
    dbg("BlinkC", "Timer 0 fired

@ %s.\n", sim_time_string());
    call Leds.led0Toggle();
  }






}
```

[Split Face execution]

The provided component defines, the user interface must implement

WSN Programming

Contiki

The Open Source OS for the Internet of Things

Kon-Tiki (2012)

★ 7.2/10 32,818

PG-13 | 1h 58min | Adventure, History | 26 April 2013 (USA)

2:19 | Trailer

Legendary explorer Thor Heyerdal's epic 4,300-mile crossing of the Pacific on a balsawood raft in 1947, in an effort to prove that it was possible for South Americans to settle in Polynesia in pre-Columbian times.

Directors: Joachim Rønning, Espen Sandberg

Writers: Petter Skavlan, Allan Scott (script consultant) | 1 more credit »

Stars: Pål Sverre Hagen, Anders Baasmo Christiansen, Gustaf Skarsgård | See full cast & crew »

62 Metascore From metacritic.com | Reviews 59 user | 145 critic | Popularity 3,796 (↓ 87)

Emphasizes on the use of standard abstractions

Allows the shift from WSN to IoT:

- Baseline to upper, middleware services
- µIP stack & interoperability

WSN Programming

Contiki

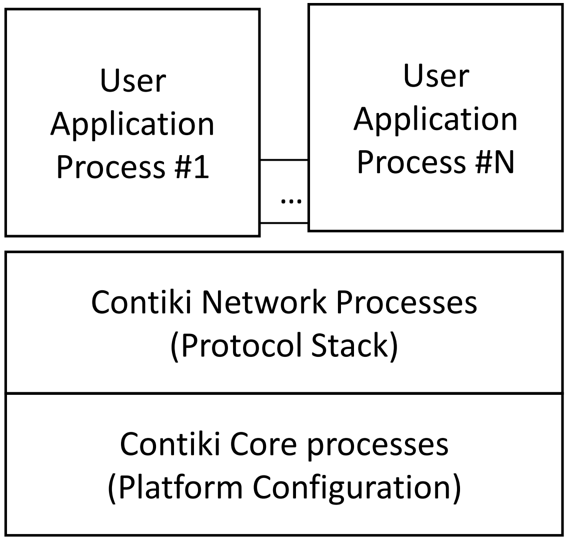
The Open Source OS for the Internet of Things

Event-based → Invoking processes

Using **protothreads**: a programming abstraction that combines events and threads

Single stack and sequential flow control

Posting events or polling



TinyOS: GOTO FLOW Vs
ContikiOS: Sequential flow control while keeping a single stack and an event handler for different threads

WSN Programming

Contiki

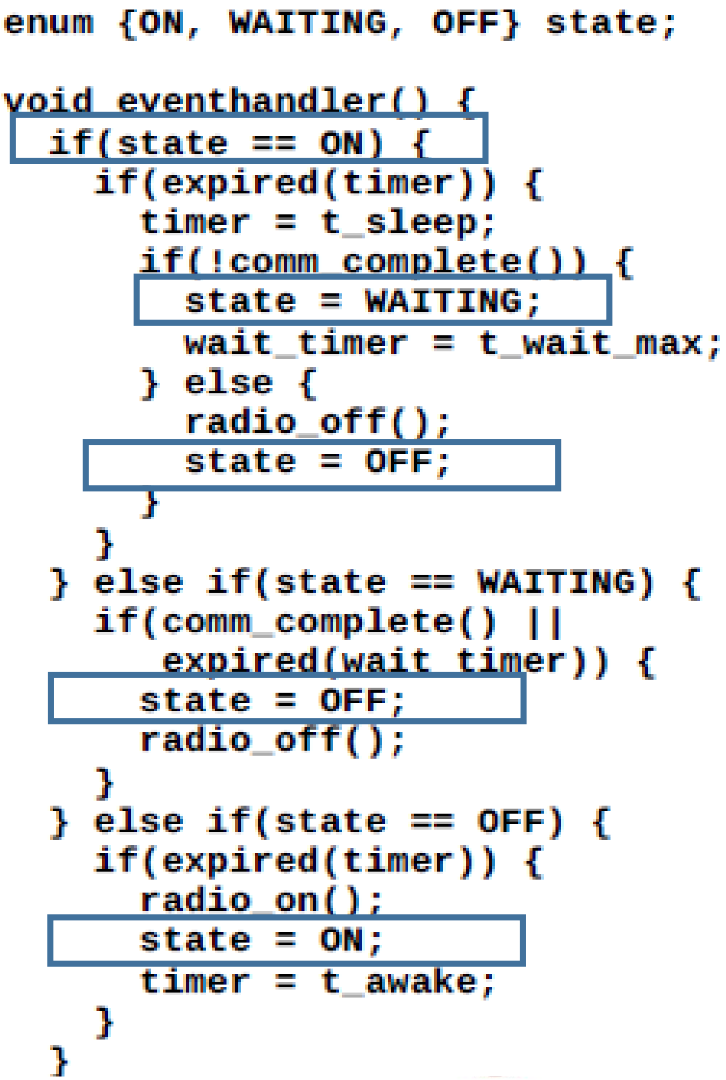
The Open Source OS for the Internet of Things

Protothreads:

- Lightweight, stackless threads
- Implement sequential flow of control without using complex state machines or full multi-threading.
- **Conditional blocking** inside a C function (on top of an event-driven system)
- To minimize the overhead of multi-threading – they run on the same stack (suitable for memory constrained systems, where a stack for a thread might use a large part of the available memory)
- **Protothread overhead: 2B of memory per protothread.**
- Scheduling -> based on repeated calls to the function: Each time the function is called, the protothread will run until it blocks or exits. Protothreads are scheduled by their applications

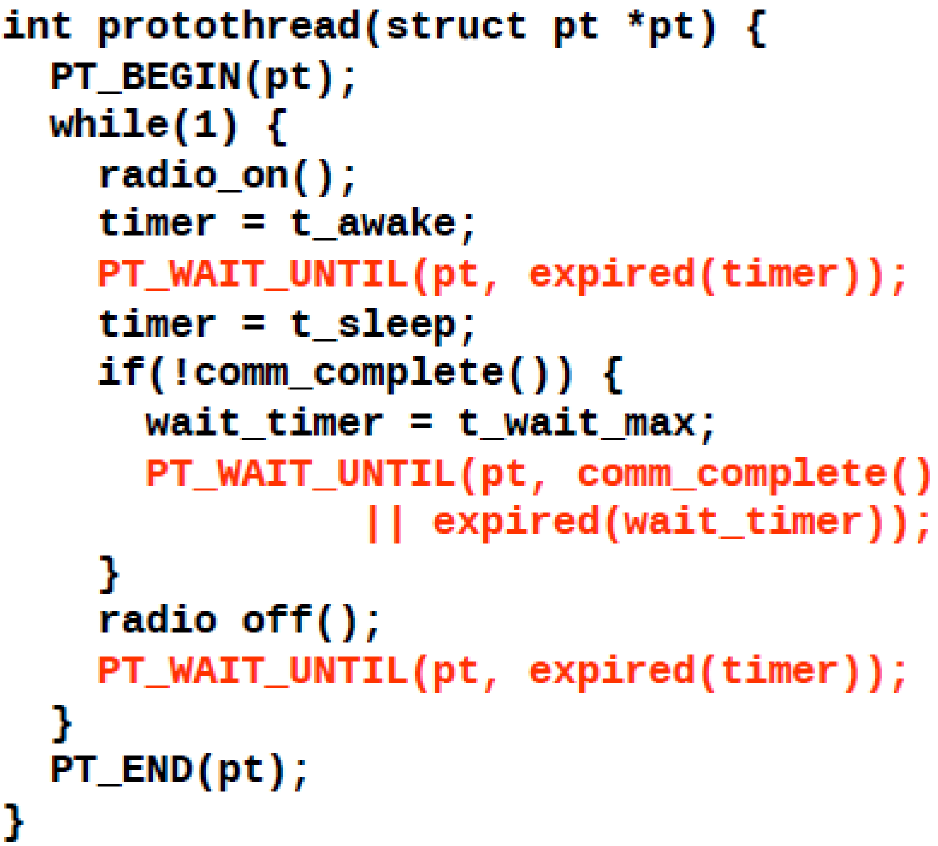
WSN Programming

Event-based state machine



```
enum {ON, WAITING, OFF} state;

void eventhandler() {
  if(state == ON) {
    if(expired(timer)) {
      timer = t_sleep;
      if(!comm_complete()) {
        state = WAITING;
        wait_timer = t_wait_max;
      } else {
        radio_off();
        state = OFF;
      }
    }
  } else if(state == WAITING) {
    if(comm_complete() ||
       expired(wait_timer)) {
      state = OFF;
      radio_off();
    }
  } else if(state == OFF) {
    if(expired(timer)) {
      radio_on();
      state = ON;
      timer = t_awake;
    }
  }
```

With Protothreads



```
int protothread(struct pt *pt) {
  PT_BEGIN(pt);
  while(1) {
    radio_on();
    timer = t_awake;
    PT_WAIT_UNTIL(pt, expired(timer));
    timer = t_sleep;
    if(!comm_complete()) {
      wait_timer = t_wait_max;
      PT_WAIT_UNTIL(pt, comm_complete()
                    || expired(wait_timer));
    }
    radio off();
    PT_WAIT_UNTIL(pt, expired(timer));
  }
  PT_END(pt);
}
```

WSN Programming

Contiki

The Open Source OS for the Internet of Things

An application implements several processes: each process is a protothread

#define **PROCESS_WAIT_EVENT()**
Wait for an event to be posted to the process.

#define **PROCESS_WAIT_EVENT_UNTIL(c)**
Wait for an event to be posted to the process, with an extra condition.

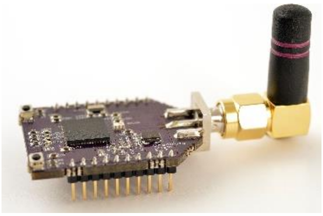
#define **PROCESS_YIELD()**
Yield the currently running process.

#define **PROCESS_YIELD_UNTIL(c)**
Yield the currently running process until a condition occurs.

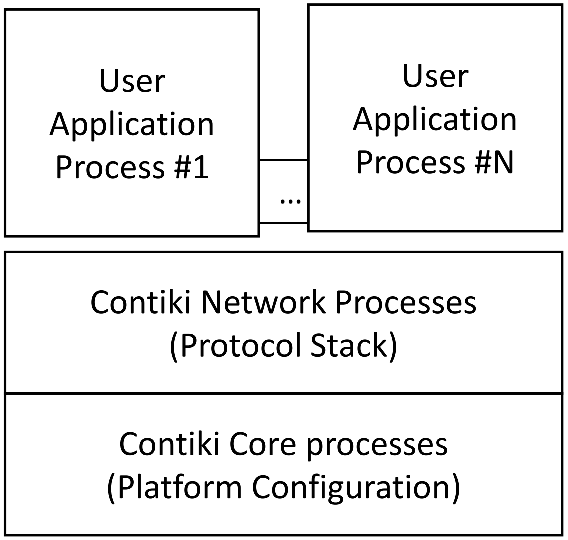
#define **PROCESS_WAIT_UNTIL(c)**
Wait for a condition to occur.

Contiki

The Open Source OS for the Internet of Things



One main.c for each platform:
Core & Network processes



```
process_init();
 process_start(&etimer_process, NULL);

 ctimer_init();

 init_platform();

 set_rime_addr();

//----------------------low level api to phy-------------------------
 cc2420_init();
 {
   uint8_t longaddr[8];
   uint16_t shortaddr;

   shortaddr = (linkaddr_node_addr.u8[0] << 8) +  linkaddr_node_addr.u8[1];
   memset(longaddr, 0, sizeof(longaddr));
   linkaddr_copy((linkaddr_t *)&longaddr, &linkaddr_node_addr);

   cc2420_set_pan_addr(IEEE802154_PANID, shortaddr, longaddr);
 }
 cc2420_set_channel(RF_CHANNEL);
memcpy(&uip_lladdr.addr, ds2411_id, sizeof(uip_lladdr.addr));

queuebuf_init();
 NETSTACK_RDC.init();
 NETSTACK_MAC.init();
```

WSN Programming

```
#include "contiki.h"
#include "dev/leds.h"

#include <stdio.h> /* For printf() */
/
*-----------------------------------------------------------------
        */
                      */
PROCESS(blink_process, "LED blink process");

/* We require the processes to be started automatically */

/AUTOSTART_PROCESSES(&blink_process);
*-----------------------------------------------------------------
--------*/- /* Implementation of the process */

PROCESS_THREAD(blink_process, ev, data)
{   static struct etimer timer;

    PROCESS_BEGIN();

    while (1)
    {   /* we set the timer from here every time */
        etimer_set(&timer, CLOCK_CONF_SECOND);
       /* and wait until the event we receive is the one we're
       waiting for */
       PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER);
        printf("Blink... (state %0.2X).\r\n", leds_get());

        leds_toggle(LEDS_GREEN); /* update the LEDs */
    }
    PROCESS_END();
}
/
```

WSN Programming

Contiki

The Open Source OS for the Internet of Things

The communication layers in Contiki

- The Rime protocol stack
  - A set of communication primitives (keeping pck headers and protocol stacks separated)
  - A pool of NWK protocols for ad-hoc networking
  - Best-effort anonymous broadcast to reliable multihop flooding and tree protocols

- The uIP TCP/IP stack
  - Lightweight TCP/IP functionalities for low complexity µControllers
  - A single network interface (IP, ICMP, UDP,TCP)
  - Compliant to RFC but the Application layer is responsible for handling retransmissions (reduce memory requirements)
  - **RPL is part of it**

WSN Programming

Contiki

The Open Source OS for the Internet of Things

How does Rime work

- Rime is a software trick
- A stack of NWK layers (e.g. broadcast, unicast, polite, etc)
- Each layer represents another type of traffic & adds something to the network header

- Complex network protocol are decomposed to simpler ones
- Each layer is associated with a channel

- 2KB memory footprint

- Interoperability and ease in changing the protocol stack

WSN Programming

How does Rime work – Example

- The Collection Tree Protocol (CTP)
  - Tree-based hop-by-hop reliable data collection
  - Large-scale network (e.g. environmental or industrial monitoring)

- Reliable Unicast Bulk
  - Event-driven data transmission of a large data volume
  - Personal health-care

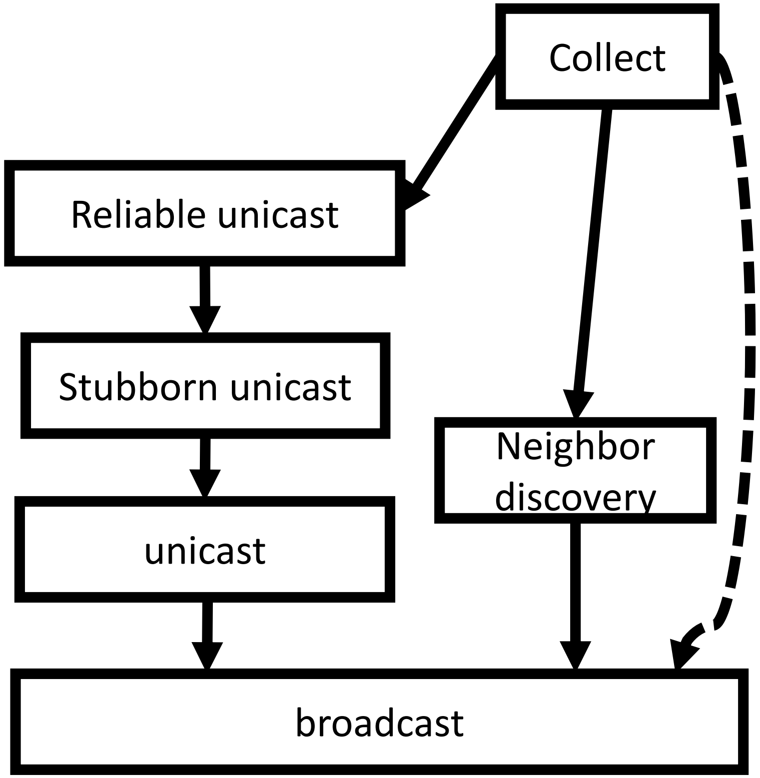
WSN Programming

How does Rime work – Example

- The Collection Tree Protocol (CTP)
  - Tree-based hop-by-hop reliable data collection

Large-scale network (e.g. environmental or industrial monitoring)

Layer	**Description**	**Channel**	**Contribution to Rime Header**
Broadcast	Best-effort local area broadcast	129	Sender ID
Neighbor discovery	Periodic Neighbor Discovery mechanism	2	Receiver ID, Application Channel
Unicast	Single-hop unicast to an identified single-hop neighbor	146	Receiver ID
Stubborn unicast	Repeatedly sends a packet until cancelled by upper layer		Receiver ID
Reliable Unicast	Single-hop reliable unicast (ACKs and retransmissions)	144	Packet Type and Packet ID



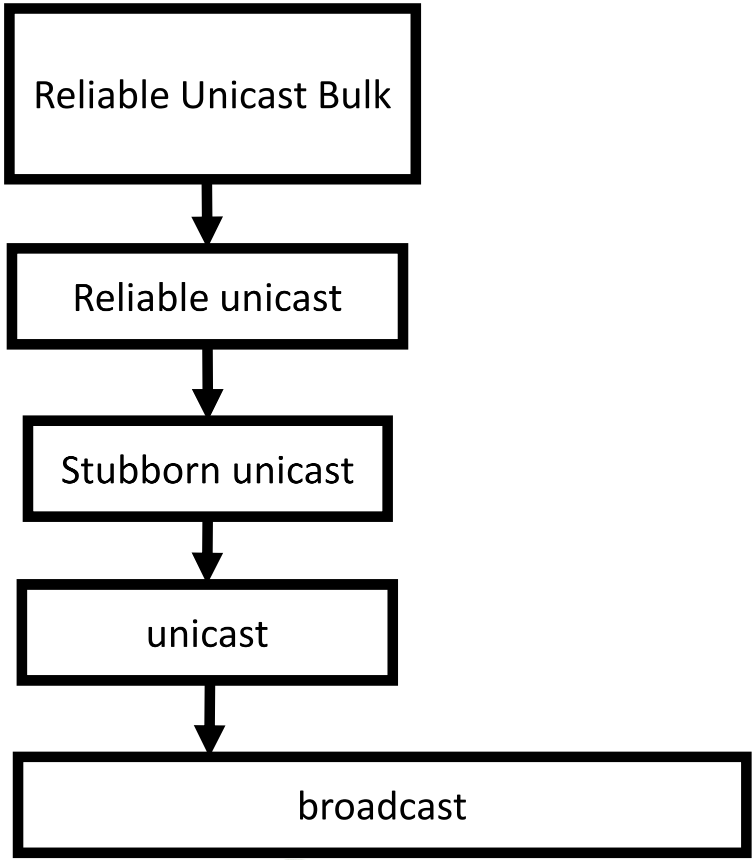
WSN Programming

Reliable Unicast Bulk

Event-driven data transmission of a large data volume

Personal health-care

Layer	**Description**	**Channel**	**Contribution to Rime Header**
Broadcast	Best-effort local area broadcast	129	Sender ID
Unicast	Single-hop unicast to an identified single-hop neighbor	146	Receiver ID
Stubborn unicast	Repeatedly sends a packet until cancelled by upper layer		Receiver ID
Reliable Unicast	Single-hop reliable unicast (ACKs and retransmissions)	144	Packet Type and Packet ID


Reliable Unicast Bulk

Reliable unicast

Stubborn unicast

unicast

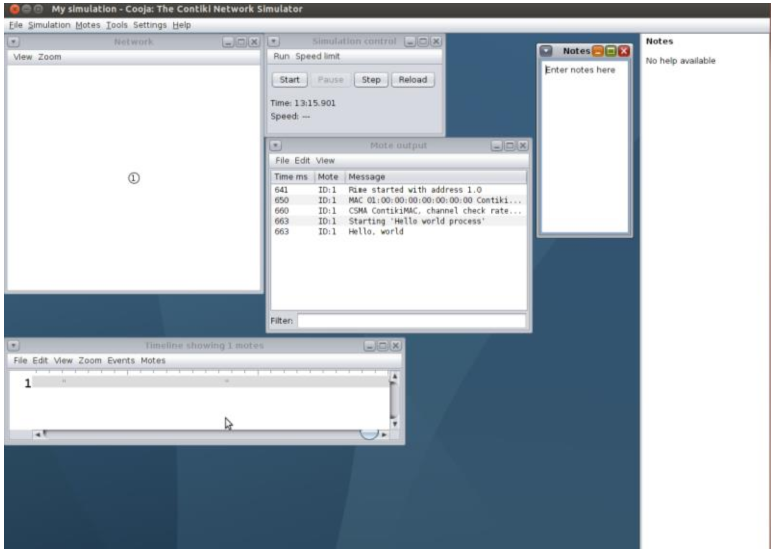
broadcast

WSN Programming

Contiki

The Open Source OS for the Internet of Things

- Cooja
  - The Contiki emulator for running WSN applications.
  - Very useful for debugging your codes – the same code you test on cooja, the same you upload to your mote
  - Evaluating the network performance – has very simplifying models for radio propagation….
    - Unit disk model: Edges are instantly configured according to power attenuation w.r.t to distance & success ratio (configurable)
    - Directed graph radio medium: Considers preconfigured edges, without checking the output power.
    - Multipath ray tracer: Simulates reflection and diffraction through homogeneous obstacles (considers that all nodes have the same transmission power)
  - Interacts with external tools, e.g. Wireshark for Network monitoring
  - Modular: Plugins for extending functionality



Time: 13:15.901

Speed: ---

```
Time ms  Mote  Message
641      ID:1  Rime started with address 1.0
650      ID:1  MAC 01:00:00:00:00:00:00:00 Contiki...
660      ID:1  CSMA ContikiMAC, channel check rate...
663      ID:1  Starting 'Hello world process'
663      ID:1  Hello, world
```

WSN Programming

	TinyOS	Contiki The Open Source OS for the Internet of Things
First Release	**1999**	**2005**
Supported Platforms (in official distributions)	**17**	**26**
Community Support & Forums	**Yes**	**Yes**
Programming Language	**nesC**	**C**
Single / Multiple Thread	**Single (multithread is optional)**	**Even-driven kernel with preemptive multithreading**
Structure	**Component-based**	**Protothreads (stack-less and lightweight)**
Simulator / Emulator	**TOSSIM (python)**	**Cooja / MSPSIm Emulator (java)**
OTAP	**Yes**	**Yes**
Protocol Stack	**(802.15.4) MAC (not fully supported)** **LPL/ Collection Tree** **RPL/6LoWPAN**	**(802.15.4) MAC (not fully supported)** **Radio Duty Cycle & MAC** **RIME / uIP** **6LoWPAN**
	Great flexibility in generating highly customizable protocol stack	**With default distribution: RIME or 6LoWPAN (modifiable)**
Interfacing with host (Serial Communication)	**Specific format (ActiveMessageC)**	**Flexible** **(but provides tools s.a. SLIP)**
Documentation & Support	**Provides both**	**Provides both & also visualization tools**

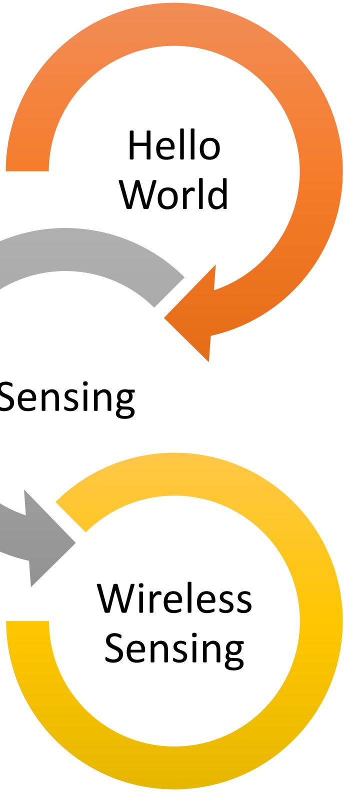
HANDS ON SESSIONS

Hands on Session

Contiki

The Open Source OS for the Internet of Things

What we are going to do…



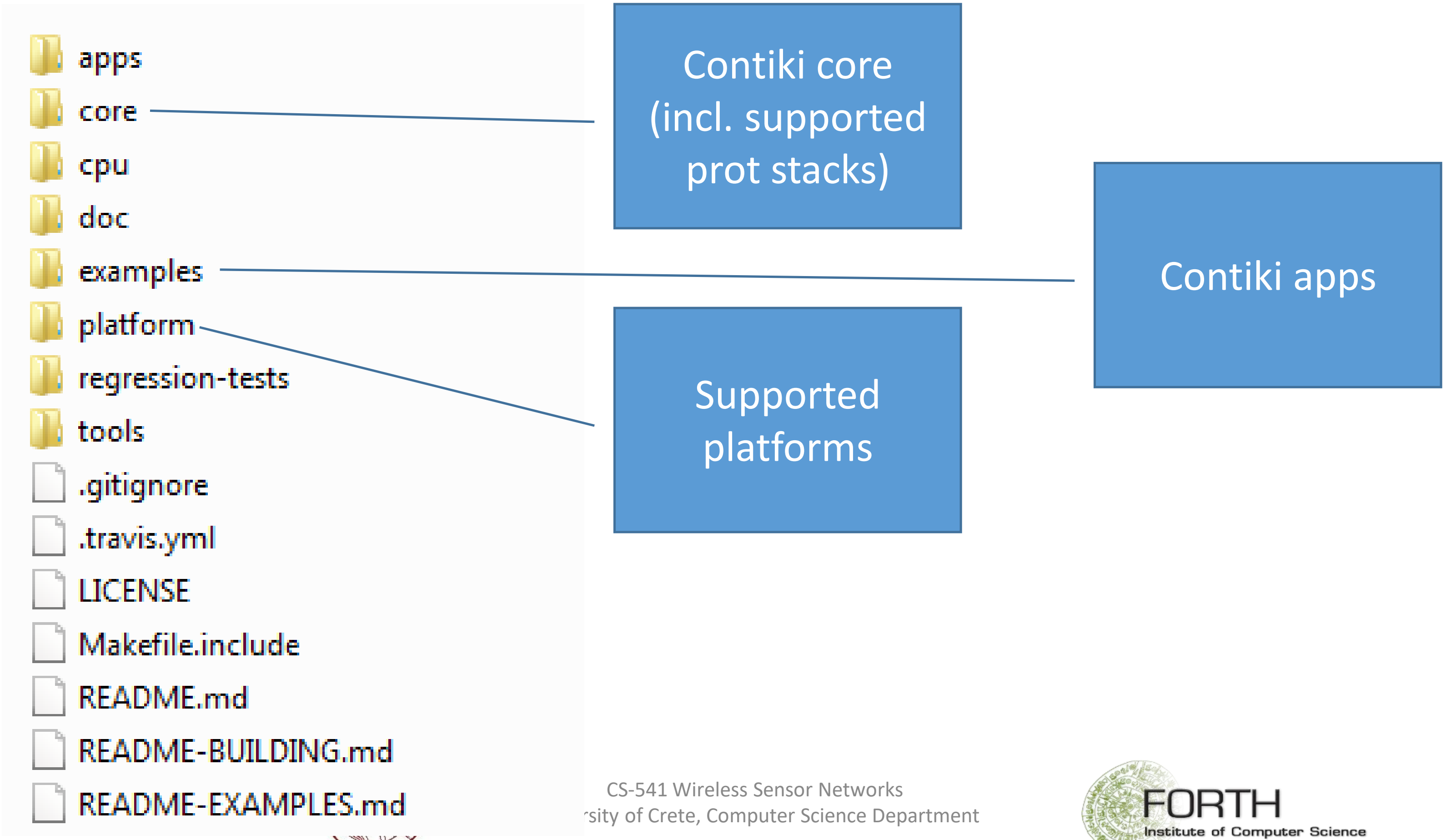
Hello World

Sensing

Wireless Sensing

Hands on Session

Your contiki directory structure:



- apps
- core
- cpu
- doc
- examples
- platform
- regression-tests
- tools
- .gitignore
- .travis.yml
- LICENSE
- Makefile.include
- README.md
- README-BUILDING.md
- README-EXAMPLES.md

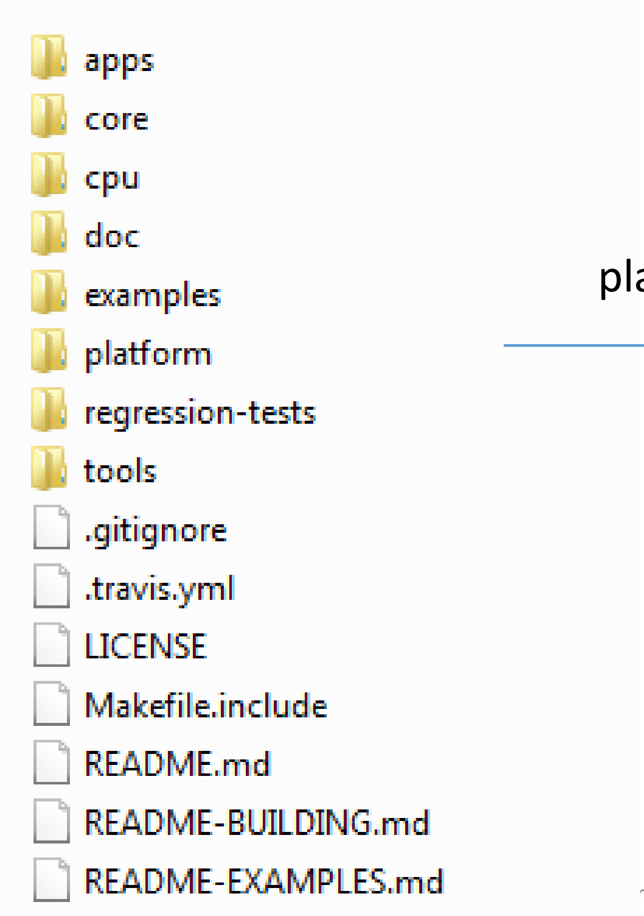
Contiki core (incl. supported prot stacks)

Contiki apps

Supported platforms

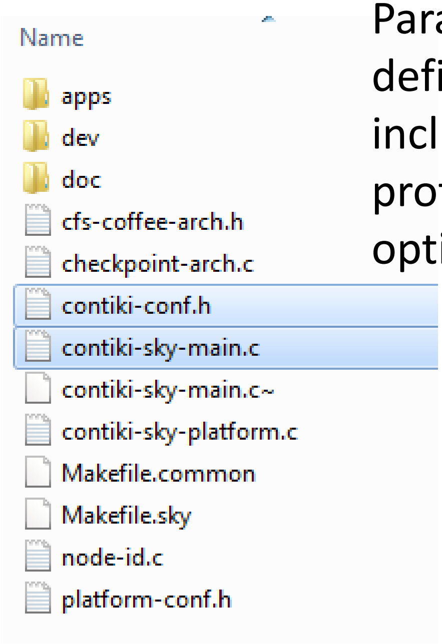
Hands on Session

Your contiki directory structure:



apps
core
cpu
doc
examples
platform
regression-tests
tools
.gitignore
.travis.yml
LICENSE
Makefile.include
README.md
README-BUILDING.md
README-EXAMPLES.md

platforms/sky



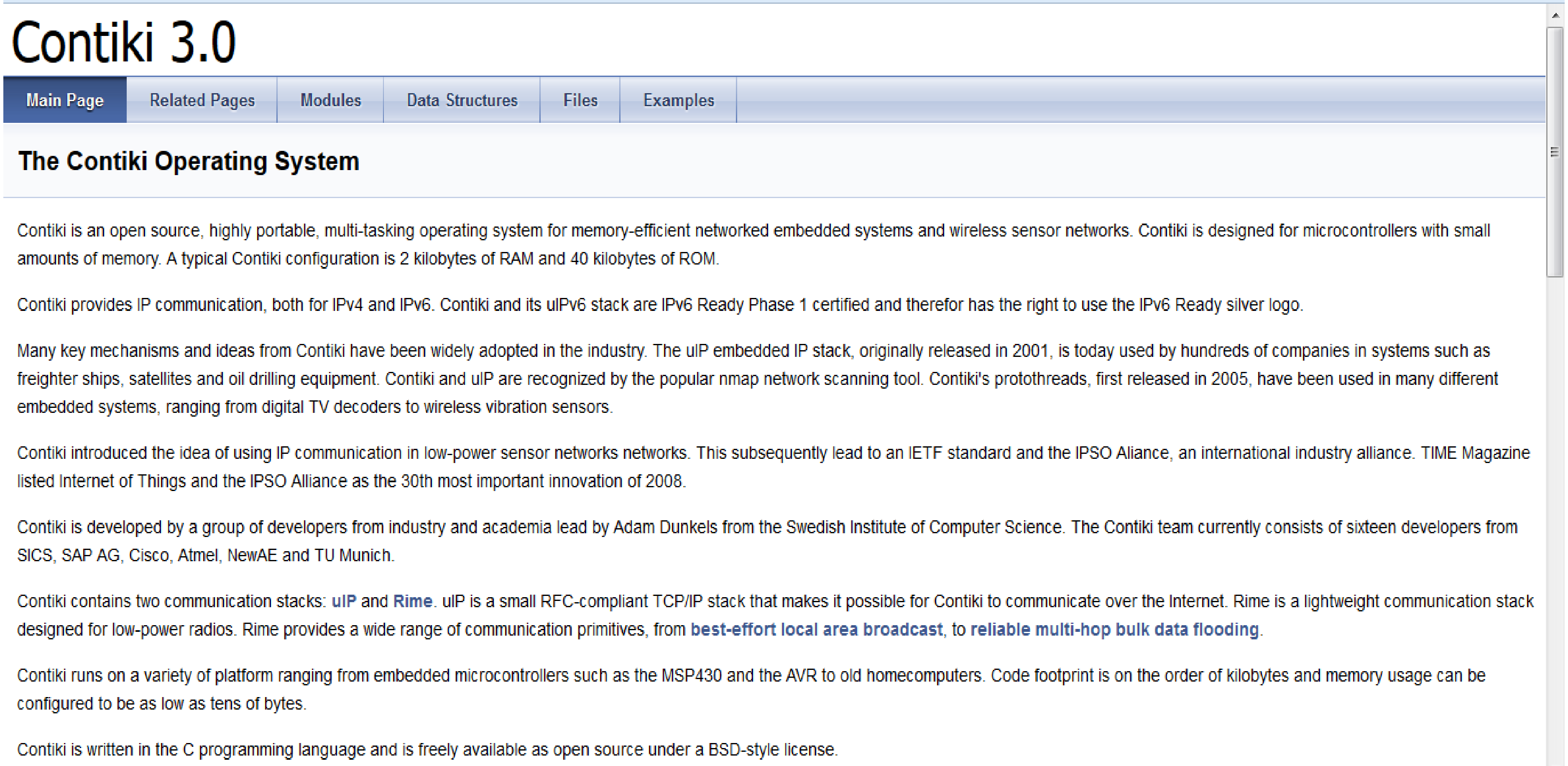
Name
apps
dev
doc
cfs-coffee-arch.h
checkpoint-arch.c
contiki-conf.h
contiki-sky-main.c
contiki-sky-main.c~
contiki-sky-platform.c
Makefile.common
Makefile.sky
node-id.c
platform-conf.h

Parameters definition incl. protocol options

Hands on Session

http://www.hopnetwork.com/contiki/index.html

Contiki
The Open Source OS for the Internet of Things



Contiki 3.0

The Contiki Operating System

Contiki is an open source, highly portable, multi-tasking operating system for memory-efficient networked embedded systems and wireless sensor networks. Contiki is designed for microcontrollers with small amounts of memory. A typical Contiki configuration is 2 kilobytes of RAM and 40 kilobytes of ROM.

Contiki provides IP communication, both for IPv4 and IPv6. Contiki and its uIPv6 stack are IPv6 Ready Phase 1 certified and therefor has the right to use the IPv6 Ready silver logo.

Many key mechanisms and ideas from Contiki have been widely adopted in the industry. The uIP embedded IP stack, originally released in 2001, is today used by hundreds of companies in systems such as freighter ships, satellites and oil drilling equipment. Contiki and uIP are recognized by the popular nmap network scanning tool. Contiki's protothreads, first released in 2005, have been used in many different embedded systems, ranging from digital TV decoders to wireless vibration sensors.

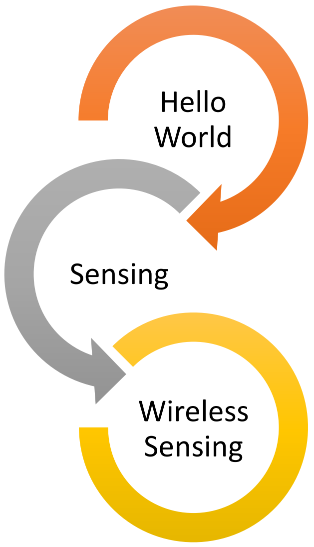
Contiki introduced the idea of using IP communication in low-power sensor networks networks. This subsequently lead to an IETF standard and the IPSO Aliance, an international industry alliance. TIME Magazine listed Internet of Things and the IPSO Alliance as the 30th most important innovation of 2008.

Contiki is developed by a group of developers from industry and academia lead by Adam Dunkels from the Swedish Institute of Computer Science. The Contiki team currently consists of sixteen developers from SICS, SAP AG, Cisco, Atmel, NewAE and TU Munich.

Contiki contains two communication stacks: **uIP** and **Rime**. uIP is a small RFC-compliant TCP/IP stack that makes it possible for Contiki to communicate over the Internet. Rime is a lightweight communication stack designed for low-power radios. Rime provides a wide range of communication primitives, from **best-effort local area broadcast**, to **reliable multi-hop bulk data flooding**.

Contiki runs on a variety of platform ranging from embedded microcontrollers such as the MSP430 and the AVR to old homecomputers. Code footprint is on the order of kilobytes and memory usage can be configured to be as low as tens of bytes.

Contiki is written in the C programming language and is freely available as open source under a BSD-style license.



Hands on Session

Hello World ☺ contiki/examples/hello-world

[Code structure & compile]

Hello-world.c

```
#include "contiki.h"

#include <stdio.h> /* For printf() */
/*---------------------------------------------------------------------------*/
PROCESS(hello_world_process, "Hello world process");   /**Process definition**/
AUTOSTART_PROCESSES(&hello_world_process);      /**Process Start**/
/*---------------------------------------------------------------------------*/
PROCESS_THREAD(hello_world_process, ev, data) /**Process implementation**/
{
  PROCESS_BEGIN(); /**Always first**/

  printf("Hello, world\n"); //process core

  PROCESS_END();  /**Always last**/
}
/*---------------------------------------------------------------------------*/
```

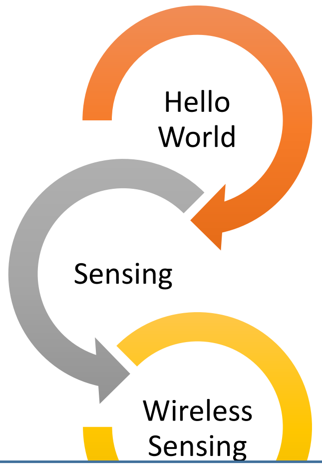
Hands on Session

Hello World ☺ contiki/examples/hello-world

[Code structure & compile]

```
CONTIKI = ../..

#TARGET_LIBFILES += -lm
#CFLAGS += -DPROJECT_CONF_H=\"project-conf.h\"
#PROJECT_SOURCEFILES += project1.c
#CONTIKI_SOURCEFILES +=mac1.c rdc1.c
#UIP_CONF_IPV6=1  ##macros...


include $(CONTIKI)/Makefile.include
```

Include headers and files.

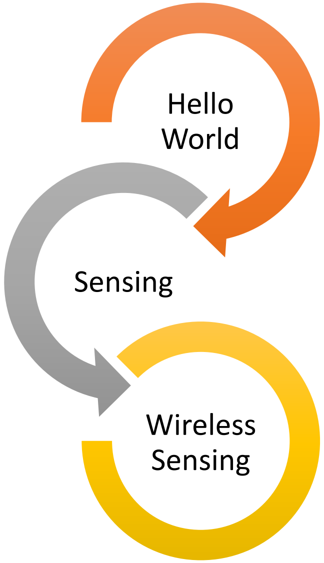


Makefile

Customized configuration of protocol stack

APP layer / AUTOSTARTED with your main app process

Source codes to use (usually containing customized network solutions) – that contiki processes will use.


Hello World

Sensing

Wireless Sensing

Hands on Session

Hello World ☺ contiki/examples/hello-world
[Code structure & compile]

Program:

1. Open command terminal.
2. cd contiki/examples/hello-world
3. **make TARGET=<platform*> hello-world.upload** (compile and program)

Serial Dump

1. At new tab (File/Open new tab).
2. make TARGET=sky MOTES=**/dev/ttyUSB0** login

*sky/xm1000/z1

Hands on Session

What we use…

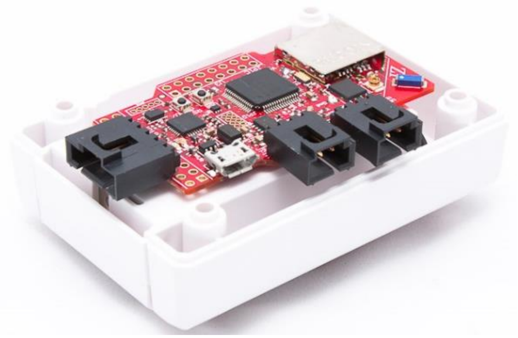
Product Name	Extras	Notes:
Z1	5dBi dipole antenna	**Temperature & accelerometer - ports to host more sensors**
[redacted]	[redacted]	[redacted]
CM5000-SMA	5dBi dipole antenna	**Temperature, humidity & light sensor** **Network compatible to Z1**

WSN Programming

Hello-world in WSN programming.

A Blinking-Led Application

- Program a mote to blink a led every T seconds.

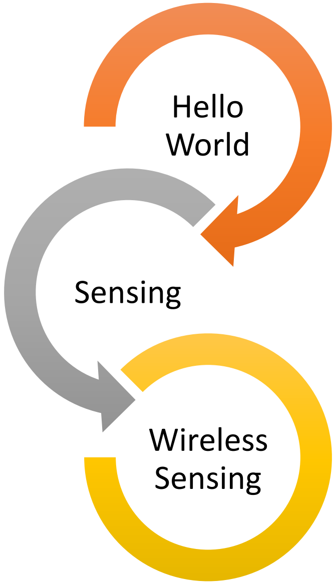


Hands on Session

Hello World ☺ contiki/examples/hello-world

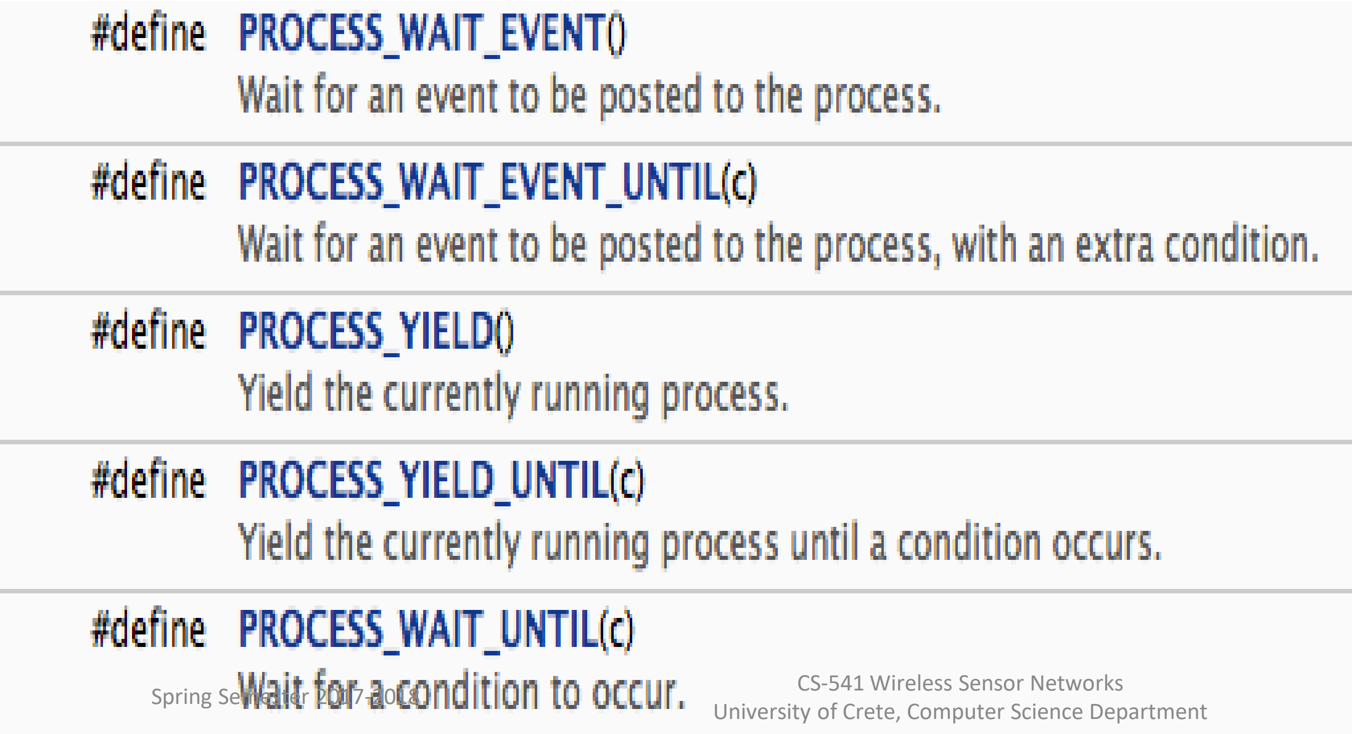
[How to trigger a process]

- How to wake up from a process


Hello World

Sensing

Wireless Sensing


```
#define PROCESS_WAIT_EVENT()
    Wait for an event to be posted to the process.

#define PROCESS_WAIT_EVENT_UNTIL(c)
    Wait for an event to be posted to the process, with an extra condition.

#define PROCESS_YIELD()
    Yield the currently running process.

#define PROCESS_YIELD_UNTIL(c)
    Yield the currently running process until a condition occurs.

#define PROCESS_WAIT_UNTIL(c)
    Wait for a condition to occur.
```

Keep on mind that:

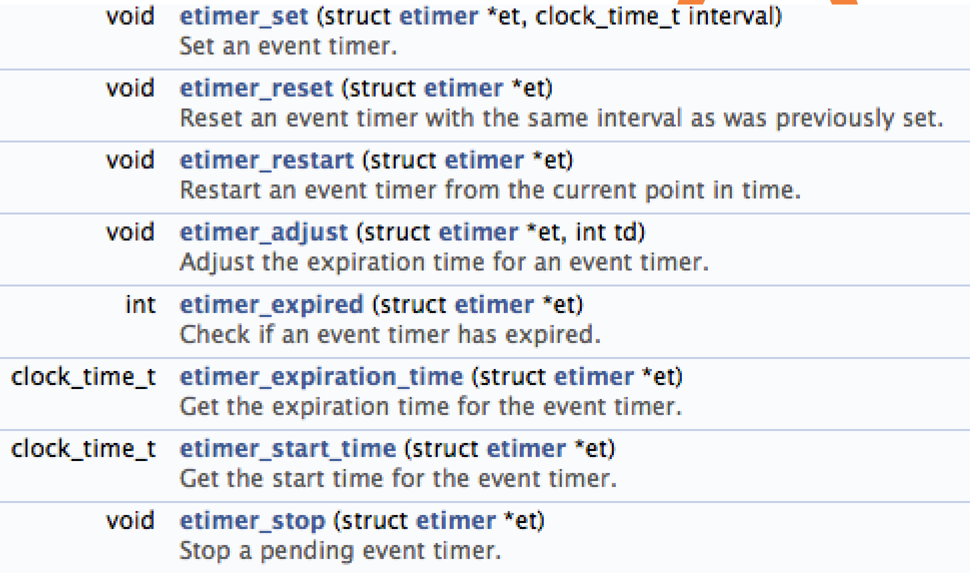
Automatic variables not stored across a blocking wait

When in doubt, use static local variables

Hands on Session

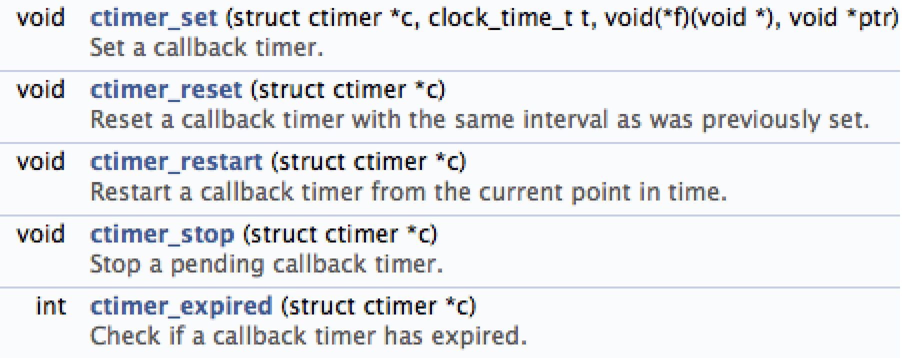
[How to trigger a process]

- Timers
  - Event timer (etimer) : Sends an event when expired



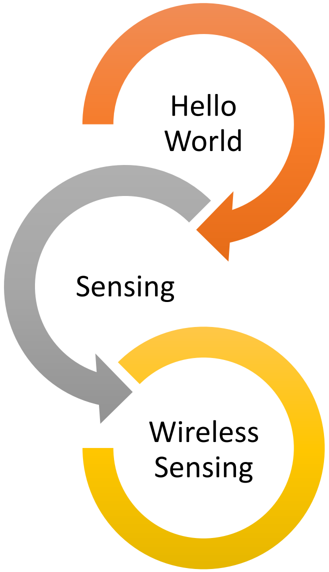
void	**etimer_set** (struct **etimer** *et, clock_time_t interval) Set an event timer.
void	**etimer_reset** (struct **etimer** *et) Reset an event timer with the same interval as was previously set.
void	**etimer_restart** (struct **etimer** *et) Restart an event timer from the current point in time.
void	**etimer_adjust** (struct **etimer** *et, int td) Adjust the expiration time for an event timer.
int	**etimer_expired** (struct **etimer** *et) Check if an event timer has expired.
clock_time_t	**etimer_expiration_time** (struct **etimer** *et) Get the expiration time for the event timer.
clock_time_t	**etimer_start_time** (struct **etimer** *et) Get the start time for the event timer.
void	**etimer_stop** (struct **etimer** *et) Stop a pending event timer.

  - Callback timer (ctimer) : Calls a function when expired – used by Rime



void	**ctimer_set** (struct ctimer *c, clock_time_t t, void(*f)(void *), void *ptr) Set a callback timer.
void	**ctimer_reset** (struct ctimer *c) Reset a callback timer with the same interval as was previously set.
void	**ctimer_restart** (struct ctimer *c) Restart a callback timer from the current point in time.
void	**ctimer_stop** (struct ctimer *c) Stop a pending callback timer.
int	**ctimer_expired** (struct ctimer *c) Check if a callback timer has expired.

Hands on Session



Hello World ☺

[How to trigger a process]

From hello-world.c generate a new application (print-and-blink.c) that:

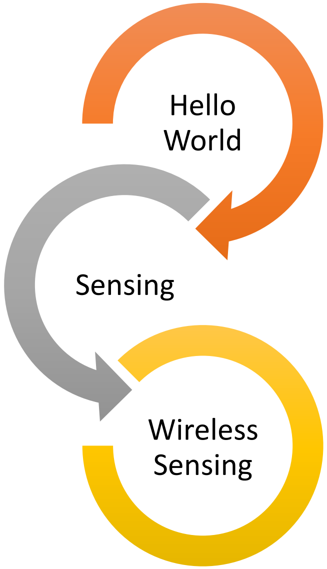
1. periodically (e.g. per second) prints a message.
2. when the message is printed a led toggles

   #include “leds.h”

   leds_toggle(LEDS_RED / LEDS_GREEN / LEDS_YELLOW)

   macro for time: **CLOCK_SECOND**

```
#include “leds.h”

/*------------------------------------------------------------------*/
PROCESS(print_and_blink_process, ”Print and blink process");
AUTOSTART_PROCESSES(&print_and_blink_process);
/*------------------------------------------------------------------*/
PROCESS_THREAD(print_and_blink_process, ev, data)
{
static struct etimer et;

PROCESS_BEGIN(); /**Always first**/

while(1) {

etimer_set(&et, 5*CLOCK_SECOND);

PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));

printf(“Echo\n”);

leds_toggle(LEDS_GREEN);

}
  PROCESS_END();  /**Always last**/
}
```

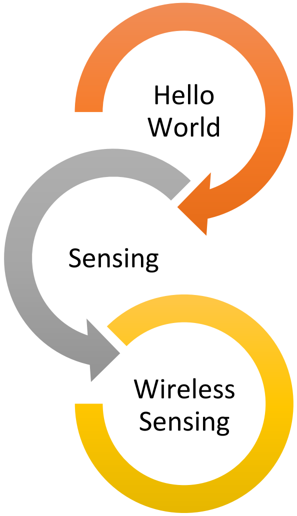


Hello World

Sensing

Wireless Sensing

WSN Programming

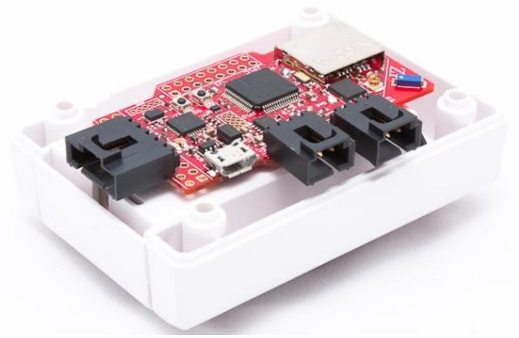

Hello World

Sensing

Wireless Sensing

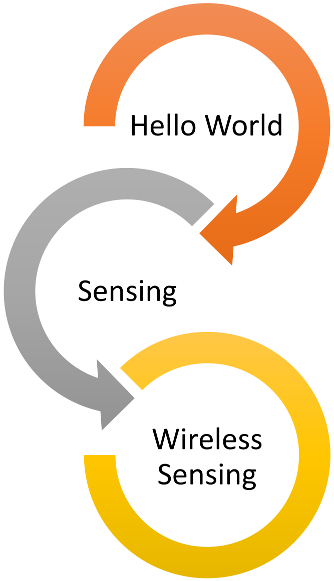
A Sense and Blink Application

- Program a mote to read its sensors every T seconds, print the values and blink a led



Temperature & battery

Temperature, humidity, radiation & battery



SHT11 -> Temperature and Humidity

HOW TO ACCESS IT:

A. READ THE VALUES of the global struct **sht11_sensor.value(type) (PROVIDED BY PLATFORM API)**

type = SHT11_SENSOR_TEMP, SHT11_SENSOR_HUMIDITY

B. USE THE API PROVIDED BY OS API

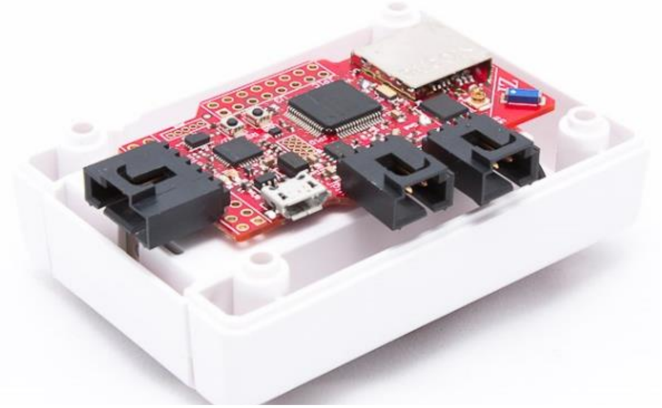
```
void sht11_init(void);
void sht11_off(void);

unsigned int sht11_temp(void);
unsigned int sht11_humidity(void);
unsigned int sht11_sreg(void);
int      sht11_set_sreg(unsigned);
```

HAMAMATSU-> RADIATION (TOTAL SOLAR & PHOTOSYNTHETICALLY AVAILABLE)

HOW TO ACCESS IT:

A. READ THE VALUES of the global struct **light_sensor.value(type) (PROVIDED BY PLATFORM API)**

type = LIGHT_SENSOR_TOTAL_SOLAR, LIGHT_SENSOR_PHOTOSYNTHETIC



BATTERY SENSOR -> READ THE INPUT VOLTAGE

HOW TO ACCESS IT:

READ THE VALUES of the global struct **battery_sensor.value(type) (PROVIDED BY PLATFORM API)**

type = 0

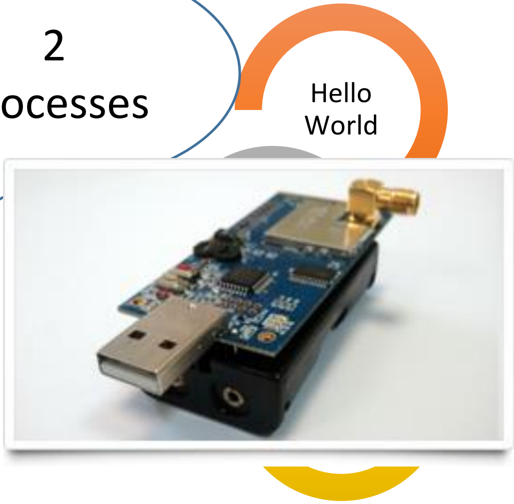
#include "dev/light-sensor.h” / "dev/sht11/sht11-sensor.h” / "dev/battery-sensor.h”

```
PROCESS_THREAD(sense_and_blink_process, ev, data)
{
                static struct etimer et;
                static struct sensor_datamsg msg;

                PROCESS_BEGIN(); /**Always first**/
//activate the sensors
//SENSORS_ACTIVATE(sht11_sensor);
SENSORS_ACTIVATE(battery_sensor);
SENSORS_ACTIVATE(light_sensor);
sht11_init();

while (1) {

                etimer_set(&et, CLOCK_SECOND);
                PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
//get the data
                msg.temp= sht11_temp();
                msg.humm = sht11_humidity();
                msg.light1 = light_sensor.value(LIGHT_SENSOR_PHOTOSYNTHETIC);
  msg.light2 = light_sensor.value(LIGHT_SENSOR_TOTAL_SOLAR);
                msg.batt = battery_sensor.value(0);


                printf("Sensor raw values: temperature:%d, humidity: %d, battery: %d, visible light: %d\n, total solar radiation: %d\n",
msg.temp, msg.humm, msg.batt,msg.light1, msg.light2);
                leds_toggle(LEDS_GREEN);
}
//deactivate
                sht11_off();
                SENSORS_DEACTIVATE(light_sensor);
                SENSORS_DEACTIVATE(battery_sensor);
                PROCESS_END(); /**Always last**/
}
```

```
//the data structure
struct sensor_datamsg{

                uint16_t temp;
                uint16_t humm;
                uint16_t batt;
                uint16_t light1;
                uint16_t light2;
}sensor_datamsg;
```

```
PROCESS(sense_process, "Sense process");
PROCESS(print_and_blink_process, "Print and blink process");
AUTOSTART_PROCESSES(&sense_process, &print_and_blink_process);
static struct sensor_datamsg msg;
static process_event_t event_data_ready;
```

2
processes



1

```
PROCESS_THREAD(sense_process, ev, data)
{
static struct etimer et;

PROCESS_BEGIN(); /**Always first**/
//activate the sensors
…
while (1) {

//read values as previously

//and now it is time to wake up the 2nd process
process_post(&print_and_blink_process,event_data_ready, &msg);
}
//deactivate
…

PROCESS_END(); /**Always last**/
}
```

2

```
PROCESS_THREAD(print_and_blink_process, ev, data)
{
PROCESS_BEGIN(); /**Always first**/

while (1) {
PROCESS_YIELD_UNTIL(ev==event_data_ready);

//and then print
}
PROCESS_END(); /**Always last**/
}
```

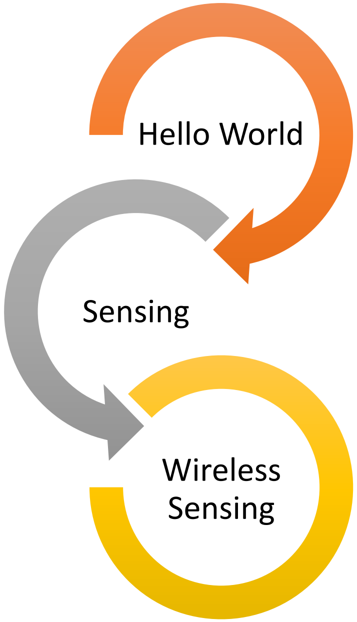
#include "dev/battery-sensor.h“, "dev/i2cmaster.h“, "dev/tmp102.h"

```
//the data structure
struct sensor_datamsg{

                uint16_t temp;
                uint16_t humm;
                uint16_t batt;
                uint16_t light1;
                uint16_t light2;
}sensor_datamsg;
```

```
PROCESS_THREAD(sense_process, ev, data)
{
                static struct etimer et;
                int16_t raw;
  uint16_t absraw;
                PROCESS_BEGIN(); /**Always first**/
//activate the sensors
 tmp102_init();

SENSORS_ACTIVATE(battery_sensor);

while (1) {

etimer_set(&et, CLOCK_SECOND);
PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
//get the data
  raw = tmp102_read_temp_raw();
    absraw = raw;
    if(raw < 0) {                  // Perform 2C's if sensor returned negative data
      absraw = (raw ^ 0xFFFF) + 1;

    }
msg.temp=  absraw;
msg.batt = battery_sensor.value(0);

//and now it is time to wake up the 2nd process
process_post(&print_and_blink_process,event_data_ready, &msg);



}
//deactivate
SENSORS_DEACTIVATE(battery_sensor);
PROCESS_END(); /**Always last**/
}
```

```
PROCESS_THREAD(print_and_blink_process, ev, data)
{
PROCESS_BEGIN(); /**Always first**/

while (1) {
PROCESS_YIELD_UNTIL(ev==event_data_ready);

//and then print
}
PROCESS_END(); /**Always last**/
}
```

Hands on Session



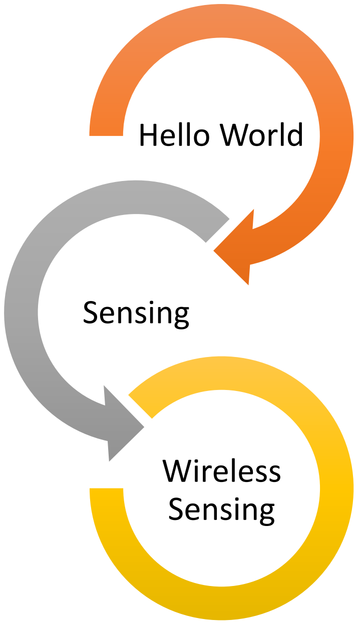
Wireless Sensing ☺
Access a sensor & trx using a broadcast RIME

Communication:

- Each type of connection (rime / uIP / 6LoWPAN) defines a structure
- Each type of rime connection defines a struct for the callback function (rx events).

  Callback function has to have a specific definition…

- Each rime-based connection is associated with a predefined channel (>128)


Hello World

Sensing

Wireless
Sensing

Hands on Session

Wireless Sensing ☺ contiki/examples/hello-world
[Access a sensor & trx]

@ rime:

- packetbuf module for packet buffer management
- Struct linkaddr_t for addressing…

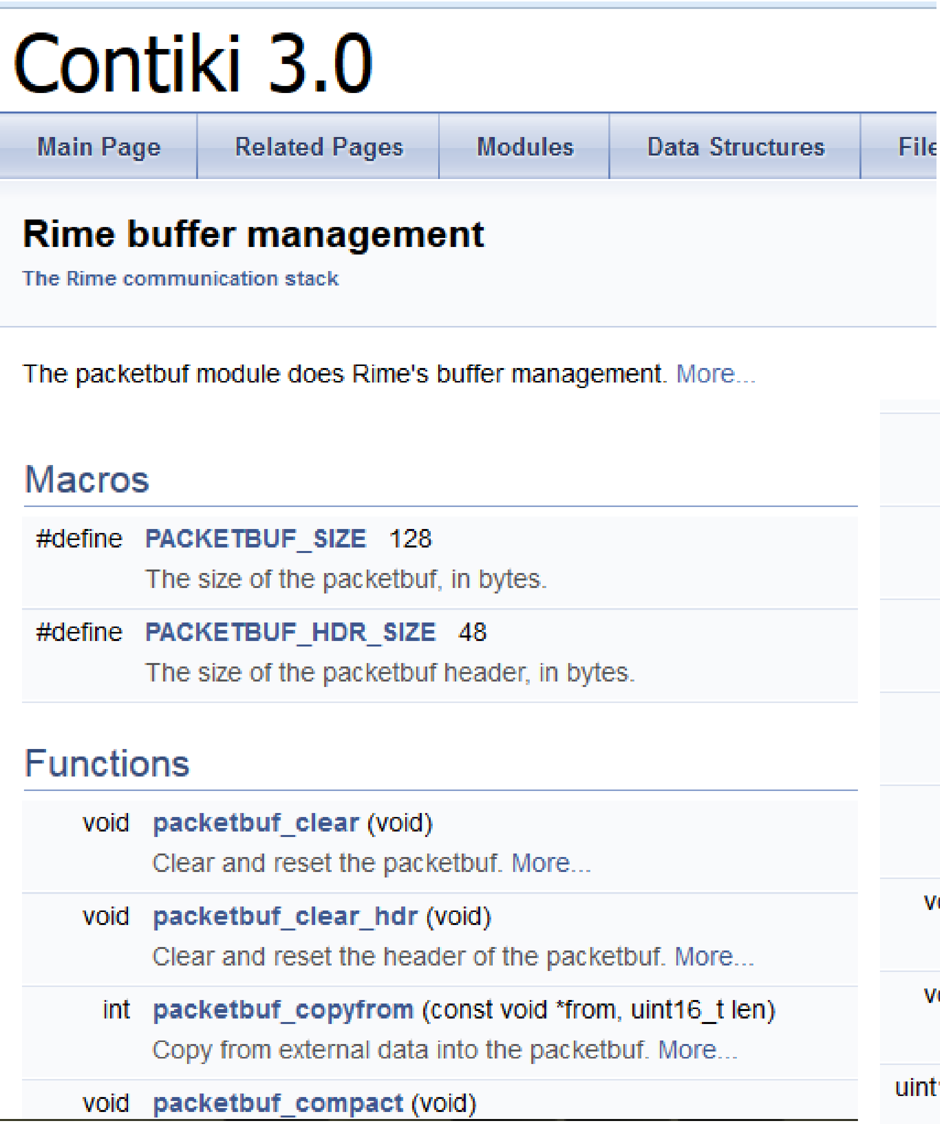
```
typedef union {
        unsigned char u8[LINKADDR_SIZE]; //=2
} linkaddr_t;
```

@ uip:

- uipbuf module for packet buffer management
- Struct ipaddr_t

Unless otherwise
specified,
IP=
176.12.RIME_ADDR[
0]. RIME_ADDR[1]

http://www.eistec.se/docs/contiki/



Contiki 3.0

Rime buffer management

The Rime communication stack

The packetbuf module does Rime's buffer management. More...

Macros

#define	**PACKETBUF_SIZE**	128	The size of the packetbuf, in bytes.
#define	**PACKETBUF_HDR_SIZE**	48	The size of the packetbuf header, in bytes.

Functions

void	**packetbuf_clear** (void)	Clear and reset the packetbuf. More...
void	**packetbuf_clear_hdr** (void)	Clear and reset the header of the packetbuf. More...
int	**packetbuf_copyfrom** (const void *from, uint16_t len)	Copy from external data into the packetbuf. More...
void	**packetbuf_compact** (void)	

int	**packetbuf_copyto_hdr** (uint8_t *to)	Copy the header portion of the packetbuf to an external buffer. More...
int	**packetbuf_copyto** (void *to)	Copy the entire packetbuf to an external buffer. More...
int	**packetbuf_hdralloc** (int size)	Extend the header of the packetbuf, for outbound packets. More...
int	**packetbuf_hdrreduce** (int size)	Reduce the header in the packetbuf, for incoming packets. More...
void	**packetbuf_set_datalen** (uint16_t len)	Set the length of the data in the packetbuf. More...
void *	**packetbuf_dataptr** (void)	Get a pointer to the data in the packetbuf. More...
void *	**packetbuf_hdrptr** (void)	Get a pointer to the header in the packetbuf, for outbound packets. More...
uint16_t	**packetbuf_datalen** (void)	Get the length of the data in the packetbuf. More...
uint8_t	**packetbuf_hdrlen** (void)	Get the length of the header in the packetbuf. More...
uint16_t	**packetbuf_totlen** (void)	Get the total length of the header and data in the packetbuf. More...
int	**packetbuf_holds_broadcast** (void)	Checks whether the current packet is a broadcast. More...

Hands on Session

```
//the data structure
struct sensor_datamsg{

		uint16_t temp;
		uint16_t humm;
		uint16_t batt;
		uint16_t light1;
		uint16_t light2;
}sensor_datamsg;
```

Wireless Sensing ☺ [Access a sensor & trx]

From the sense-and-tx, generate a new application (sense-and-trx.c) that:

1. Periodically samples from on-board temperature sensor
2. When done broadcast the value
3. Upon the reception of a incoming packet, print its contents and the source node id

Temperature & battery

Temperature, humidity, radiation & battery

```
#include “net/rime/rime.h”
#include “random.h”


static void
broadcast_recv(struct broadcast_conn *c, const linkaddr_t*from)
{
//processing upon RX
}

//DEFINE THE RX CALLBACK FUNCTION
static const struct broadcast_callbacks broadcast_call = {broadcast_recv}; -- visible outside
process



//DECLARE THE BROADCAST CHANNEL

static struct broadcast_conn broadcast;                                -- visible outside
process


PROCESS_THREAD(send_and_blink_process, ev, data)
{

}
```

Transmit process

```
PROCESS_THREAD(send_and_blink_process, ev, data)
{

static uint8_t data2send[sizeof(sensor_datamsg)];
static struct etimer send_timer;

PROCESS_EXITHANDLER(broadcast_close(&broadcast);)

PROCESS_BEGIN(); /**Always first**/

broadcast_open(&broadcast, 129, &broadcast_call);

while (1) {
        PROCESS_YIELD_UNTIL(ev==event_data_ready);

        data2send[0] = msg.temp & 255;//lsb
        data2send[1] = msg.temp >> 8;//msb

        data2send[2] = msg.humm & 255;
        data2send[3] = msg.humm >> 8;

        data2send[4] = msg.batt & 255;
        data2send[5] = msg.batt >> 8;

        PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&send_timer));
        packetbuf_clear();
        packetbuf_clear_hdr();
        packetbuf_copyfrom(data2send,sizeof(sensor_datamsg));
        broadcast_send(&broadcast);
        leds_toggle(LEDS_GREEN);

}
```

Instead of PROCESS_THREAD(print_and_blink_process, ev, data)

Receive function

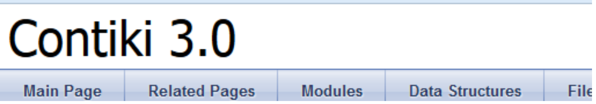
```
#include “net/rime/rime.h”
#include “random.h”
static void
broadcast_recv(struct broadcast_conn *c, const linkaddr_t*from)
{

 uint8_t *appdata;
  uint8_t i;
  appdata = (uint8_t *)packetbuf_dataptr();
  struct sensor_datamsg rxmsg;
 printf("******************broadcast message received from %d.%d\n", from->u8[0], from->u8[1]);
  if (packetbuf_datalen() == sizeof(struct sensor_datamsg)){
                  rxmsg.temp = appdata[0] | appdata[1]<<8;
                  rxmsg.humm = appdata[2] | appdata[3]<<8;
                  rxmsg.batt = appdata[4] | appdata[5]<<8;

                 printf("temp: %d, humm: %d, batt:%d\n", rxmsg.temp, rxmsg.humm, rxmsg.batt);


 }
 else{

                 for (i=0;i<packetbuf_datalen();i++){
                  printf("%u,", appdata[i]);
                  }

                printf("\n");

                 }
  }
```

Hands on Session

Wireless Sensing ☺ [Access a sensor & trx]

From the sense-and-tx, generate a new application (sense-and-trx.c) that:

1. Periodically samples from on-board temperature sensor
2. When done broadcast the value
3. Upon the reception of a incoming packet, print its contents and the source node id
4. USE **PACKETBUF attributes** to READ ALSO RSSI AND LQI VALUES



```
packetbuf_attr (at core/net/packetbuf.h):

PACKETBUF_ATTR_LINK_QUALITY,
PACKETBUF_ATTR_RSSI,
```

To use it: packetbuf_attr(type of attribute)

```
/* Scope 0 attributes: used only on the local node. */
PACKETBUF_ATTR_CHANNEL,//or  integer value=1
PACKETBUF_ATTR_NETWORK_ID,//or  integer value=2
PACKETBUF_ATTR_LINK_QUALITY,//or  integer value=3
PACKETBUF_ATTR_RSSI,//or  integer value=4
PACKETBUF_ATTR_TIMESTAMP,//or  integer value=5
PACKETBUF_ATTR_RADIO_TXPOWER,//or  integer value=6
PACKETBUF_ATTR_LISTEN_TIME,//or  integer value=7
PACKETBUF_ATTR_TRANSMIT_TIME,//or  integer value=8
PACKETBUF_ATTR_MAX_MAC_TRANSMISSIONS,//or  integer value=9
PACKETBUF_ATTR_MAC_SEQNO,//or  integer value=10
PACKETBUF_ATTR_MAC_ACK,//or  integer value=11

/* Scope 1 attributes: used between two neighbors only. */
PACKETBUF_ATTR_RELIABLE,//or  integer value=12
PACKETBUF_ATTR_PACKET_ID,//or  integer value=13
PACKETBUF_ATTR_PACKET_TYPE,//or  integer value=14
PACKETBUF_ATTR_REXMIT,//or  integer value=15
PACKETBUF_ATTR_MAX_REXMIT,//or  integer value=16
PACKETBUF_ATTR_NUM_REXMIT,//or  integer value=17
PACKETBUF_ATTR_PENDING,//or  integer value=18

/* Scope 2 attributes: used between end-to-end nodes. */
PACKETBUF_ATTR_HOPS,//or  integer value=19
PACKETBUF_ATTR_TTL,//or  integer value=20
PACKETBUF_ATTR_EPACKET_ID,//or  integer value=21
PACKETBUF_ATTR_EPACKET_TYPE,//or  integer value=22
PACKETBUF_ATTR_ERELIABLE,//or  integer value=23

/* These must be last */
PACKETBUF_ADDR_SENDER,//or  integer value=24
PACKETBUF_ADDR_RECEIVER,//or  integer value=25
PACKETBUF_ADDR_ESENDER,//or  integer value=26
PACKETBUF_ADDR_ERECEIVER,//or  integer value=27
```

```
broadcast_recv(struct broadcast_conn *c, const linkaddr_t *from)
{
  uint8_t *appdata;
  uint8_t i;
  appdata = (uint8_t *)packetbuf_dataptr();
  struct sensor_datamsg rxmsg;
 printf("******************broadcast message received from %d.%d\n", from->u8[0], from->u8[1]);
  if (packetbuf_datalen() == sizeof(struct sensor_datamsg)){
                    rxmsg.temp = appdata[0] | appdata[1]<<8;
                    rxmsg.humm = appdata[2] | appdata[3]<<8;
                    rxmsg.batt = appdata[4] | appdata[5]<<8;

                   printf("temp: %d, humm: %d, batt:%d\n", rxmsg.temp, rxmsg.humm, rxmsg.batt);


 }
 else{

                   for (i=0;i<packetbuf_datalen();i++){
                    printf("%u,", appdata[i]);
                    }

                  printf("\n");

                   }
                   //this is the id of the sender (as defined in compile time).
//printf(" from: %d.%d ",from->u8[0], from->u8[1]);
                  printf("with RSSI: %d and LQI:%d*****************\n", packetbuf_attr(PACKETBUF_ATTR_RSSI),
packetbuf_attr(PACKETBUF_ATTR_LINK_QUALITY));

}
```

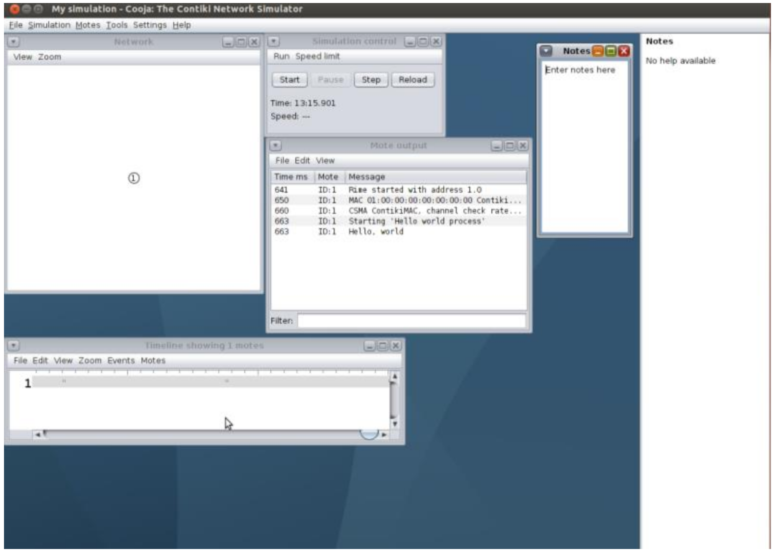
COOJA SESSION

WSN Programming II

Contiki

The Open Source OS for the Internet of Things

- Cooja
  - The Contiki emulator for running WSN applications.
  - Very useful for debugging your codes – the same code you test on cooja, the same you upload to your mote
  - Evaluating the network performance – has very simplifying models for radio propagation….
    - Unit disk model: Edges are instantly configured according to power attenuation w.r.t to distance & success ratio (configurable)
    - Directed graph radio medium: Considers preconfigured edges, without checking the output power.
    - Multipath ray tracer: Simulates reflection and diffraction through homogeneous obstacles (considers that all nodes have the same transmission power)
  - Interacts with external tools, e.g. Wireshark for Network monitoring
  - Modular: Plugins for extending functionality



Time: 13:15.901
Speed: ---

```
Time ms  Mote  Message
641      ID:1  Rime started with address 1.0
650      ID:1  MAC 01:00:00:00:00:00:00:00 Contiki...
660      ID:1  CSMA ContikiMAC, channel check rate...
663      ID:1  Starting 'Hello world process'
663      ID:1  Hello, world
```

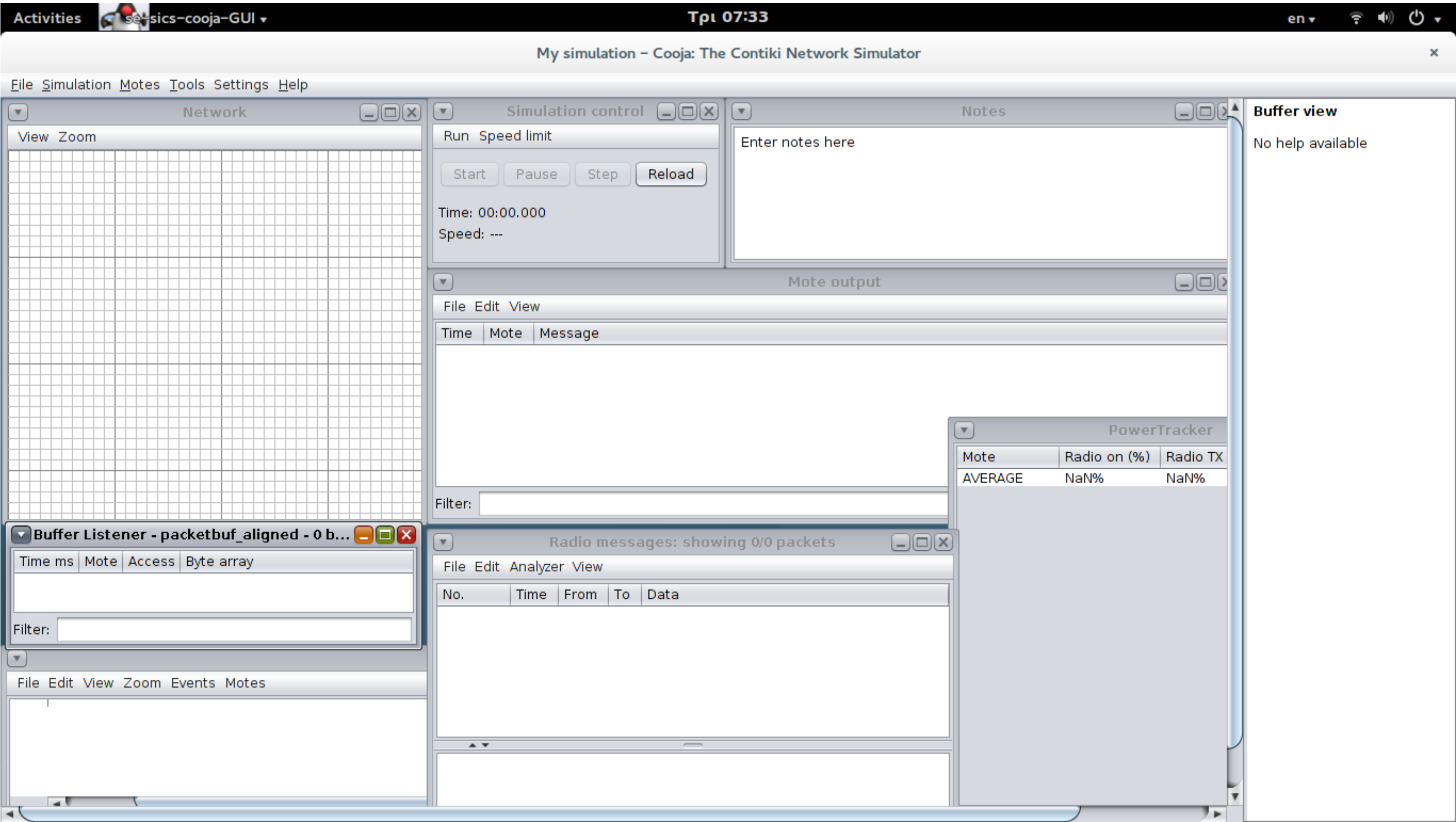
WSN Programming II

At your working directory (terminal):

cd tools/cooja

ant run

File -> New Simulation



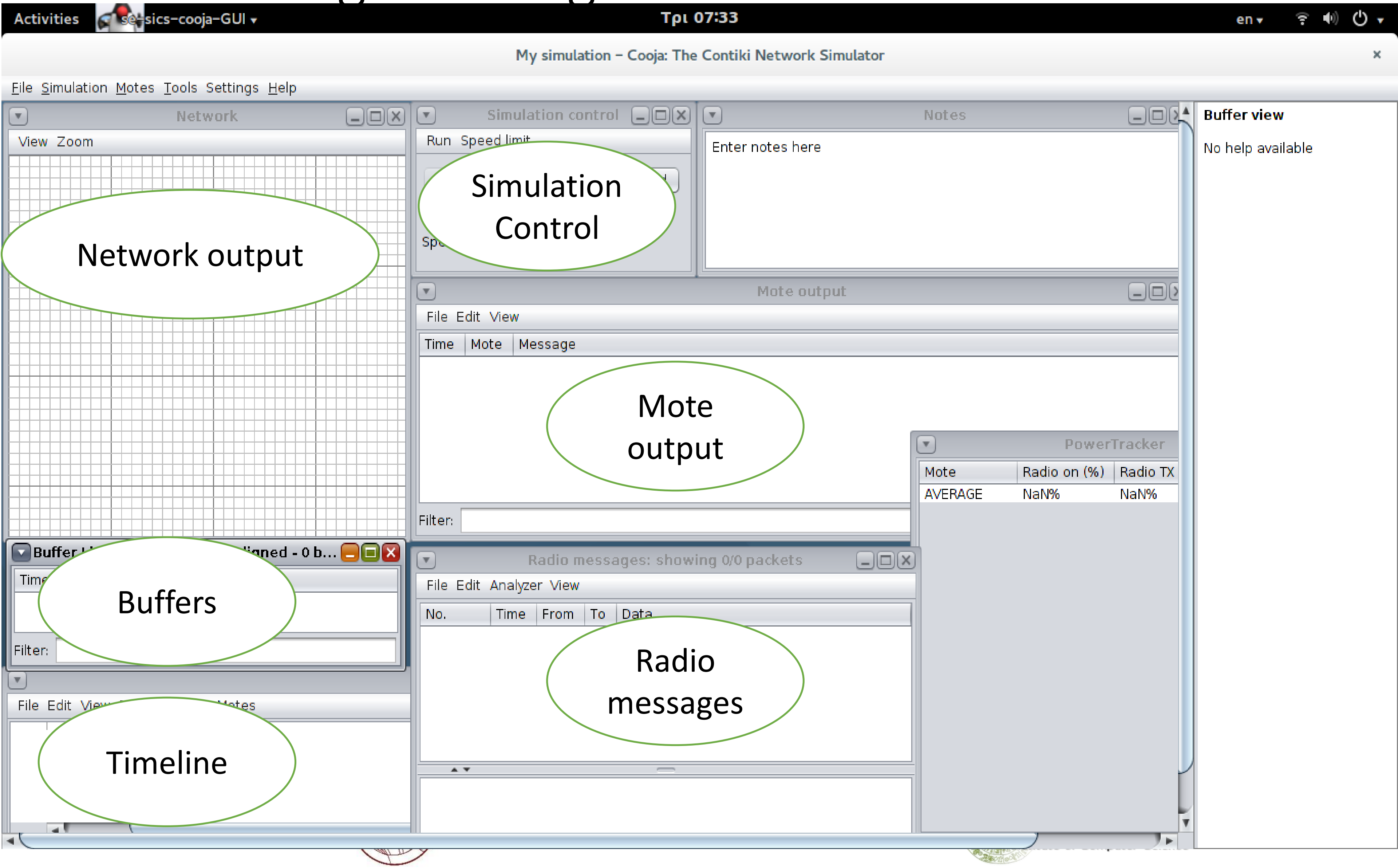
Time: 00:00.000
Speed: ---

Enter notes here

No help available

Mote	Radio on (%)	Radio TX
AVERAGE	NaN%	NaN%

WSN Programming II



Network output

Simulation Control

Mote output

Buffers

Timeline

Radio messages

Mote	Radio on (%)	Radio TX
AVERAGE	NaN%	NaN%

WSN Programming II

Create a new mote and upload a program:

Motes -> Create New Mote -> Sky

Locate program sense-and-trx.c

Clean

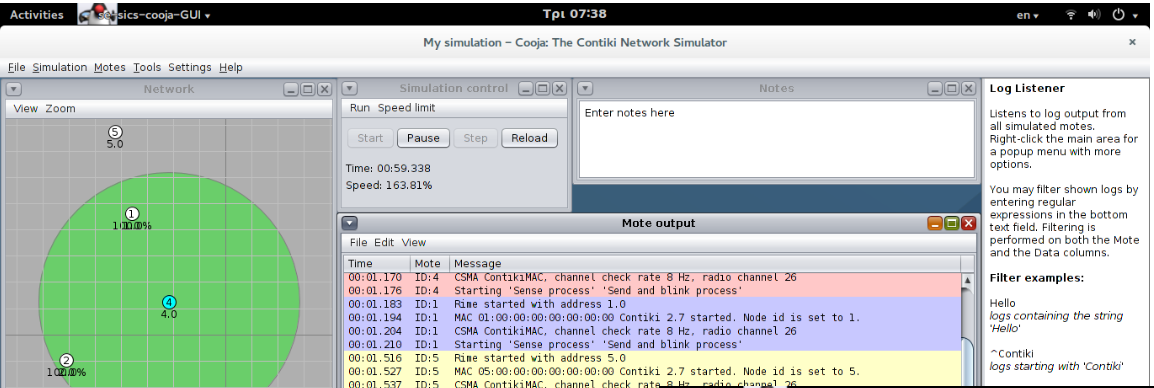
Compile

Create (5)

@simulation control: Start

WSN Programming II

sense-and-trx.c



Time: 00:59.338
Speed: 163.81%

Enter notes here

```
Time       Mote  Message
00:01.170  ID:4  CSMA ContikiMAC, channel check rate 8 Hz, radio channel 26
00:01.176  ID:4  Starting 'Sense process' 'Send and blink process'
00:01.183  ID:1  Rime started with address 1.0
00:01.194  ID:1  MAC 01:00:00:00:00:00:00:00 Contiki 2.7 started. Node id is set to 1.
00:01.204  ID:1  CSMA ContikiMAC, channel check rate 8 Hz, radio channel 26
00:01.210  ID:1  Starting 'Sense process' 'Send and blink process'
00:01.516  ID:5  Rime started with address 5.0
00:01.527  ID:5  MAC 05:00:00:00:00:00:00:00 Contiki 2.7 started. Node id is set to 5.
00:01.537  ID:5  CSMA ContikiMAC, channel check rate
00:01.542  ID:5  Starting 'Sense process' 'Send and
00:01.701  ID:3  Rime started with address 3.0
00:01.711  ID:3  MAC 03:00:00:00:00:00:00:00 Contiki
00:01.721  ID:3  CSMA ContikiMAC, channel check rate
00:01.727  ID:3  Starting 'Sense process' 'Send and
```

Log Listener

Listens to log output from all simulated motes. Right-click the main area for a popup menu with more options.

You may filter shown logs by entering regular expressions in the bottom text field. Filtering is performed on both the Mote and the Data columns.

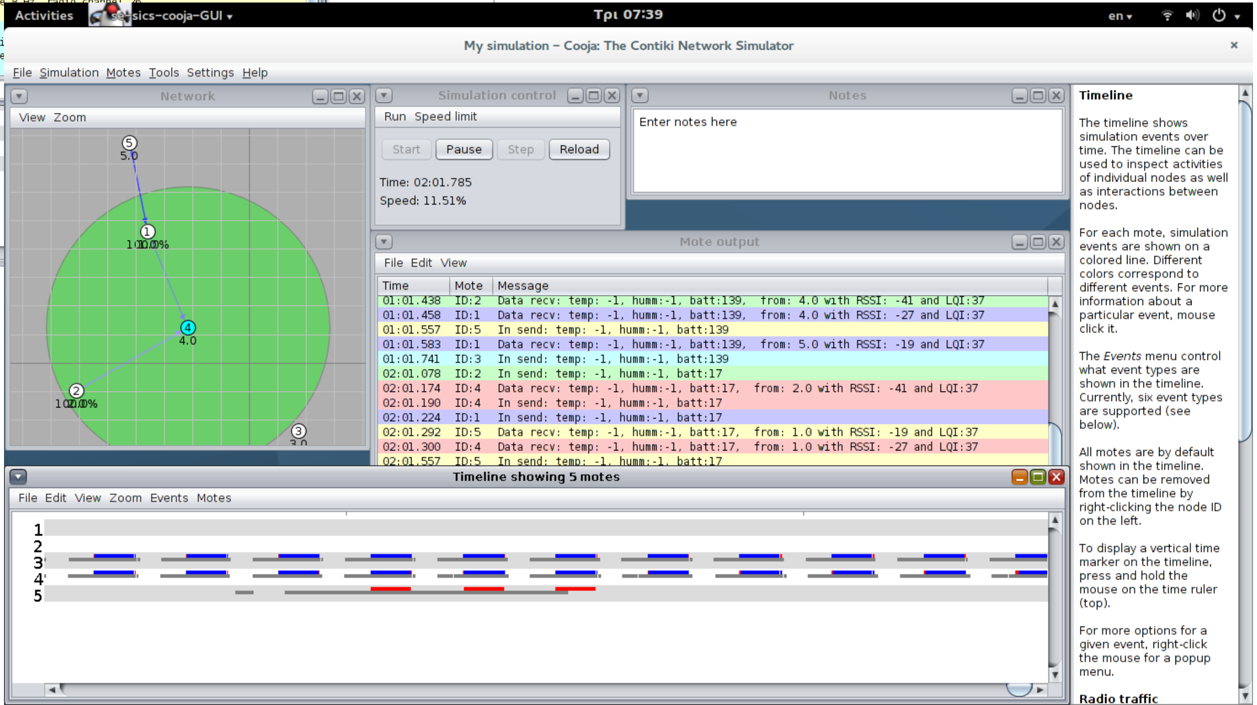
Filter examples:

Hello
logs containing the string 'Hello'

^Contiki
logs starting with 'Contiki'

Check out the timeline

Transmissions: blue
Receptions: green
Congestions: red



Time: 02:01.785
Speed: 11.51%

Enter notes here

```
Time       Mote  Message
01:01.438  ID:2  Data recv: temp: -1, humm:-1, batt:139,  from: 4.0 with RSSI: -41 and LQI:37
01:01.458  ID:1  Data recv: temp: -1, humm:-1, batt:139,  from: 4.0 with RSSI: -27 and LQI:37
01:01.557  ID:5  In send: temp: -1, humm:-1, batt:139
01:01.583  ID:1  Data recv: temp: -1, humm:-1, batt:139,  from: 5.0 with RSSI: -19 and LQI:37
01:01.741  ID:3  In send: temp: -1, humm:-1, batt:139
02:01.078  ID:2  In send: temp: -1, humm:-1, batt:17
02:01.174  ID:4  Data recv: temp: -1, humm:-1, batt:17,  from: 2.0 with RSSI: -41 and LQI:37
02:01.190  ID:4  In send: temp: -1, humm:-1, batt:17
02:01.224  ID:1  In send: temp: -1, humm:-1, batt:17
02:01.292  ID:5  Data recv: temp: -1, humm:-1, batt:17,  from: 1.0 with RSSI: -19 and LQI:37
02:01.300  ID:4  Data recv: temp: -1, humm:-1, batt:17,  from: 1.0 with RSSI: -27 and LQI:37
02:01.557  ID:5  In send: temp: -1, humm:-1, batt:17
```

Timeline

The timeline shows simulation events over time. The timeline can be used to inspect activities of individual nodes as well as interactions between nodes.

For each mote, simulation events are shown on a colored line. Different colors correspond to different events. For more information about a particular event, mouse click it.

The *Events* menu control what event types are shown in the timeline. Currently, six event types are supported (see below).

All motes are by default shown in the timeline. Motes can be removed from the timeline by right-clicking the node ID on the left.

To display a vertical time marker on the timeline, press and hold the mouse on the time ruler (top).

For more options for a given event, right-click the mouse for a popup menu.

Radio traffic

WSN Programming II

Pause, Reload, Save & Re-open

(also: open .csc file!)

- Tools & Extensions:
  - Bufferline, Mobility, MSP Code Watcher, Powertracker

WSN Programming II

Change protocol stack parameters: e.g. change RDC policy to no RDC

project-conf.h

```
#define NETSTACK_CONF_RDC      nullrdc_driver

#define NEIGHS_TIMEOUT 120

#define MAX_NEIGHS 16

#define NET_SIZE 10
```

@Makefile

```
#CFLAGS += -DPROJECT_CONF_H=\"project-conf.h\"
```

WSN Programming II

sense-and-trx.c: opens a bcast channel (RIME), senses a few data and sends out values & print the RSSI and LQI values & **store the 1st hop neighbors**

Use of LIST in CONTIKI.

```
/* This MEMB() definition defines a memory pool from which we allocate
packet entries entries. */
MEMB(neighs_memb, struct neighs, MAX_NEIGHS);
/* The packets2send_list is a Contiki list that holds the packets pending for sending.*/
LIST(neighs_list);
```

```
/* This structure holds information about the 1st hop neighbours. */
struct neighs {
 /* The ->next pointer is needed since we are placing these on a
Contiki list. */
 struct neighs   *next;
 linkaddr_t linkaddr;
 //the time out timer for removing old entries
 struct ctimer ctimer; };
static void update_neighs(void);
static void remove_neighs(void *n);
```

```
 static void update_neighs(void)
 {
           struct neighs *n;
           linkaddr_t *tmp;

           tmp = (linkaddr_t *)packetbuf_addr(PACKETBUF_ADDR_SENDER);

           // Check if we already know this child.
           for(n = list_head(neighs_list); n != NULL; n = list_item_next(n)) {

                      /* We break out of the loop if the linkaddr of the sender matches
                      the address of the neigbour from which we received this msg */
                      if (linkaddr_cmp(tmp, &n->linkaddr)){
                       /* Our neigh was found, so we update the timeout. */
                      ctimer_set(&n->ctimer, NEIGHS_TIMEOUT*CLOCK_SECOND, remove_neighs, n);
                                 break;
                      }}

  /* If n is NULL, this child was not found in our list, and we
allocate a new struct child from the children_memb memory
pool.*/
  if(n == NULL) {
    n = memb_alloc(&neighs_memb);

    /* If we could not allocate a new children entry, we give up. We
could have reused an old neighbor entry, but we do not do this
for now. */
    if(n != NULL) {
    /* Initialize the fields. */
    linkaddr_copy(&n->linkaddr, packetbuf_addr(PACKETBUF_ADDR_SENDER));
    memcpy(&n->linkaddr, (linkaddr_t *)packetbuf_addr(PACKETBUF_ADDR_SENDER),sizeof(linkaddr_t));

    /* Place the child on the children list at the end of the list. */
    list_add(neighs_list, n);

    ctimer_set(&n->ctimer, NEIGHS_TIMEOUT*CLOCK_SECOND, remove_neighs, n);
    }
   }
}
```

```
/*
 * This function is called by the ctimer present in each neighbor
 * table entry. The function removes the neighbor from the table
 * because it has become too old.*/

static void remove_neighs(void *n)
{
  struct neighs *e = n;
 //removing old items...
 //printf("now removing node: %d\n",e->addr.addr[5]);
  list_remove(neighs_list, e);
  memb_free(&neighs_memb, e);
}
```

```
uint8_t get_neighslist(linkaddr_t *listaddr, uint8_t size)
{

   struct neighs *n;
   linkaddr_t tmp;

   uint8_t i;
   if (size >0){

   i=0;
   for(n = list_head(neighs_list); n != NULL; n =
list_item_next(n)) {
       memcpy(&tmp, &n->linkaddr, sizeof(linkaddr_t));
       listaddr[i++] = tmp;
   }
   }
   else {

       listaddr = NULL;
   }
   return i;

}
```

WSN Programming II

sense-and-trx.c: opens a bcast channel (RIME), senses a few data and sends out values & print the RSSI and LQI values & **store the 1st hop neighbors**

Use of LIST in CONTIKI.

Checks and print the 1st hop neighs every 1 min

```
list_length(list)
```

WSN Programming II

sense-and-trx.c: opens a bcast channel (RIME), senses a few data and sends out values & print the RSSI and LQI values & **store the 1st hop neighbors**

Use of LIST in CONTIKI.

Checks and print the 1st hop neighs every 1 min

```
        printf("Current length of neighbours list:%d\n",
list_length(neighs_list));

        if (list_length(neighs_list) >0 )
        {
                 static linkaddr_t tmplist[MAX_NEIGHS];
                 get_neighslist(tmplist, list_length(neighs_list));
                 for (ii=0; ii<list_length(neighs_list);ii++)
                 {

                  printf("**%d.%d:: **", tmplist[ii].u8[0],tmplist[ii].u8[1]);

                 }
                 printf("\n");

        }
```

WSN Programming II

example-multihop.c: a simplified routing algorithm (RIME) using *announcements* for creating 1st hop neighborhood and generate traffic towards a specific sensor node (1) when pressing a button.

```
/* Initialize the memory for the neighbor table entries. */
  memb_init(&neighbor_mem);

  /* Initialize the list used for the neighbor table. */
  list_init(neighbor_table);

  /* Open a multihop connection on Rime channel CHANNEL. */
  multihop_open(&multihop, CHANNEL, &multihop_call);

  /* Register an announcement with the same announcement ID as the
     Rime channel we use to open the multihop connection above. */
  announcement_register(&example_announcement,
                        CHANNEL,
                        received_announcement);
```

WSN Programming II

example-multihop.c: a simplified routing algorithm (RIME) using announcements for creating 1st hop neighborhood and generate traffic towards a specific sensor node (1) when pressing a button.

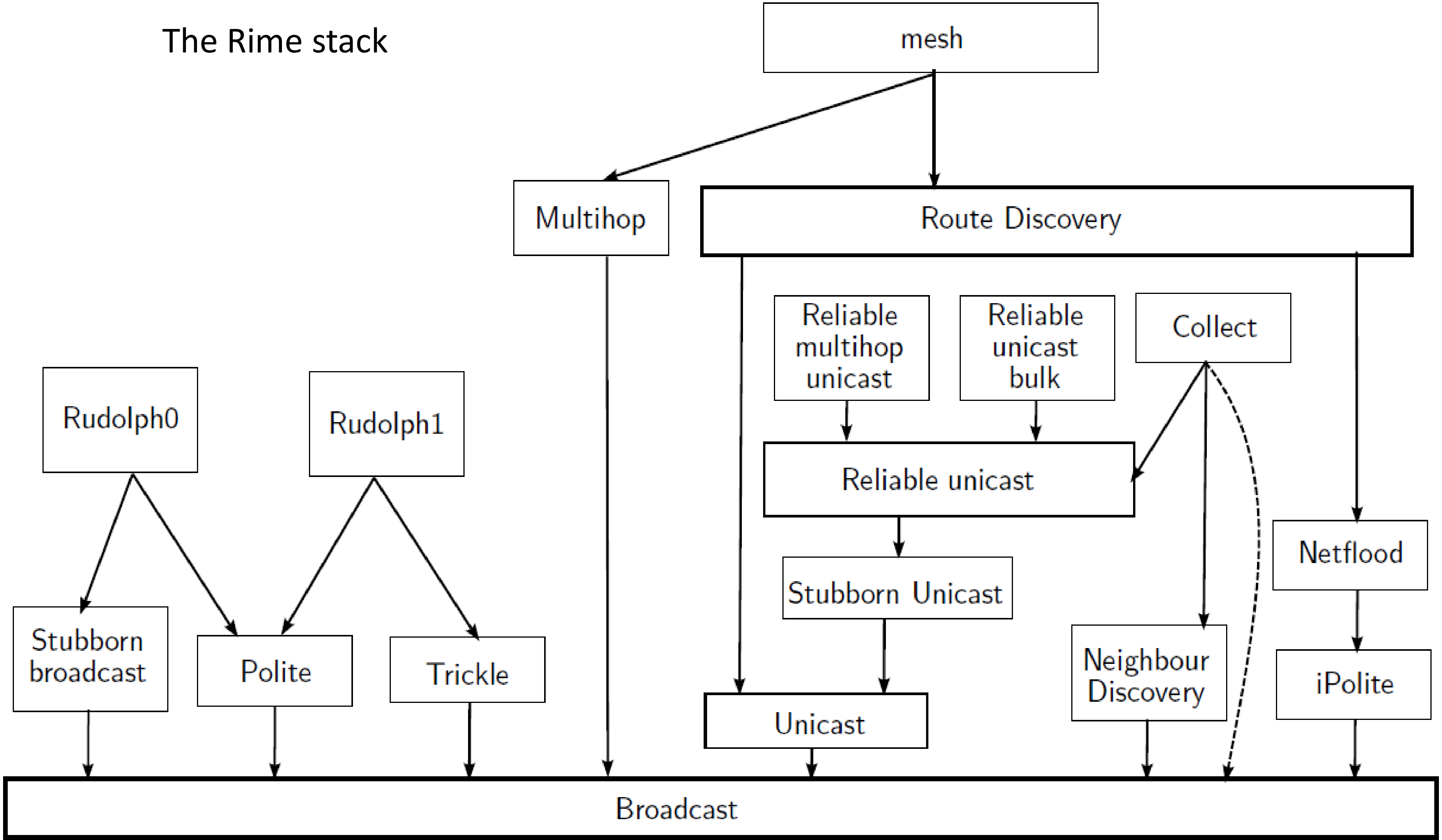
```
 /* Activate the button sensor. We use the button to drive 
traffic -
     when the button is pressed, a packet is sent. */
  SENSORS_ACTIVATE(button_sensor);
```

```
 /* Wait until we get a sensor event with the button sensor as 
data. */
    PROCESS_WAIT_EVENT_UNTIL(ev == sensors_event &&
                             data == &button_sensor);
```

WSN Programming

Contiki
The Open Source OS for the Internet of Things

The Rime stack



Hands on Session

Contiki

The Open Source OS for the Internet of Things

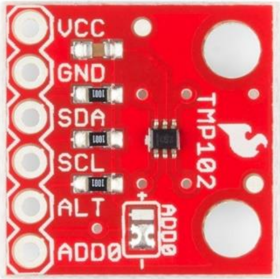
What we are going to use…in order to upload code to the motes

- FTDI drivers (for Windows machines only) – USB2Serial
- How the host computer reserves a mote:
  - COM<No> (Windows – Device Manager)
  - /dev/ttyUSB<No> (Linux) [**cat /var/log/syslog**]
  - Make sure that you have access on device (for programming it)

    sudo addgroup <user> dialout (log out & then back in)
  - Serial dump: make TARGET=sky MOTES=**/dev/ttyUSB0** login

    (note: make sure you have permissions to execute serialdump-linux @ $contikifolder/tools/sky)

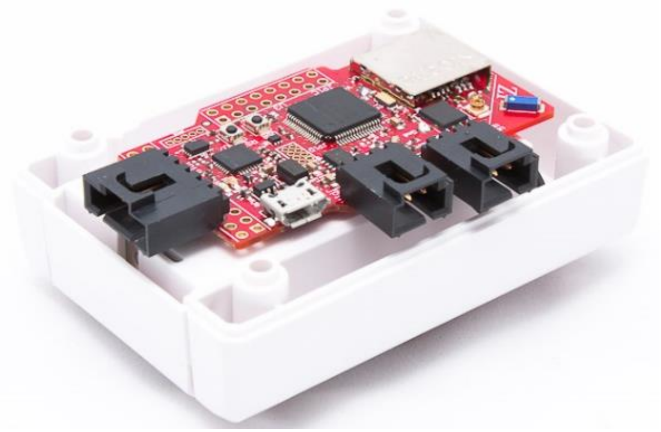
TI TMP102 -> TEMPERATURE

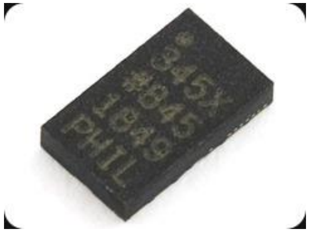


HOW TO ACCESS IT:

A. USE THE API PROVIDED BY THE PLATFORM'S API

```
void  tmp102_init(void);

uint16_t tmp102_read_temp_raw();
```





ADLX345 -> 3-axis digital accelerometer

HOW TO ACCESS IT:

A. USE THE API PROVIDED BY THE PLATFORM'S API

```
void    accm_init(void);
int16_t accm_read_axis(enum ADXL345_AXIS axis);
```

CHECK examples/z1/test-adlx345.c to see how to access the sensor.

WSN Programming II

example-multihop.c: a simplified routing algorithm (RIME) using announcements for creating 1st hop neighborhood and generate traffic towards a specific sensor node (1) when pressing a button.

Modifications:

(1) random selection of the destination

(2) hop-to-live = 16

(3) bi-directional link: Send a “Request” and the destination replies with a “Reply”

```
 while (to.u8[0] == linkaddr_node_addr.u8[0] || to.u8[0] ==0){

      to.u8[0] = random_rand() % NET_SIZE;
      }
   to.u8[1] = 0;
   printf("packet ready to send to:%d.%d\n", to.u8[0],
to.u8[1]);
```

```
packetbuf_copyfrom("Request", 7);
```

```
 //reply to sender if message is 'Request'.
  if (strcmp((char *)packetbuf_dataptr(), "Request") == 0){
  linkaddr_copy(&request_sender, sender);
  process_post(&reply_process,event_data_ready,
&request_sender);
  }
```

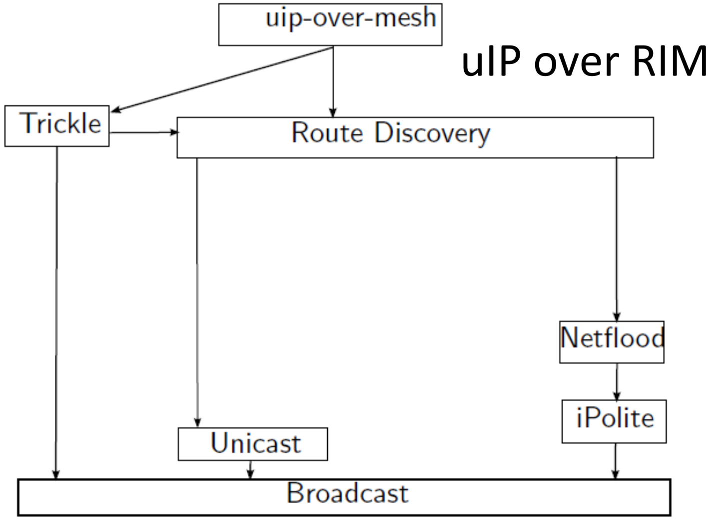
```
/*---------------------------------------------------------------------------*/
PROCESS_THREAD(reply_process, ev, data)
{

static rimeaddr_t toreply;
PROCESS_BEGIN(); /**Always first**/



while (1) {
//this process sleeps until somebody wakes it up.
            PROCESS_YIELD_UNTIL(ev==event_data_ready);
            rimeaddr_copy(&toreply, &request_sender);
            //and prepare the buffer
            packetbuf_clear();
            packetbuf_clear_hdr();
//          packetbuf_copyfrom(data2send,sizeof(sensor_datamsg));
            packetbuf_copyfrom("Reply", 5);
            //and now send
             multihop_send(&multihop, &toreply);

            leds_toggle(LEDS_GREEN);

}
PROCESS_END(); /**Always last**/
}
```

WSN Programming II

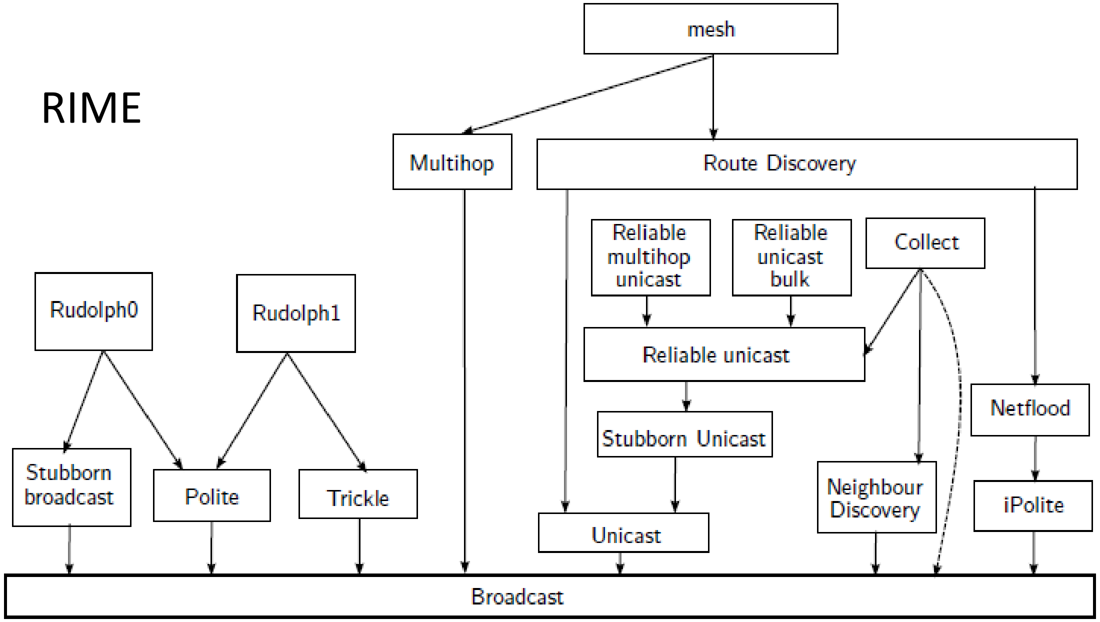
RPL and uIP stack (client-server connection).

Now we are using the uIP stack – not RIME

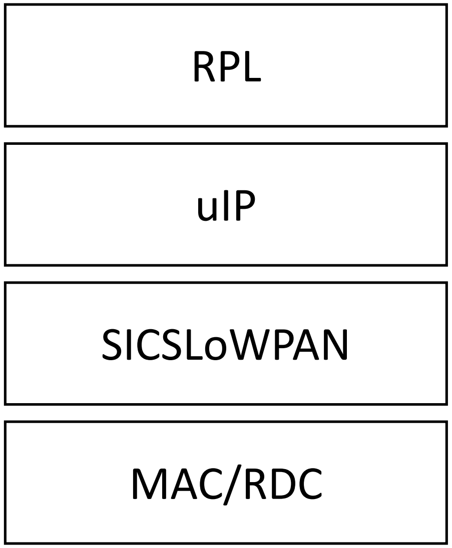
uIP over RIME



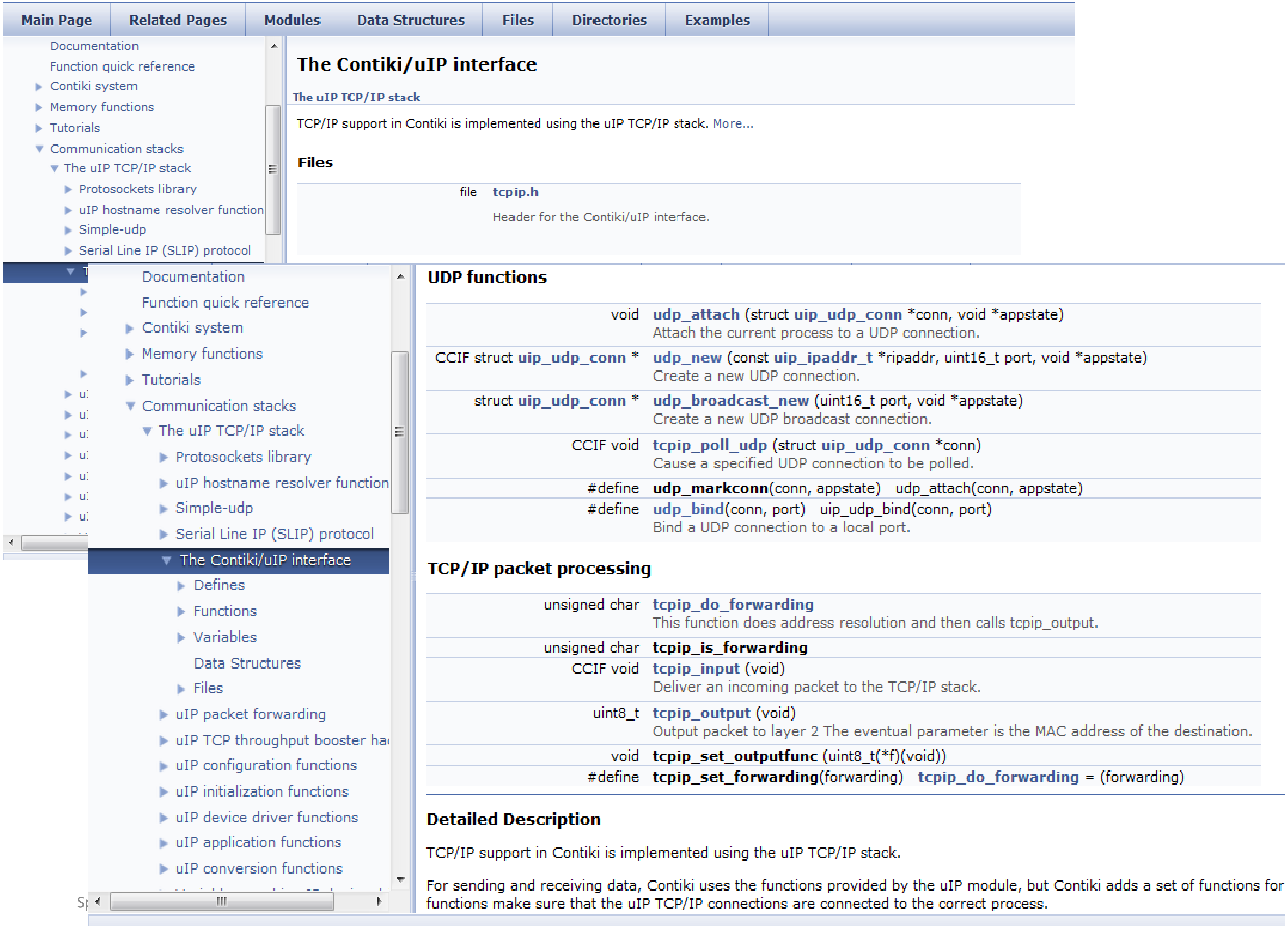
RIME



uIP with RPL



Contiki 2.6



The Contiki/uIP interface

The uIP TCP/IP stack

TCP/IP support in Contiki is implemented using the uIP TCP/IP stack. More...

Files

file	**tcpip.h** Header for the Contiki/uIP interface.

UDP functions

void	**udp_attach** (struct **uip_udp_conn** *conn, void *appstate) Attach the current process to a UDP connection.
CCIF struct **uip_udp_conn** *	**udp_new** (const **uip_ipaddr_t** *ripaddr, uint16_t port, void *appstate) Create a new UDP connection.
struct **uip_udp_conn** *	**udp_broadcast_new** (uint16_t port, void *appstate) Create a new UDP broadcast connection.
CCIF void	**tcpip_poll_udp** (struct **uip_udp_conn** *conn) Cause a specified UDP connection to be polled.
#define	**udp_markconn**(conn, appstate) udp_attach(conn, appstate)
#define	**udp_bind**(conn, port) uip_udp_bind(conn, port) Bind a UDP connection to a local port.

TCP/IP packet processing

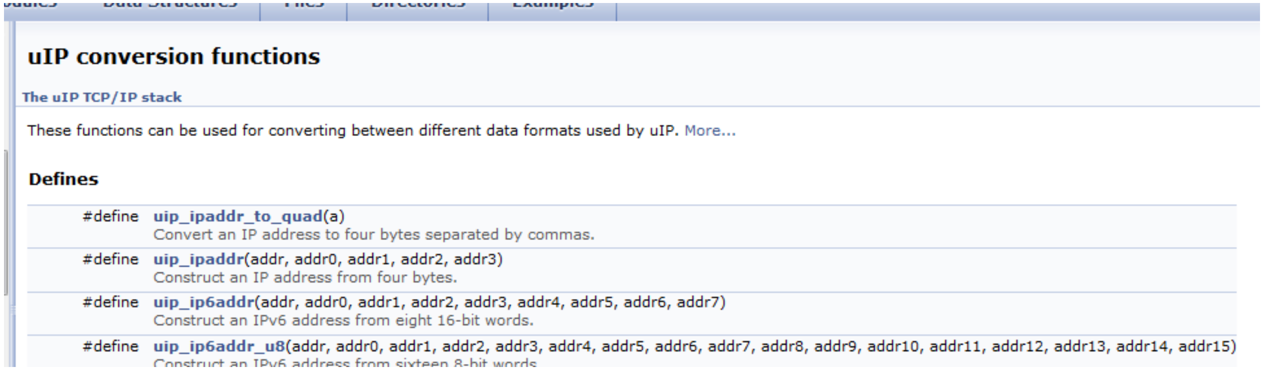
unsigned char	**tcpip_do_forwarding** This function does address resolution and then calls tcpip_output.
unsigned char	**tcpip_is_forwarding**
CCIF void	**tcpip_input** (void) Deliver an incoming packet to the TCP/IP stack.
uint8_t	**tcpip_output** (void) Output packet to layer 2 The eventual parameter is the MAC address of the destination.
void	**tcpip_set_outputfunc** (uint8_t(*f)(void))
#define	**tcpip_set_forwarding**(forwarding) **tcpip_do_forwarding** = (forwarding)

Detailed Description

TCP/IP support in Contiki is implemented using the uIP TCP/IP stack.

For sending and receiving data, Contiki uses the functions provided by the uIP module, but Contiki adds a set of functions for
functions make sure that the uIP TCP/IP connections are connected to the correct process.

WSN Programming II



uIP conversion functions

The uIP TCP/IP stack

These functions can be used for converting between different data formats used by uIP. More...

Defines

#define	**uip_ipaddr_to_quad**(a) Convert an IP address to four bytes separated by commas.
#define	**uip_ipaddr**(addr, addr0, addr1, addr2, addr3) Construct an IP address from four bytes.
#define	**uip_ip6addr**(addr, addr0, addr1, addr2, addr3, addr4, addr5, addr6, addr7) Construct an IPv6 address from eight 16-bit words.
#define	**uip_ip6addr_u8**(addr, addr0, addr1, addr2, addr3, addr4, addr5, addr6, addr7, addr8, addr9, addr10, addr11, addr12, addr13, addr14, addr15) Construct an IPv6 address from sixteen 8-bit words.

At Makefile:

```
WITH_UIP6=1
UIP_CONF_IPV6=1
```

test-rpl-sink.c

```
if(root_if != NULL) {
    rpl_dag_t *dag;
    dag = rpl_set_root(RPL_DEFAULT_INSTANCE,(uip_ip6addr_t
*)&ipaddr);
    uip_ip6addr(&ipaddr, 0xaaaa, 0, 0, 0, 0, 0, 0, 0);
    rpl_set_prefix(dag, &ipaddr, 64);
    PRINTF("created a new RPL dag\n");

  }
```

WSN Programming II

RPL and uIP stack (client-server connection).

test-rpl-sink.c

```
server_conn = udp_new(NULL, UIP_HTONS(UDP_CLIENT_PORT),
NULL);
  udp_bind(server_conn, UIP_HTONS(UDP_SERVER_PORT));
```

test-rpl-source.c

```
client_conn = udp_new(NULL,UIP_HTONS(UDP_SERVER_PORT), NULL);
```

```
      if(client_conn == NULL) {
  //  PRINTF("No UDP connection available, exiting the
process!\n");
      PROCESS_EXIT();
      }
      udp_bind(client_conn, UIP_HTONS(UDP_CLIENT_PORT));
```

WSN Programming II

RPL and uIP stack (client-server connection).

test-rpl-sink.c

```
server_conn = udp_new(NULL, UIP_HTONS(UDP_CLIENT_PORT),
NULL);
  udp_bind(server_conn, UIP_HTONS(UDP_SERVER_PORT));
```

test-rpl-source.c

Schedule the transmission at ~6seconds after the expire of the *periodic* timer.