Design and Development of IoT Applications

Dr. –Ing. Vo Que Son

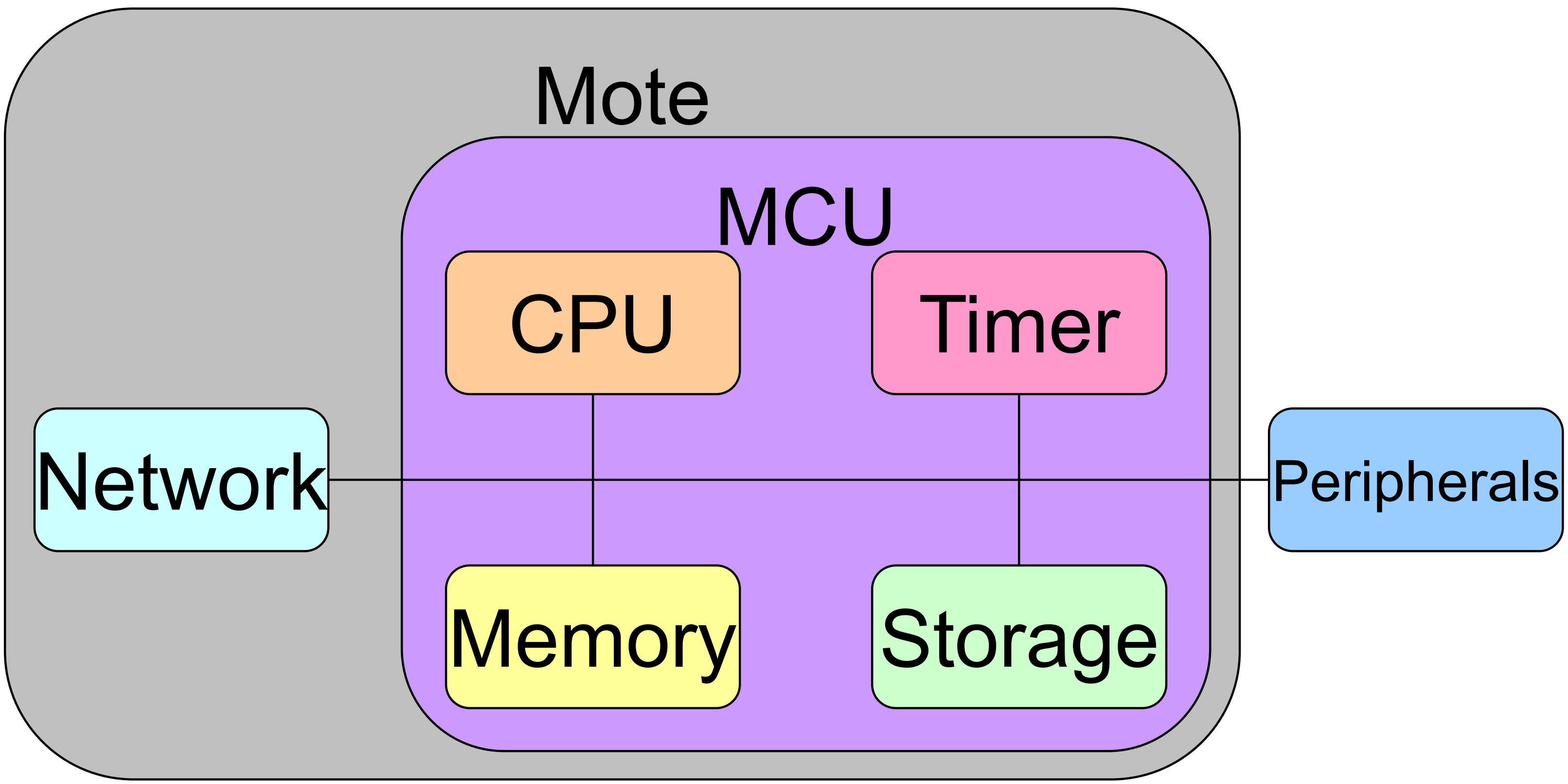
Email: sonvq@hcmut.edu.vn

Content

- Chapter 3: Embedded OS for end-devices
  - Intro to Contiki-OS
  - Programming using Contiki
  - I/O interfaces
  - Networking stack
  - Cooja Emulator
- Chapter 4: MAC protocols for WSNs
  - Low-power link
  - Robust communication
  - Radio Duty Cycling
  - Synchronized and Asynchronized Protocols

Computer Systems

- Traditional systems: separate chips
- Microcontroller: integrate on single chip

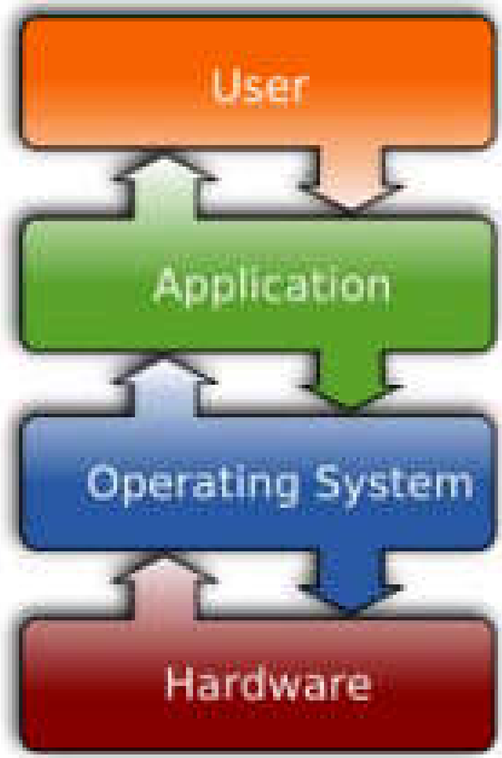


Mote Characteristics

- ❑ Limited resources
  - ❖ RAM, ROM, Computation, Energy
    - Wakeup, do work as quickly as possible, sleep
- ❑ Hardware modules operate concurrently
  - ❖ No parallel execution of code (not Core 2 Duos!)
    - Asynchronous operation is first class
- ❑ Diverse application requirements
  - ❖ Efficient modularity
- ❑ Robust operation
  - ❖ Numerous, unattended, critical
    - Predictable operation

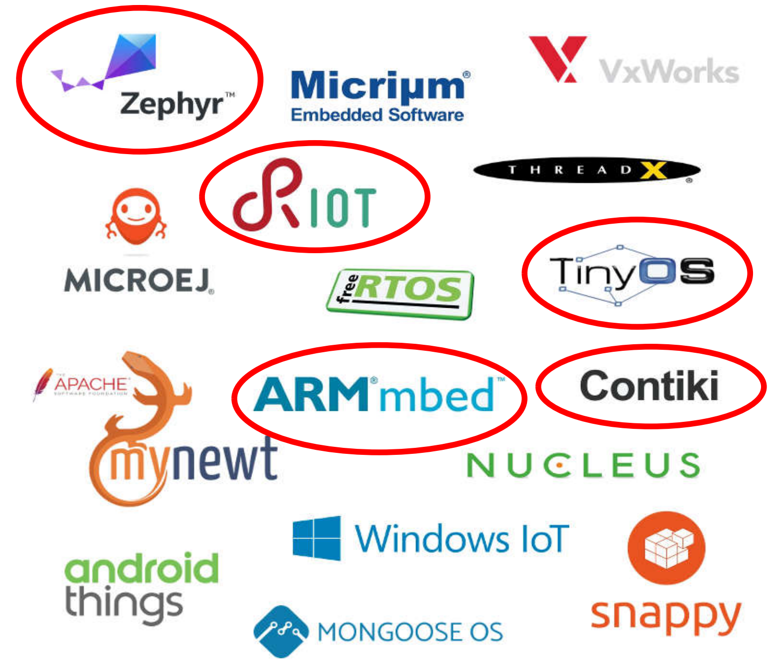
What is an Operating System (OS)?

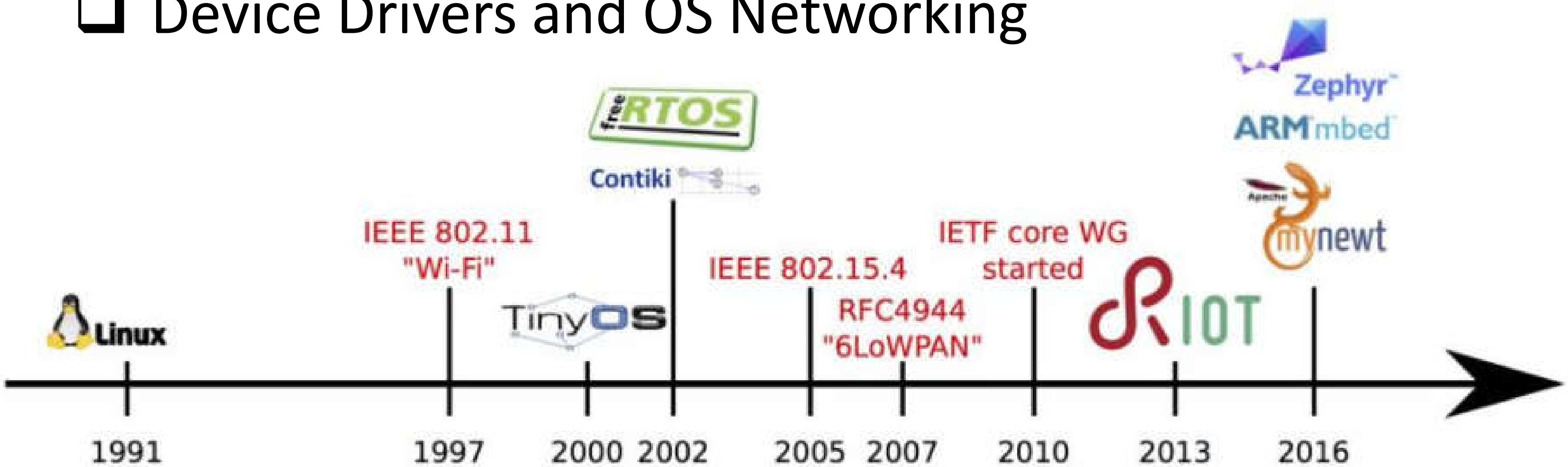
- OS is a software component in computer systems
- OS controls resources of the computer system
  - Allocation of memory for processes
  - power management
- OS coordinates activities in the computer system
  - which process uses the CPU
  - when I/O takes place
- Application programs run on top of OS Services
- Many criteria to classify OS
  - Design goal: general purpose Design goal: general purpose vs OS for specific purposes
  - Target platforms: imitation of resources, reliability guarantees



General purpose OS

- ❑ Design principles and Concepts
- ❑ Execution model
- ❑ Memory management
- ❑ Multitasking and concurrency
- ❑ Support for File systems
- ❑ Safety and Security Features
- ❑ GUI, I/O System, Portability
- ❑ Device Drivers and OS Networking



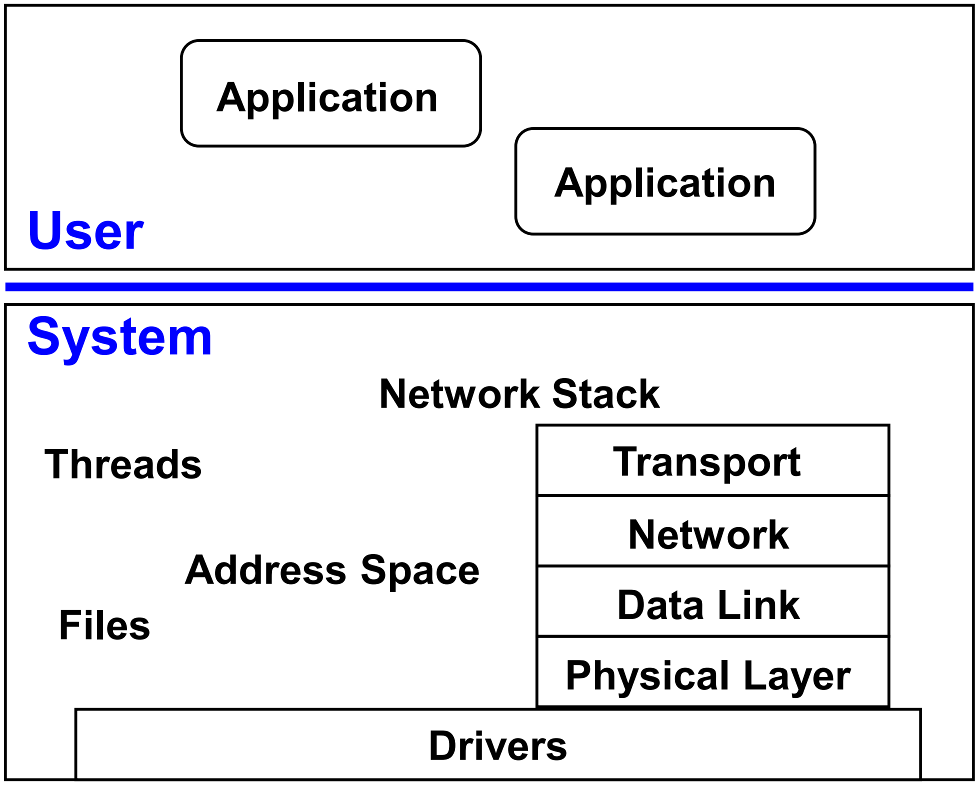


Why OS for WSNs?

- Nodes are designed to operate with limited resources
  - Power: WSN use batteries as a power supply
  - Memory and operational capabilities: sensing is less resource demanding than computation in conventional OS
- Sensor networks are often designed for reliable real time services
  - Additional limitation towards some characteristics of conventional OS
- Power management in WSN OS is very important
- Variety of solutions for WSN
  - Several WSN OS: TinyOS, Contiki OS, etc
  - Several Simulator tools: NS-2, TOSSIM, etc.
  - Several programming languages: C, nesC, etc.
- TinyOS and Contiki-OS: the most popular WSN OS

Traditional Systems

- ❑ Well established layers of abstractions
- ❑ Strict boundaries
- ❑ Ample resources
- ❑ Independent Applications at endpoints communicate P2P through routers
- ❑ Well attended



by comparison, WSNs ...

- Highly Constrained resources
  - processing, storage, bandwidth, power
- Applications spread over many small nodes
  - self-organizing Collectives
  - highly integrated with changing environment and network
  - communication is fundamental
- Concurrency intensive in bursts
  - streams of sensor data and network traffic
- Robust
  - inaccessible, critical operation
- Unclear where the boundaries belong
  - even HW/SW will move

=> Provide a framework for:

• Resource-constrained concurrency

• Defining boundaries

• Application-specific processing and power management

allow abstractions to emerge

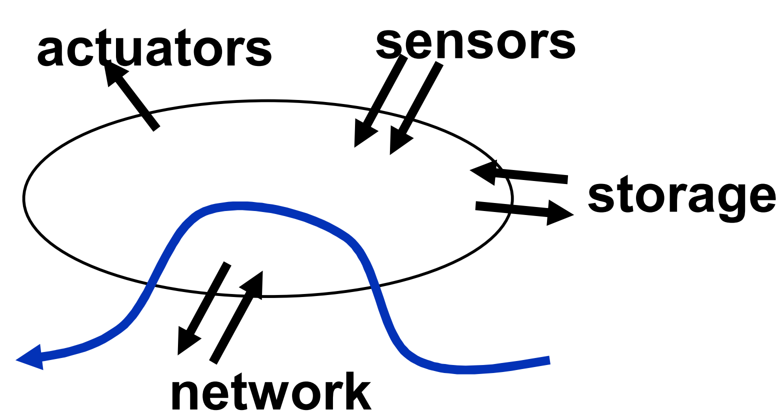
Characteristics of Network Sensors

- ❑ Small physical size and low power consumption
- ❑ Concurrency-intensive operation
  - ❖ multiple flows, not wait-command-respond
- ❑ Limited Physical Parallelism and Controller Hierarchy
  - ❖ primitive direct-to-device interface
- ❑ Asynchronous and synchronous devices
- ❑ Diversity in Design and Usage
  - ❖ application specific, not general purpose
  - ❖ huge device variation

  => efficient modularity

  => migration across HW/SW boundary
- ❑ Robust Operation
  - ❖ numerous, unattended, critical

  => narrow interfaces



Classical RTOS approaches

- ❑ Responsiveness
  - => Provide some form of user-specified interrupt handler
    - User threads in kernel, user-level interrupts
  - ❖ Guarantees?
- ❑ Deadlines / Controlled Scheduling
  - ❖ Static set of tasks with pre-specified constraints
    - Generate overall schedule: => Doesn't deal with unpredictable events, especially communication
  - ❖ Threads + synchronization operations: => Complex scheduler to coerce into meeting constraints
    - Priorities, earliest deadline first, rate monotonic
    - Priority inversion, load shedding, live lock, deadlock
  - ❖ Sophisticated mutex and signal operations
- ❑ Communication among parallel entities
  - ❖ Shared (global) variables: ultimate unstructured programming
  - ❖ Mail boxes (msg passing): => external communication considered harmful
  - ❖ Fold in as RPC
- ❑ Requires multiple (sparse) stacks
- ❑ Preemption or yield

Alternative Starting Points

- Event-driven models
  - Easy to schedule handfuls of small, roughly uniform things
    - State transitions (but what storage and communication model?)
  - Usually results in brittle monolithic dispatch structures
- Structured event-driven models
  - Logical chunks of computation and state that service events via execution of internal threads
- Threaded Abstract machine
  - Developed as compilation target of inherently parallel languages
    - vast dynamic parallelism
    - Hide long-latency operations
  - Simple two-level scheduling hierarchy
  - Dynamic tree of code- block activations with internal inlets and threads
- Active Messages
  - Both parties in communication know format of the message
  - Fine-grain dispatch and consume without parsing
- Concurrent Data-structures
  - Non-blocking, lock-free

Concurrency is tricky

- ❑ Event-driven vs multi-threaded
- ❑ Event-driven (e.g., TinyOS)
  - ❖ Compact, low context switching overhead, fits well for reactive systems
  - ❖ Not suitable for e.g., long running computations: Public/private key cryptography
- ❑ Multi-threading
  - ❖ Suitable for long running computations
  - ❖ Requires more resources

Contiki OS

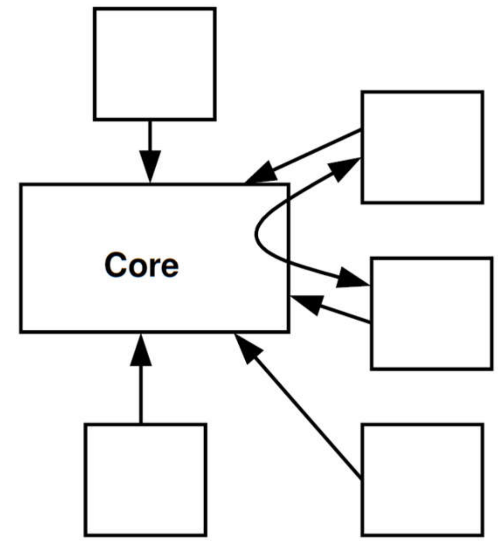
Contiki

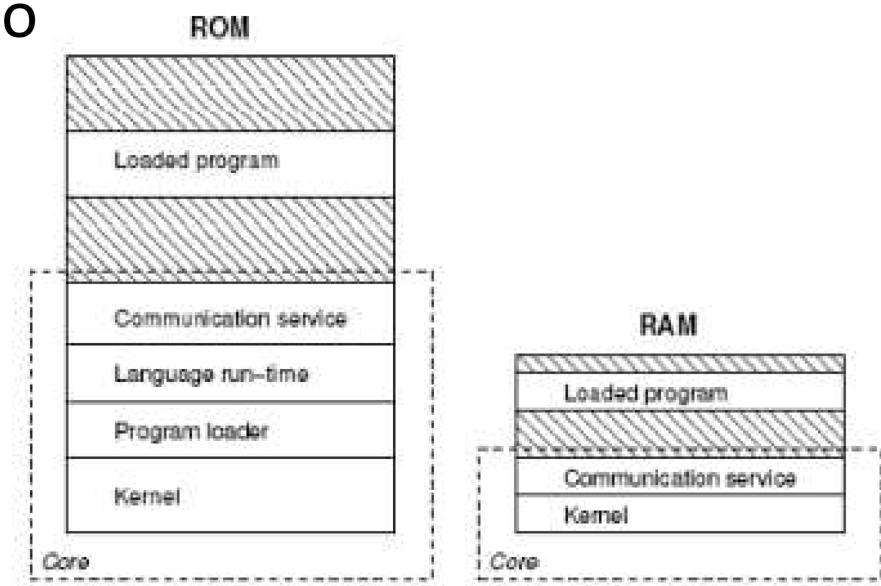
The Open Source OS for the Internet of Things

- Architectures: 8-bit, 16-bit, 32-bit
- Open Source (source code openly available)
- IPv4/IPv6/Rime networking
- Devices with < 8KB RAM
- Typical applications < 50KB Flash
- Vendor and platform independent
- C language
- Developed and contributed by Universities, Research centers and industry

Contiki OS

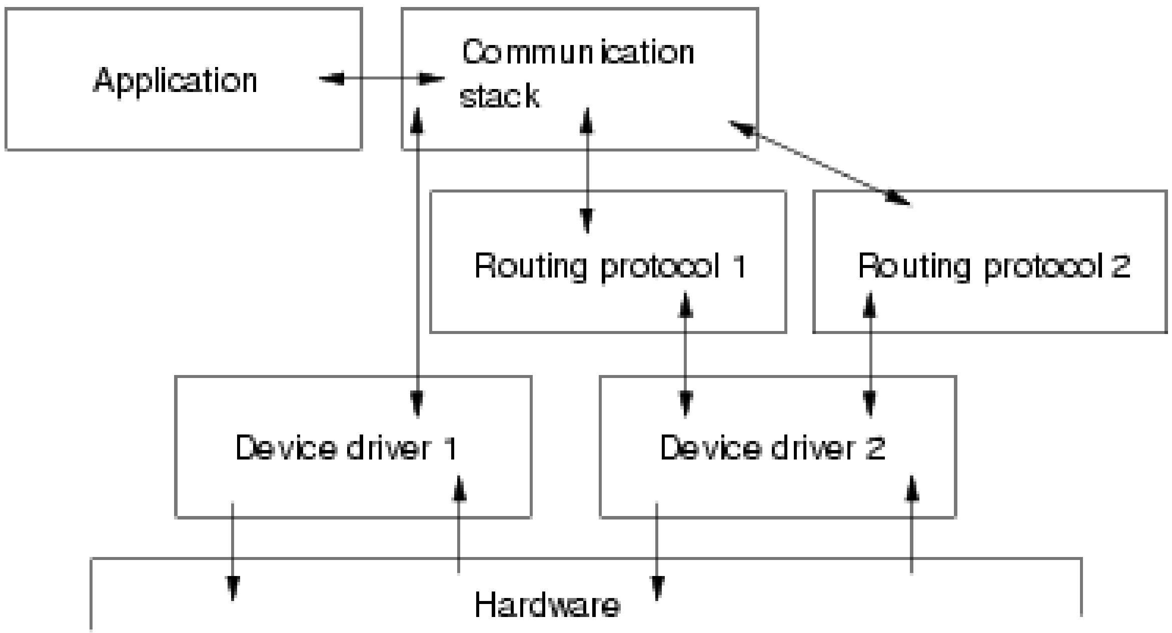
- Contiki is designed to address four essential demand
  - Need for a lightweight OS
  - Dynamic loading/ re-programming in a resource constrained environment
  - Event-driven kernel model desired
  - Optional Protothread/ Preemptive multithreading
- A Contiki system is partitioned into core and loaded programs
  - Processes
    - Application programs
    - Services
  - Core
    - Kernel
    - Program loader
    - Libraries





Contiki Goals

- Portability
- Flexibility
  - Loadable application programs, device drivers
- Multitasking
- Networking (TCP/IP)
  - Event-driven interfaces
- Small size

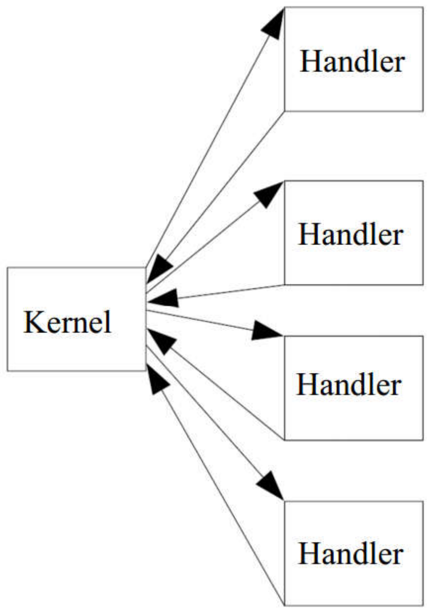


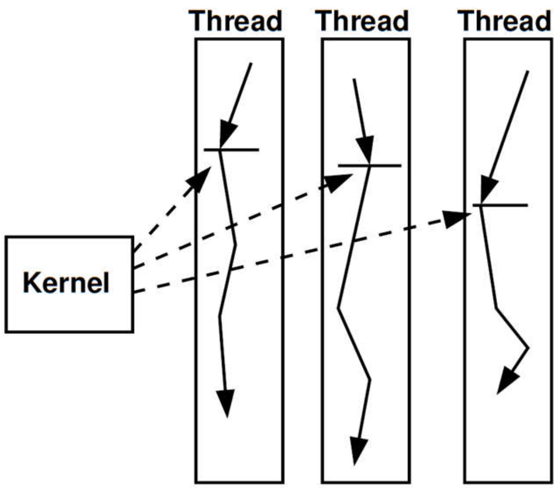
Contiki Execution model

- Lightweight event scheduler
  - Dispatches events to running processes
  - Periodically call processes’ polling handlers
- Triggering of program execution
  - Events dispatched by the kernel
  - Through a polling mechanism
- Threads are driven by events
  - Real threads can be used if needed
  - There is a memory problem however

Contiki Concurrency

- Management mechanism:
  - Events *and* threads/process
  - Trade-offs: pre-emption, size
- Synchronous and asynchronous events
  - Synchronous events: like function calls
  - Asynchronous events: like posting new events
- Events can not preempt each other
  - The kernel does not pre-empt an event handler
  - They can only be pre-empted by interrupts
- Complete multithreaded concurrency possible
  - Implemented as an optional library



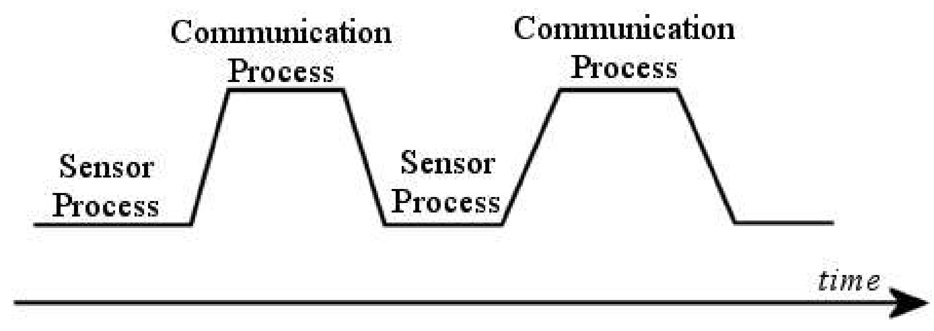


Concurrency

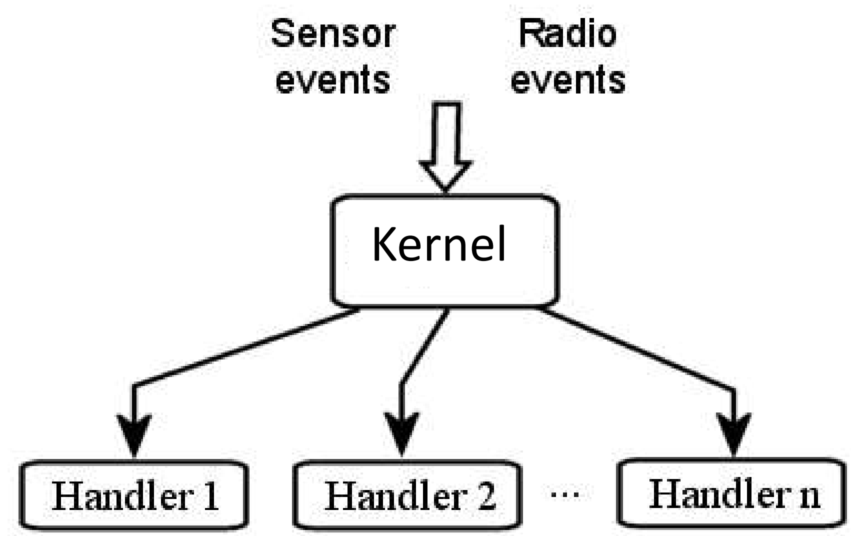
- Concurrency occurs when two or more execution flows run simultaneously
- It introduces many problems such as:
  - Race conditions from shared resources
  - Deadlock and starvation
- OS needs to coordinate between tasks
  - Data exchange, memory, execution, resources
- There are two main techniques: Process-based and Event-based

Concurrency

- Process-based
  - CPU time split between execution tasks
  - Embedded systems typically use lighter threads



- Event-based:
  - Processes do not run without events
  - Event occurs: kernel invokes event handler
  - Event handler runs to completion (explicit return;



Dynamic linking

- ❑ Dynamically download code at run-time
  - ❖ Ability to load and unload applications and services
- ❑ Dynamically linked – replaced at run-time

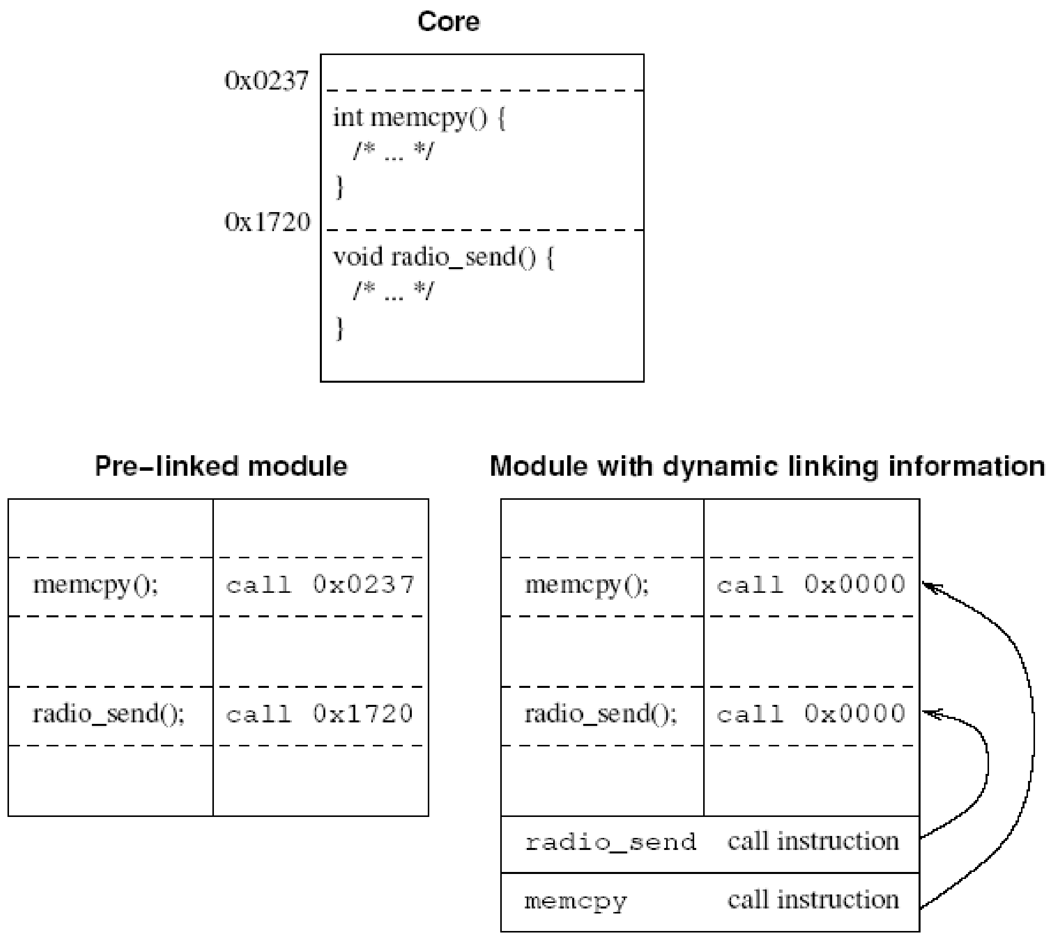
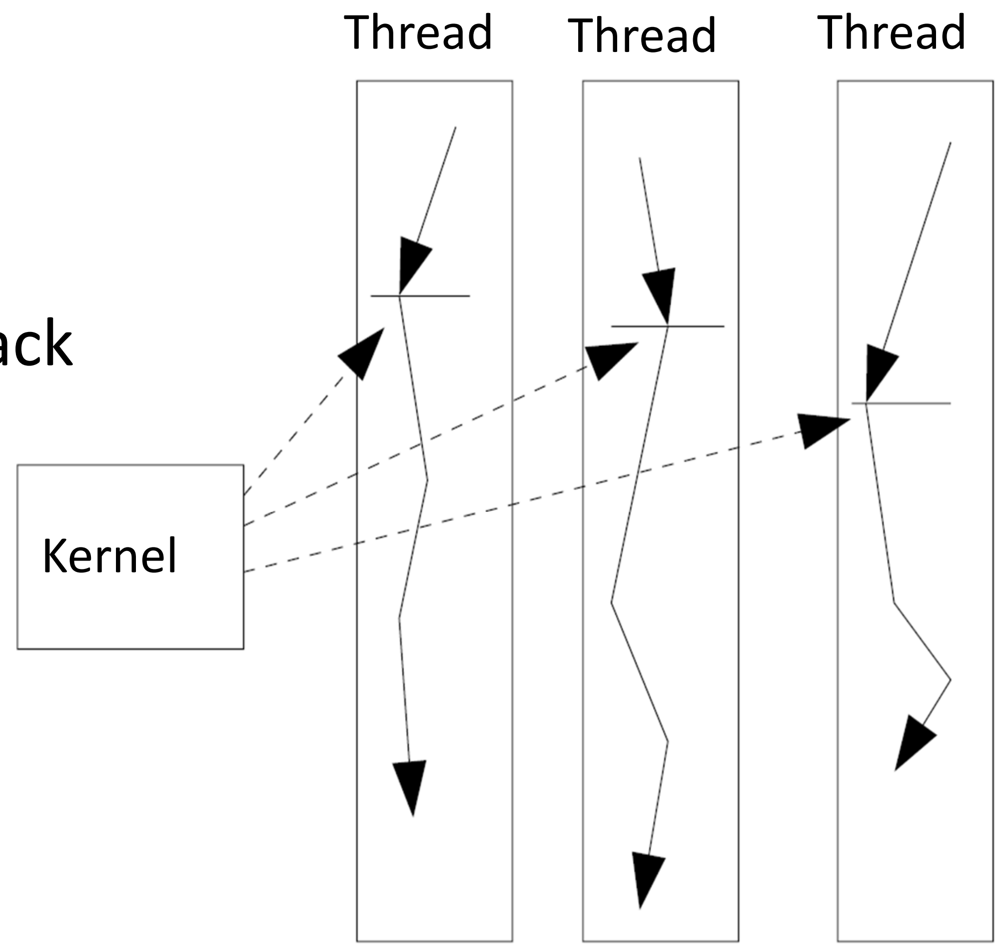


Figure 1. The difference between a pre-linked module and a module with dynamic linking information: the pre-linked module contains physical addresses whereas the dynamically linked module contains symbolic names.

Multi-threaded computation

- ❑ Threads blocked, waiting for events
- ❑ Kernel **unblocks** threads when event occurs
- ❑ Thread runs until next blocking statement
- ❑ Each thread requires its own stack
  - ❖ Larger memory usage



Contiki Memory management

- ❑ Middle ground approach
  - ❖ Modules use static memory almost exclusively
  - ❖ Modules allocate space dynamically for each of its modules variables, when loading
- ❑ No virtual memory
- ❑ No support for protection mechanisms
  - ❖ No memory protection between applications
  - ❖ Single shared stack except for threads
  - ❖ Each thread with a separate stack

Event-driven vs multi-threaded

Event-driven	Multi-threaded
- No `wait()` statements	+ `wait()` statements
- No preemption	+ Preemption possible
- State machines	+ Sequential code flow
+ Compact code	- Larger code overhead
+ Locking less of a problem	- Locking problematic
+ Memory efficient	- Larger memory requirements

Why don't we try to combine them?

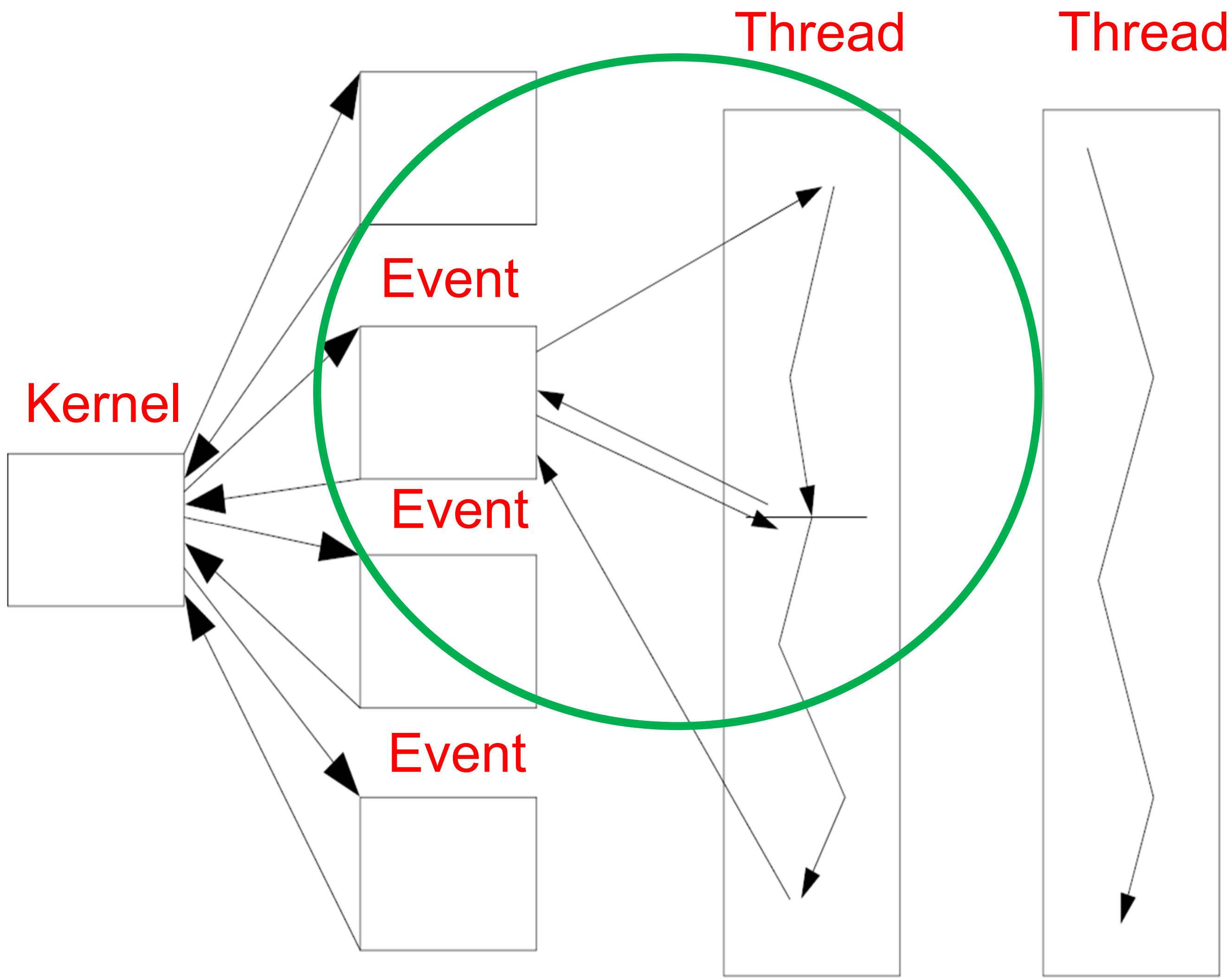
More aspects of Contiki

- Communication
  - Inter process communication via event posting – Communication implemented as a service
- Peripherals
  - C functions used for communication with hardware
  - No general-purpose support for implementing either shared or virtualized services
- Portability
  - Highly portable: including TI MSP430 and the Atmel AVR

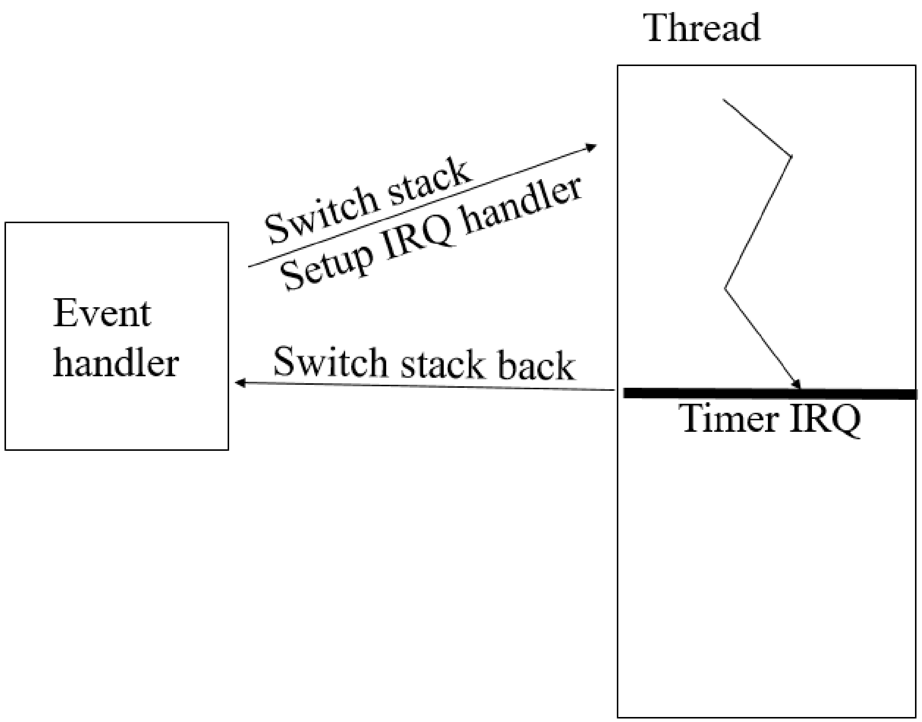
Contiki: event-based kernel with threads

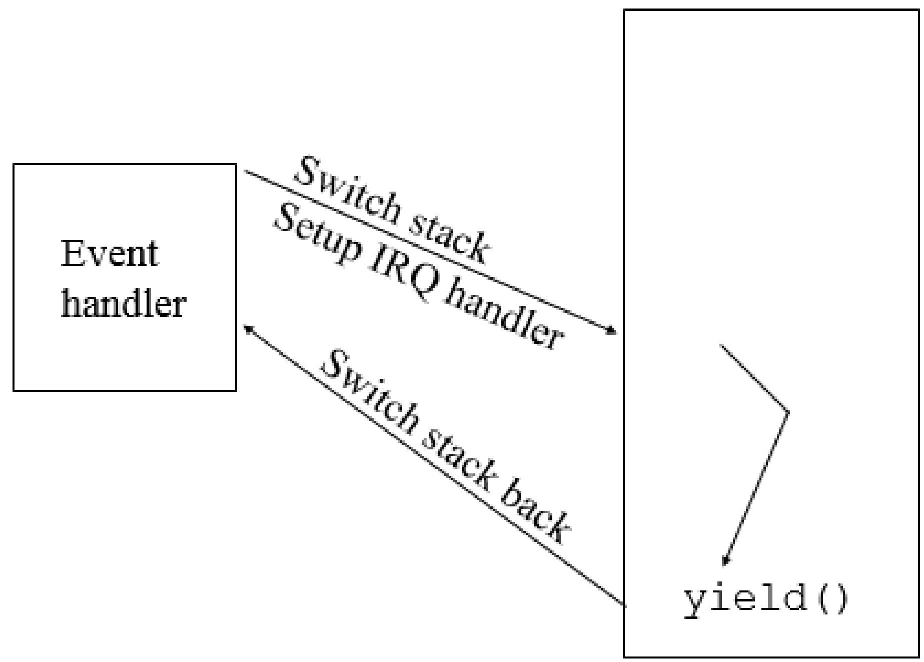
- **Kernel** is **event-based**
  - Most programs run directly on top of the kernel
- **Multi-threading** implemented as a library
- Threads only used if **explicitly** needed
  - Long running computations, ...
- **Pre-emption** possible
  - Responsive system with running computations

Threads implemented top of an Event-based kernel



Thread preemption





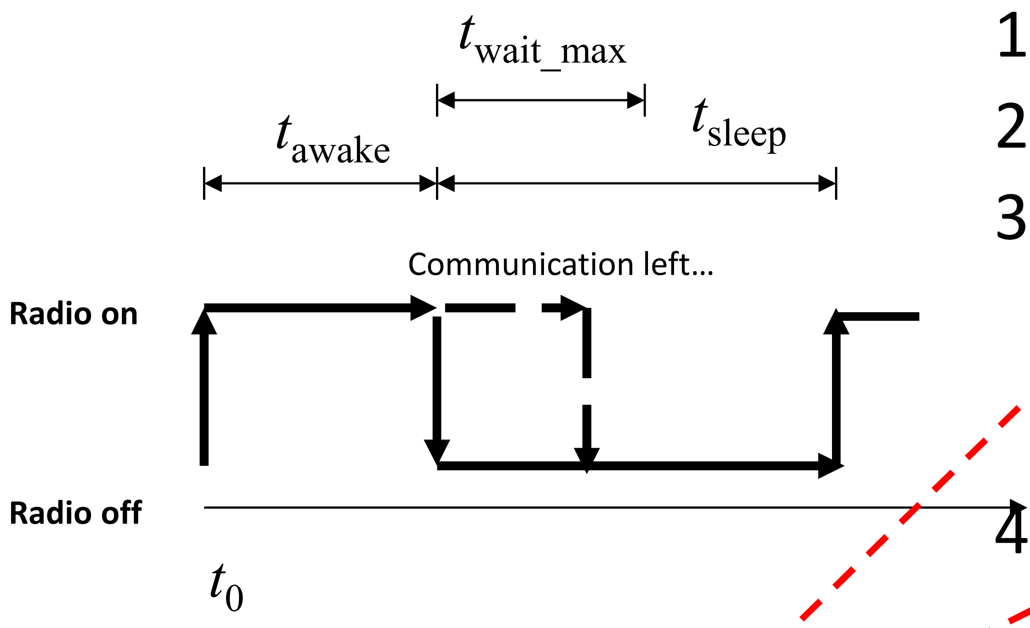
Implementing preemptive threads 1 Implementing preemptive threads 2

Protothreads

- Protothreads – a new programming abstraction
  - For memory-constrained embedded systems
  - A design point between events and threads
  - Very simple, yet powerful idea
- Programming primitive: allow conditional blocking wait
  - PT_WAIT_UNTIL(*condition*)
  - Sequential flow of control
    - Programming language helps us: **if** and **while**
- Protothreads run on a single stack, like the event-driven model
  - Memory requirements (almost) same as for event-driven

- *Note: Contiki processes are **protothreads***

Five-step specification



1. Turn radio on.
2. Wait until $t = t_0 + t_{awake}$.
3. If communication has not completed, wait until it has completed or $t = t_0 + t_{awake} + t_{wait\_max}$.
4. Turn the radio off. Wait until $t = t_0 + t_{awake} + t_{sleep}$.
5. Repeat from step 1.

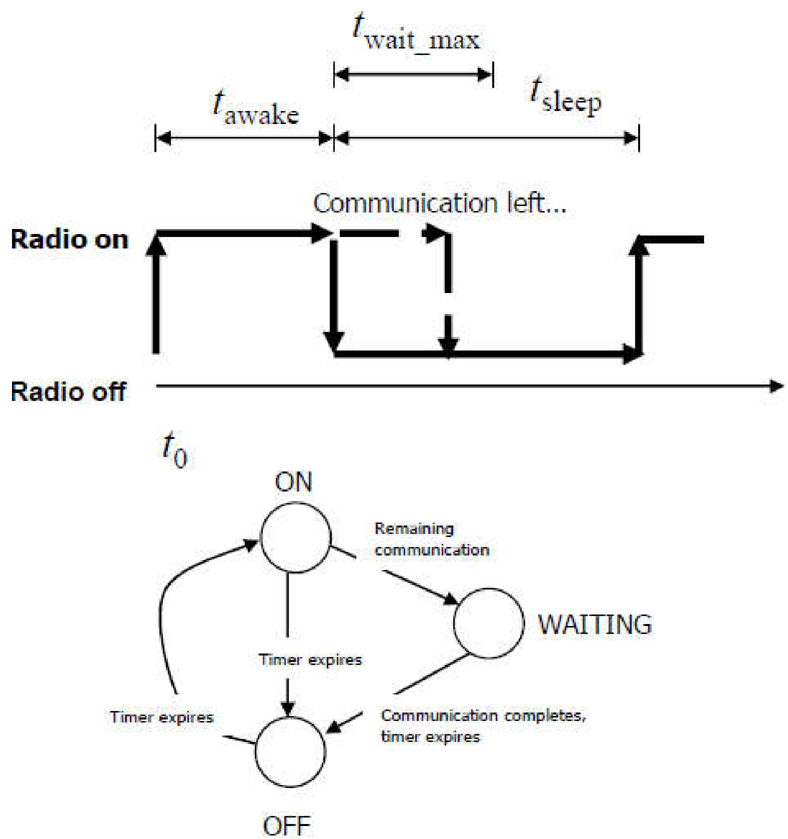
No blocking wait!

Problem: with events, we cannot implement this as a five-step program!

Source: Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. 2006. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In Proceedings of the 4th international conference on Embedded networked sensor systems (SenSys '06). ACM.

Radio sleep cycle code with events

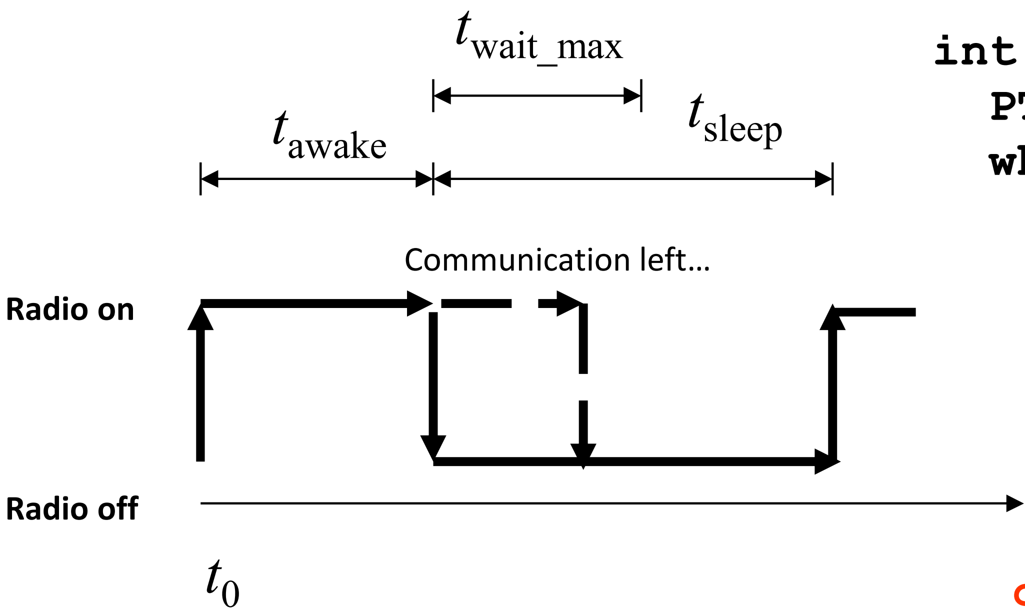
Event-driven code can be messy and complex




```
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;
    }
  }
}
```

Source: Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. 2006. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In Proceedings of the 4th international conference on Embedded networked sensor systems (SenSys '06). ACM.

Radio sleep cycle with Protothreads

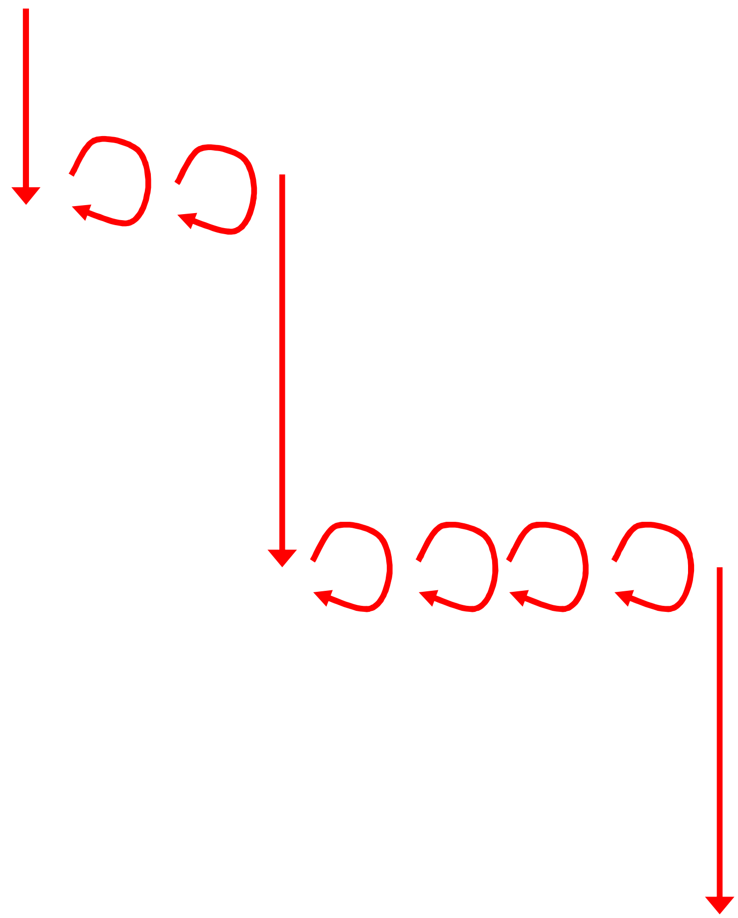



```
int protothread(struct pt *pt) {
  PT_BEGIN(pt);
  while(1) {
    radio_on();
    timer = t_awake;
    PT_WAIT_UNTIL(pt, expired(timer));
    timer = t_sleep;
    if(!comm_complete()) {
      wait_timer = t_wait_max;
      PT_WAIT_UNTIL(pt,
 comm_complete()
               ||
 expired(wait_timer));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

- Protothreads – conditional blocking wait: **`PT_WAIT_UNTIL()`**
- No need for an explicit state machine
- Sequential code flow

An example protothread

```
int a_protothread(struct pt *pt) {
  PT_BEGIN(pt);

  /* … */

  PT_WAIT_UNTIL(pt, condition1);

  /* … */

  if(something) {

    /* … */

    PT_WAIT_UNTIL(pt, condition2);

    /* … */

  }

  PT_END(pt);
}
```

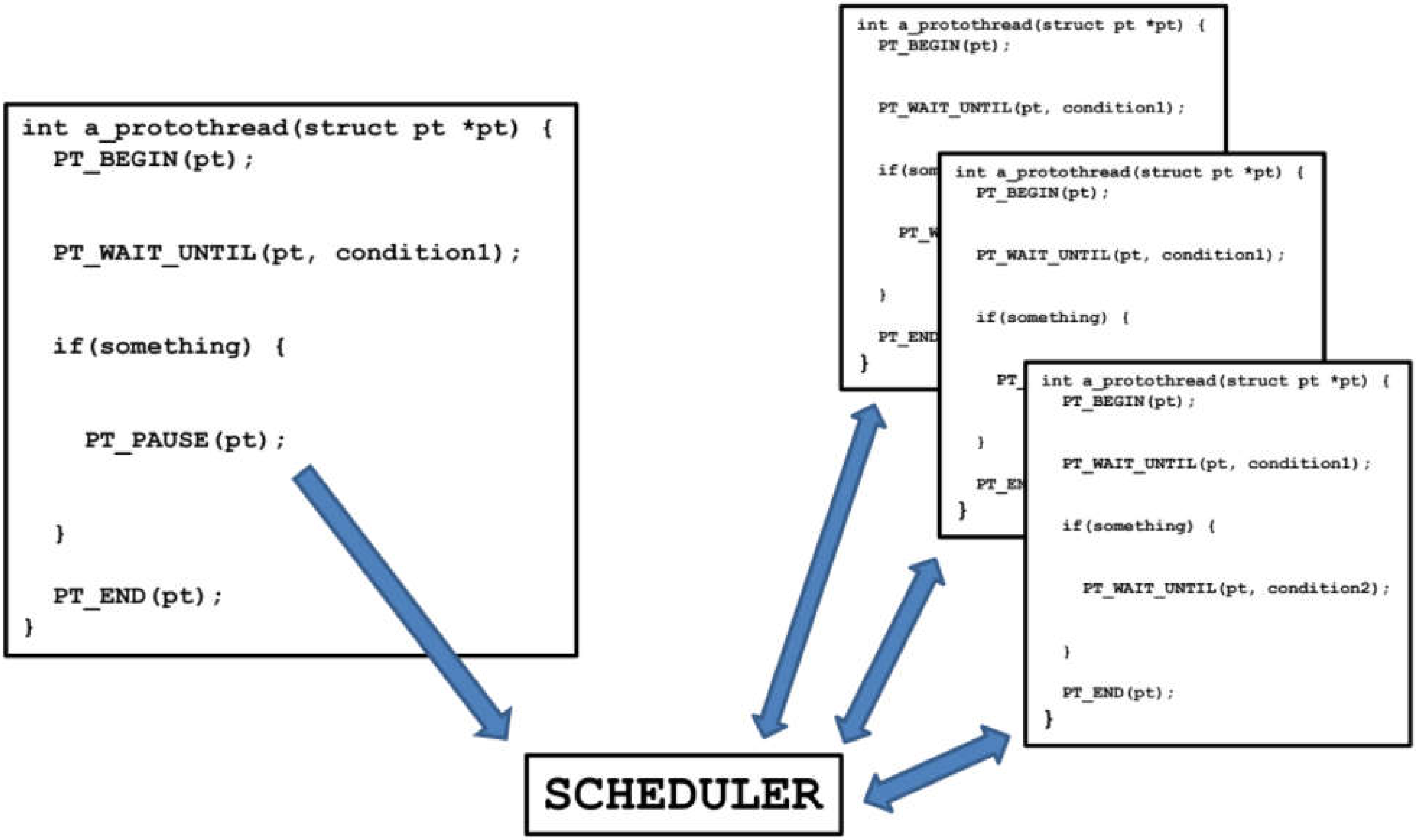


Hierarchical protothreads

```
int a_protothread(struct pt *pt) {
  static struct pt child_pt;

  PT_BEGIN(pt);

  PT_INIT(&child_pt);
  PT_WAIT_UNTIL(pt2(&child_pt) != 0);

  PT_END(pt);
}
```

```
int pt2(struct pt *pt) {
  PT_BEGIN(pt);

  PT_WAIT_UNTIL(pt, condition);

  PT_END(pt);
}
```

Prototheads: Yielding



Contiki Power management

- No standard mechanisms for managing the power state of peripheral devices state of peripheral devices
- Power optimizations
  - Microcontroller in a sleep mode
  - Power estimation as additional feature

Multi-Platform Build System

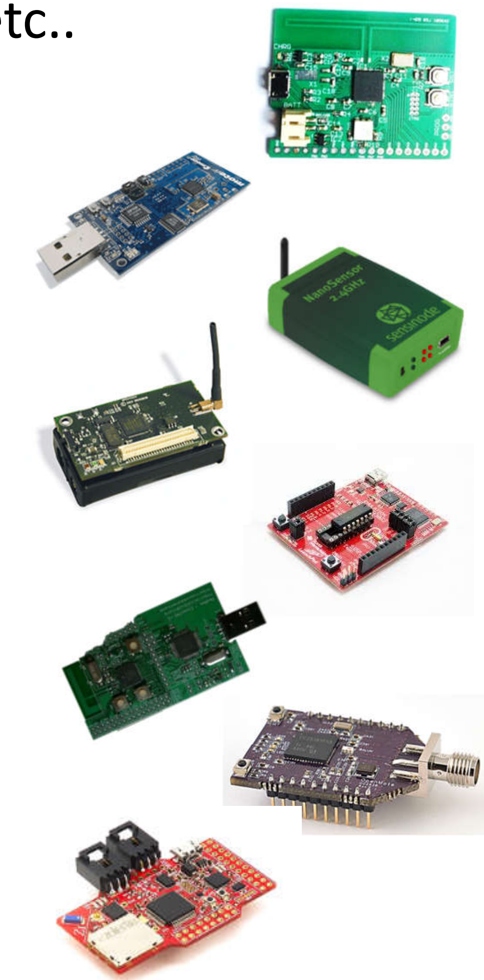
- Support: Imote2, MicaZ, TMote Sky, Z1, CC2538, CC2650, etc..
- Other porting: CC2530, nRF (Nordic), etc…

User App → **Platform** → **Contiki**

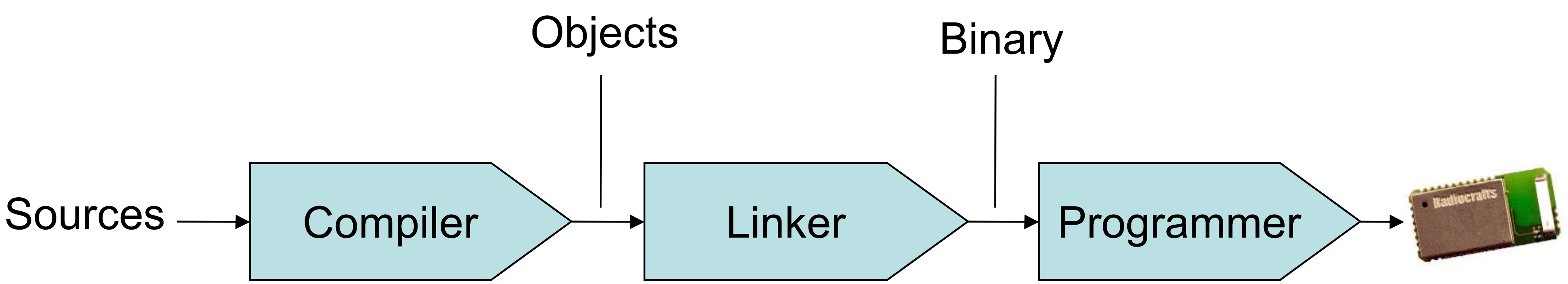
App-specific conf → Platform-specific conf → Default Conf

Contiki Modules & Applications

RPL	Coffee	Shell	Profiling	…



Embedded Development



Embedded Development

- Software resides in system non-volatile memory
  - External or internal flash/EEPROM/PROM memories
  - Modern microcontrollers are capable of writing the internal flash memory run-time and often do not have an external memory bus
- Development is done outside the system
- Cross-compilers are used to create binary files
  - Cross-compiler creates binary files for a different architecture than it is running on
  - Various commercial and free compilers available
- System is programmed by uploading the binary
  - In-system programming tools or external flashers

Cross-compiler Environments

- Integrated development environments (IDEs)
  - Commercial compilers are usually of this type
  - Usually dependent on a specific OS (Windows)
  - Integrate a text editor, compiler tools and project management along with C library
  - System programmer tool usually tightly integrated
  - Also open-source IDEs available
    - Open-source IDEs usually employ "plugin" architecture
    - General-purpose extensible environments
    - Include scripting tools for running any command line tools: compilers, linkers, external editors and programmers
    - Example: Eclipse (implemented in Java)

Cross-compiler Environments

- Command line utilities
  - Separate compiler/linker, editor and project management tools, architecture-dependent C library
- Project management: make
  - make is an automated software building tool
  - Based on target-dependency-operation style blocks
  - Allows use of project templates and separate platform build rules by using “include files”
  - Most common way of managing open-source software projects
  - *automake* and *autoconf* tools extend functionality to platform-independent software development

Cross-compiler Environments

- Command line compilers
  - most common is gcc: available for a multitude of microcontroller and -processor architectures
  - ***sdcc***: Small Device C Compiler: PICs, 8051's etc.
  - single-architecture compilers
- System programming tools
  - usually specific to a single microcontroller family
  - vary greatly in their ease of use and interface type
  - most require some sort of programming cable or a programmer device to upload software
  - dependent on the microcontroller programming algorithm
    - standard buses (SPI, UART, JTAG) vs. proprietary buses

Cross-compiler Environments

- Command-line tools vs. (commercial) IDEs
  - IDEs are easily accessible: single installer, single GUI
  - Commercial IDEs vary greatly in usability, standards compliance and are (usually) tied to a single architecture -> bad portability
  - Most commercial IDEs don't really support templates
    - Programmer must go through various dialogs to create a new project
    - Often project files can not just be copied (contain directory paths and such) and may be binary format
  - Command line tools have a steeper learning curve
    - Once learned, applicable to most architectures
    - Higher flexibility and ease of duplicating projects

Cross-compiler Issues

- Portability
  - Header files may not follow standard naming
  - Hardware-specific header files might not be automatically selected
    - Most commercial IDEs use different names for each different hardware model -> difficulties in portability
    - gcc e.g. uses internal macros for model selection -> easier portability via environment variables, no header changes
- Hardware register access and interrupt handlers
  - Interrupt handler declaration is compiler-dependent
    - Declaration format is not standardized
    - Can be worked around via macros (in most cases)
  - Some compilers (and C libraries) require I/O macros
    - gcc ports implement direct register access modes

Open-source Tools

- Various text editors available: *nedit*, *emacs*, *vi* ...
- Project build system: *make*
- Compilers/linkers: *binutils* & *gcc*, *sdcc*
  - *binutils*: as, ld, objcopy, size etc.
  - *gcc*: c compiler; uses binutils to create binary files
- Standard C libraries
  - Provide necessary development headers and object files for linking and memory mapping
  - msp430-libc for MSP430, avr-libc for AVR
- Programmers
  - AVR: uisp, avrdude
  - MSP430: msp430-bsl, msp430-jtag
  - CC2430: nano_programmer
- IDE: Eclipse, Simplicity

SDCC Compiler

- Simple Device C Compiler (http://www.sdcc.org)
- Specialized in 8051, PIC, HC08 etc. microcontrollers
  - Has CC2430, **CC2530** and CC2510 support
- *sdcc* application handles both compilation and linking
- Uses make build environment
- Compatible with Eclipse
- Support for banking (needed in 8051 with 64k+ ROM)
  - Thanks to Peter Kuhar for banking support

Events in Contiki

- **timer events**: a process may set a timer to generate an event after a given time, it will block until the timer expires and then continue its execution. This is useful for periodic actions, or for networking protocols e.g. involving synchronization;
- **external events**: peripheral devices connected to I/O pins of the microcontroller with interrupt capabilities may generate events when triggering interruptions. A push-button, a radio chip or a shock detector accelerometer are a few examples of devices that could generate interruptions, thus events. Processes may wait for such events to react accordingly.
- **internal events**: any process has the possibility to address events to any other process, or itself. This is useful for inter-process communication as informing a process that data is ready for computation.

Events in Contiki

- Events are said **posted**. An interrupt service routine will post an event to a process when it is executed. Events have the following information:
  - **process**: the process addressed by the event. It can be either one specific process or all the registered processes;
  - **event type**: the type of event. The user can define some event types for the processes to differentiate them, such as one when a packet is received, one when a packet is sent;
  - **data**: additionally, some data may be provided along with the event for the process.
- This is the main principle of the Contiki OS: events are posted to processes; these execute when they receive them until they block waiting for another event.

Process in Contiki

- **Processes** are the task-equivalent of Contiki. The process mechanism *uses the underlying protothread library* which in turn uses the local continuation library.

- A process is a C function most likely containing an infinite loop and some blocking macro calls. Since the Contiki event-driven kernel is not preemptive, each process when executed will run until it blocks for an event. Several macros are defined for the different blocking possibilities. This allows programming state-machines as a sequential flow of control.

The Contiki code

```
#include "contiki.h"

PROCESS(sample_process, "My sample process");

AUTOSTART_PROCESSES(&sample_process);

PROCESS_THREAD(sample_process, ev, data) {
        PROCESS_BEGIN();
                while(1) {
                        PROCESS_WAIT_EVENT();
                }
        PROCESS_END();
}
```

Header files

Defines the name of the process

Defines the process will be started every time module is loaded

contains the process code

Event parameter; process can respond to events

Threads must have an end statement

process can receive data during an event

The Contiki code

```
#include "contiki.h"

PROCESS(sample_process, "My sample process");

AUTOSTART_PROCESSES(&sample_process, &LED_process);


PROCESS_THREAD(sample_process, ev, data) {
        static struct etimer t;
        static int c = 0;
          PROCESS_BEGIN();
                  etimer_set(&t, CLOCK_CONF_SECOND);
                   while(1) {
                                PROCESS_WAIT_EVENT();
                                if(ev == PROCESS_EVENT_TIMER)  {
                                             printf("Timer event #%i\n", c);
                                             c++;
                                             etimer_reset(&t);
                                }
                   }
          PROCESS_END();
}

PROCESS_THREAD(LED_process, ev, data) {
        static uint8_t leds_state = 0;
        PROCESS_BEGIN();
                leds_off(0xFF);
                leds_on(leds_state);
        PROCESS_END();
}
```

Process thread names

Process thread 1

Process thread 2

Typedefs

typedef uint8_t	**u8_t** The 8-bit unsigned data type.
typedef uint16_t	**u16_t** The 16-bit unsigned data type.
typedef uint32_t	**u32_t** The 32-bit unsigned data type.
typedef int32_t	**s32_t** The 32-bit signed data type.
typedef unsigned short	**uip_stats_t** The statistics data type.

Hello-world example in Contiki

Make file

```
CONTIKI_PROJECT = hello-world
all: $(CONTIKI_PROJECT)

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

Hello-world example in Contiki

```
/* Declare the process */
PROCESS(hello_world_process, "Hello world");

/* Make the process start when the module is loaded */

AUTOSTART_PROCESSES(&hello_world_process);

/* Define the process code */

PROCESS_THREAD(hello_world_process, ev, data) {

  PROCESS_BEGIN();               /* Must always come first */

  printf("Hello, world!\n");     /* Initialization code goes here */

  while(1) {                     /* Loop for ever */

    PROCESS_WAIT_EVENT();        /* Wait for something to happen */

  }

  PROCESS_END();                 /* Must always come last */

}
```

Running Contiki on a Hardware

- ❑ Write your code
- ❑ Compile Contiki and the application
  - ❖ *make TARGET=XM1000 sample_process*
  - ❖ Make file

```
CONTIKI = ../..
all: simple_process
include $(CONTIKI)/Makefile.include
```

- ❑ If you plan to compile your code on the chosen platform more than once;
  - ❖ *make TARGE=sky savetarget*
- ❑ Upload your code
  - ❖ *make simple_process.upload*
- ❑ Login to the device
  - ❖ *make login*

Contiki events

- `process_post(&process, eventno, evdata);`
  - Process will be invoked later
- `process_post_synch(&process, evno, evdata);`
  - Process will be invoked now
  - Must not be called from an interrupt (device driver)
- `process_poll(&process);`
  - Sends a PROCESS_EVENT_POLL event to the process
  - Can be called from an interrupt

Using events

```
PROCESS_THREAD(rf_test_process, ev, data) {
  while(1) {
    PROCESS_WAIT_EVENT();
    if (ev == EVENT_PRINT) printf("%s", data);
  }
}
```

Contiki timers

- Contiki has two main timer types; *etimer* and *rtimer*
- *Etimer*: generates timed events

  Declarations:

  ```
  static struct etimer et;
  ```

  In main process:

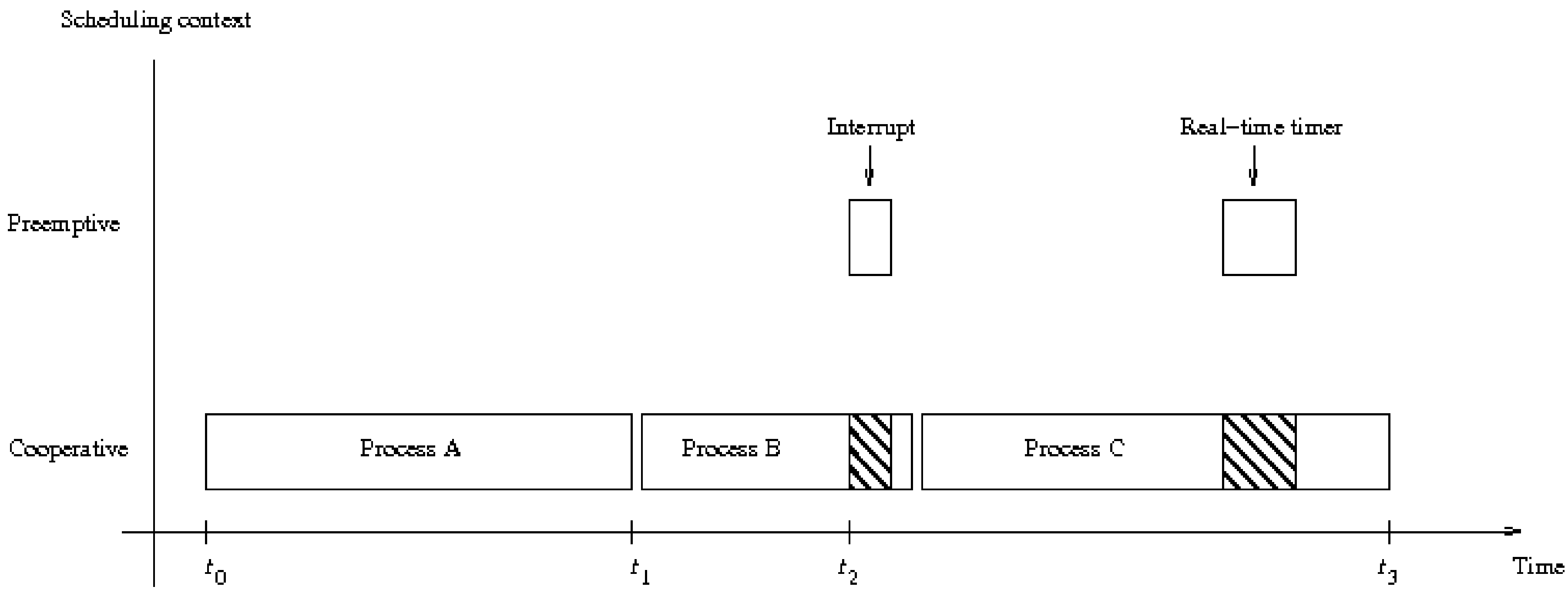
  ```
  while(1) {
      etimer_set(&et, CLOCK_SECOND);
      PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
      etimer_reset(&et);
  }
  ```

- *Rtimer*: uses callback function
  - Callback executed after specified time

  ```
  rtimer_set(&rt, time, 0 , &callback_function, void *argument);
  ```

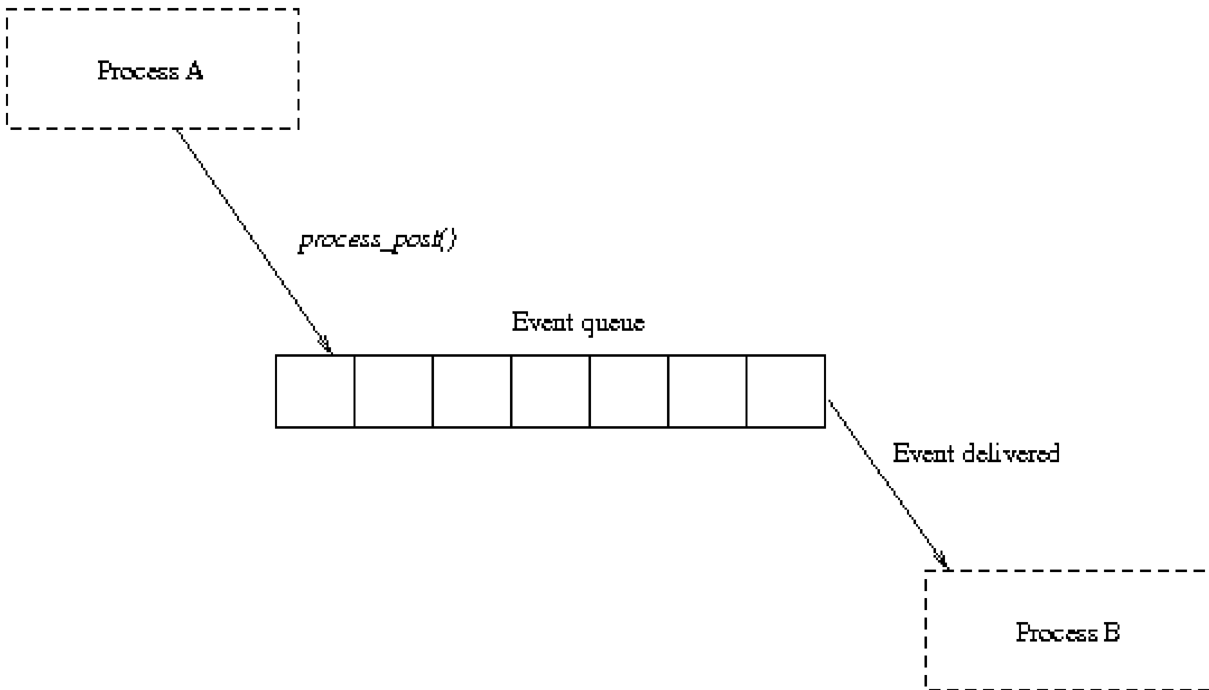
Process and Interrupt

- Code in Contiki runs in either of two execution contexts: cooperative or preemptive. Cooperative code runs sequentially with respect to other cooperative code. Preemptive code temporarily stops the cooperative code. Contiki processes run in the cooperative context, whereas interrupts and real-time timers run in the preemptive context



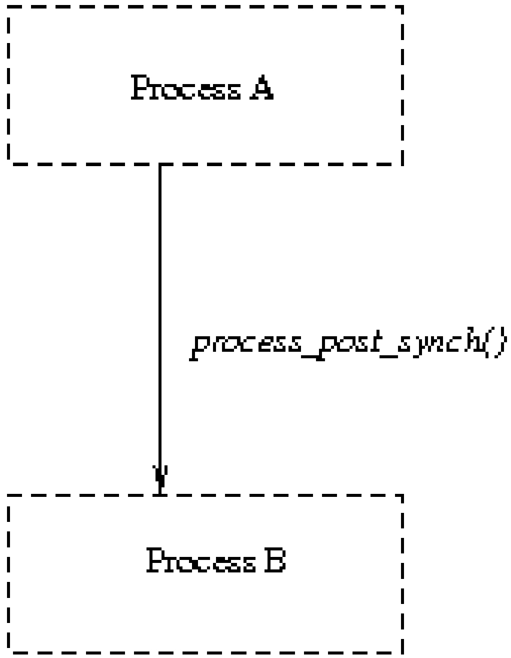
Async and Sync Events

- In Contiki, a process is run when it receives an event. There are two types of events: asynchronous events and synchronous events.
- When an *asynchronous events* is posted, the event is put on the kernel's event queue and *delivered to the receiving process at some later time*.



Async and Sync Events

- When a *synchronous event is posted*, the event is *immediately delivered* to the *receiving process*.
- Because synchronous events are delivered immediately, posting a synchronous event is *functionally equivalent to a function call*: the process to which the event is delivered is directly invoked, and the process that posted the event is blocked until the receiving process has finished processing the event. The receiving process is, however, not informed whether the event was posted synchronously or asynchronously



I/O Interface

- ❑ Connect Relays to PD0, PD1, PD4 và PD5 to CC2538
- ❑ Modify *board.h* in folder *platform/cc2538dk/dev*

board.h | project-conf.h | table.tex

```c
#define RELAY_1                31 //1 // PD0
#define RELAY_2                32 //2 // PD1
#define RELAY_3                33 //1 // PD4
#define RELAY_4                34 //2 // PD5
```

- ❑ Modify *leds-arch.h* in folder *platform/cc2538dk/dev*

leds-arch.c | my_leds.c | my_leds.h | server.c

```c
/*---------------------------------------------------------------------------
void
leds_arch_init(void)
{
  GPIO_SET_OUTPUT(GPIO_C_BASE, LEDS_GPIO_PIN_MASK);
#if TARGET_HAS_CC2592__
    // relay config
  GPIO_SET_OUTPUT(GPIO_D_BASE, 0x33);        // PORT = 7654.3210
  GPIO_WRITE_PIN(GPIO_D_BASE, 0x33, 0x33); // MASK = 0011.0011 = 0x33
```

- ❑ Design API for use

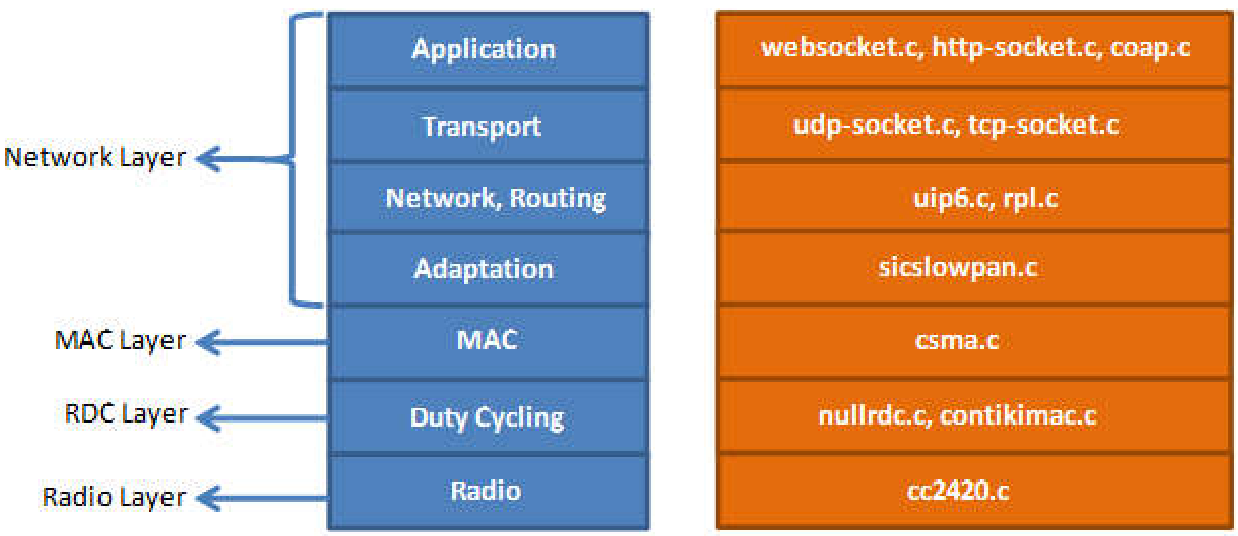
UART interface

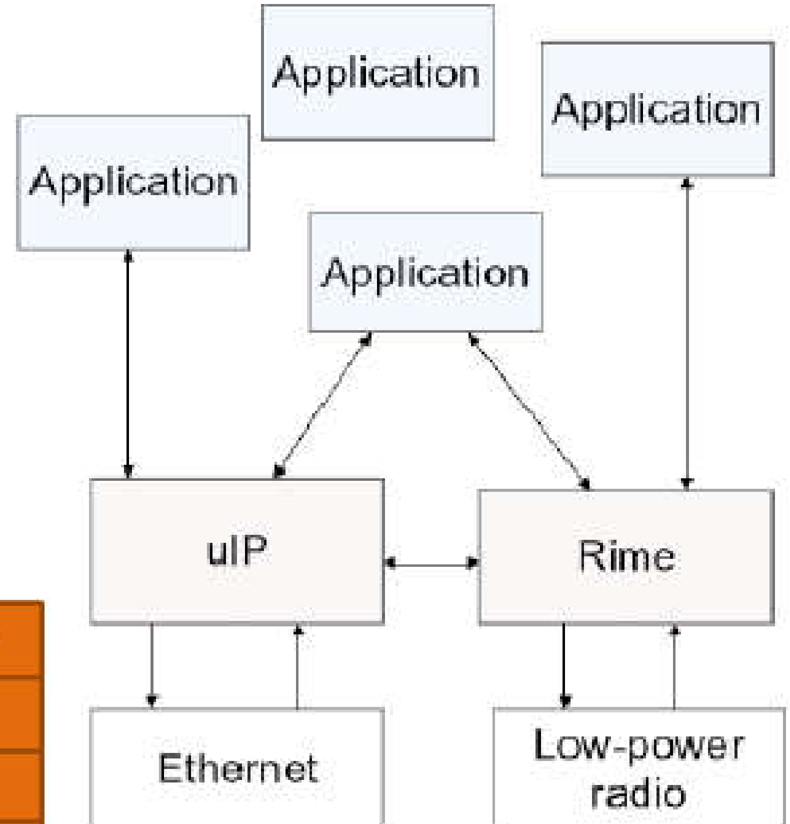
```
#include "dev/uart.h"

static int uart0_input_byte(unsigned char c) {}

static unsigned int uart0_send_bytes(const     unsigned  char *s,
unsigned int len) {
  unsigned int i;
  for (i = 0; i<len; i++)
     uart_write_byte(0, (uint8_t) (*(s+i)));
   return 1;
}

PROCESS_THREAD(udp_echo_server_process, ev, data) {
        PROCESS_BEGIN();

        uart_init(0);
        uart_set_input(0,uart0_input_byte);

        while(1) {
               PROCESS_YIELD();
        }
        PROCESS_END();
}
```

Contiki Protocol Stacks

- Contiki has 2 different protocol stacks: **uIP** and **Rime**
- uIP provides a full TCP/IP stack
  - For interfaces that allow protocol overhead
  - Ethernet devices
  - Serial line IP
  - Includes IPv4 and IPv6/6LoWPAN support
- Rime provides compressed header support
  - Application may use MAC layer only
- Protocol stacks may be interconnected
  - uIP data can be transmitted over Rime and vice versa

Networking Stack: uIP

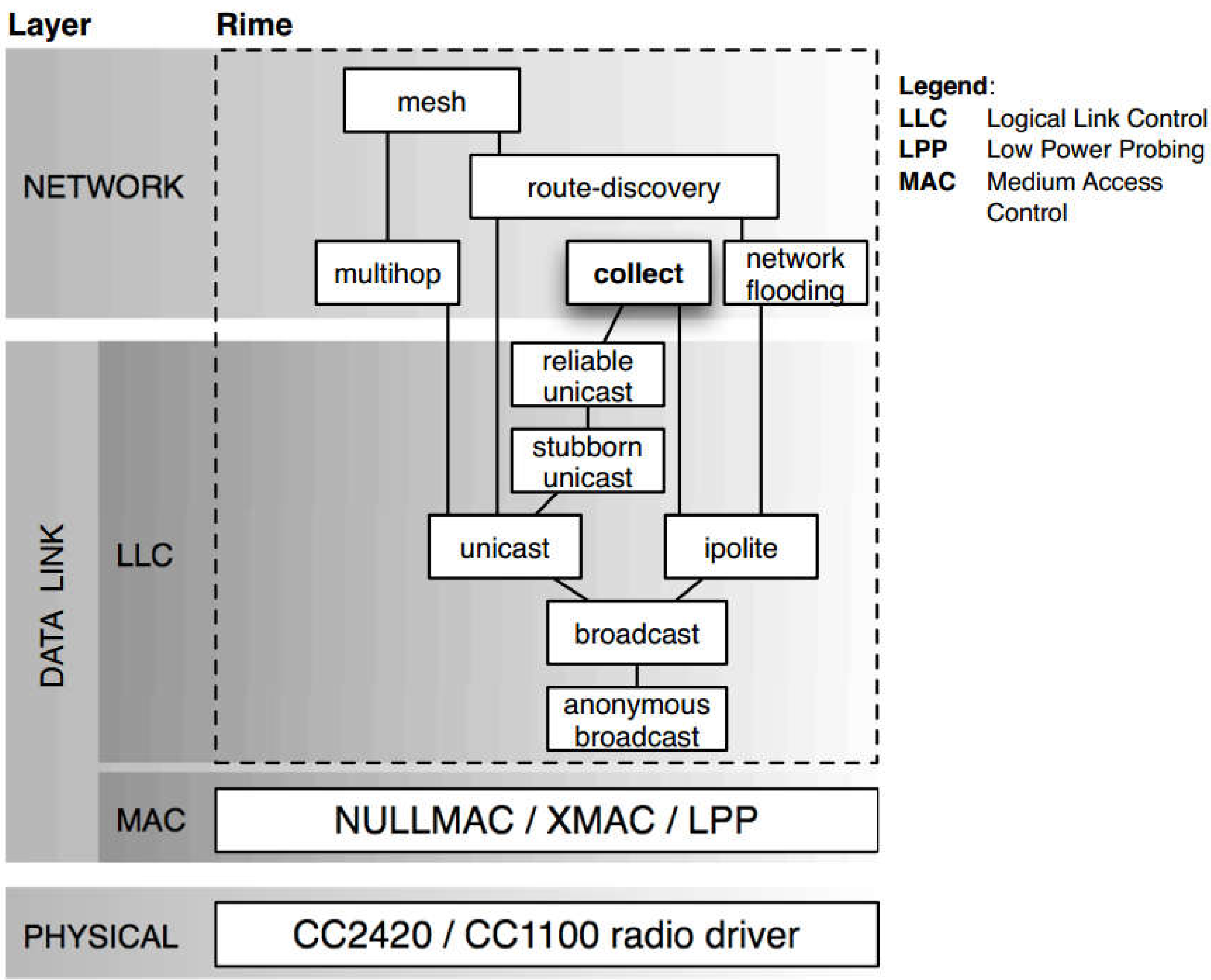




RIME stack

- The Rime communication stack provides a set of basic communication primitives ranging from best-effort single-hop broadcast and best-effort single-hop unicast to best-effort network flooding and hop-by-hop reliable multi-hop unicast. It has been designed to map onto typical sensor network protocols: data dissemination, data collection, and mesh routing.
- The Rime stack builds on top of the physical layer and the MAC layer. The physical layer is handled by the radio driver

RIME Stack



Layer
Rime
NETWORK
mesh
route-discovery
multihop
collect
network flooding
DATA LINK
LLC
reliable unicast
stubborn unicast
unicast
ipolite
broadcast
anonymous broadcast
MAC
NULLMAC / XMAC / LPP
PHYSICAL
CC2420 / CC1100 radio driver
Legend:
LLC Logical Link Control
LPP Low Power Probing
MAC Medium Access Control

The Rime protocol stack

- Separate modules for protocol parsing and state machines
  - Rime contains the protocol operation modules
  - Chameleon contains protocol parsing modules
- Rime startup: an example
  - Configure Rime to use *sicslowmac* over cc2430 rf
  - Startup is done in platform main function: platform/sensinode/contiki-sensinode-main.c

```
rime_init(sicslowmac_init(&cc2430_rf_driver));
set_rime_addr(); //this function reads MAC from flash and places
                 //it to Rime address
```

Rime: Receiving

- Setting up Rime receiving: broadcast
  - Set up a call-back function

Declarations:

```
static struct broadcast_conn bc;
static const struct broadcast_callbacks broadcast_callbacks =
    {recv_bc};
```

The call-back definition:

```
static void
recv_bc(struct broadcast_conn *c, rimeaddr_t *from);
```

In main process:

```
broadcast_open(&bc, 128, &broadcast_callbacks);
```

- Unicast receive in a similar manner

Rime: Sending

❑ Sending broadcast data using Rime

Declarations:

```
static struct broadcast_conn bc;
```

In main process:

```
packetbuf_copyfrom("Hello everyone", 14);
broadcast_send(&bc);
```

❑ Sending unicast data using Rime

Declarations:

```
static struct unicast_conn uc;
```

In your function:

```
rimeaddr_t *addr;
addr.u8[0] = first_address_byte;
addr.u8[1] = second_address_byte;
packetbuf_copyfrom("Hello you", 9);
unicast_send(&uc, &addr);
```

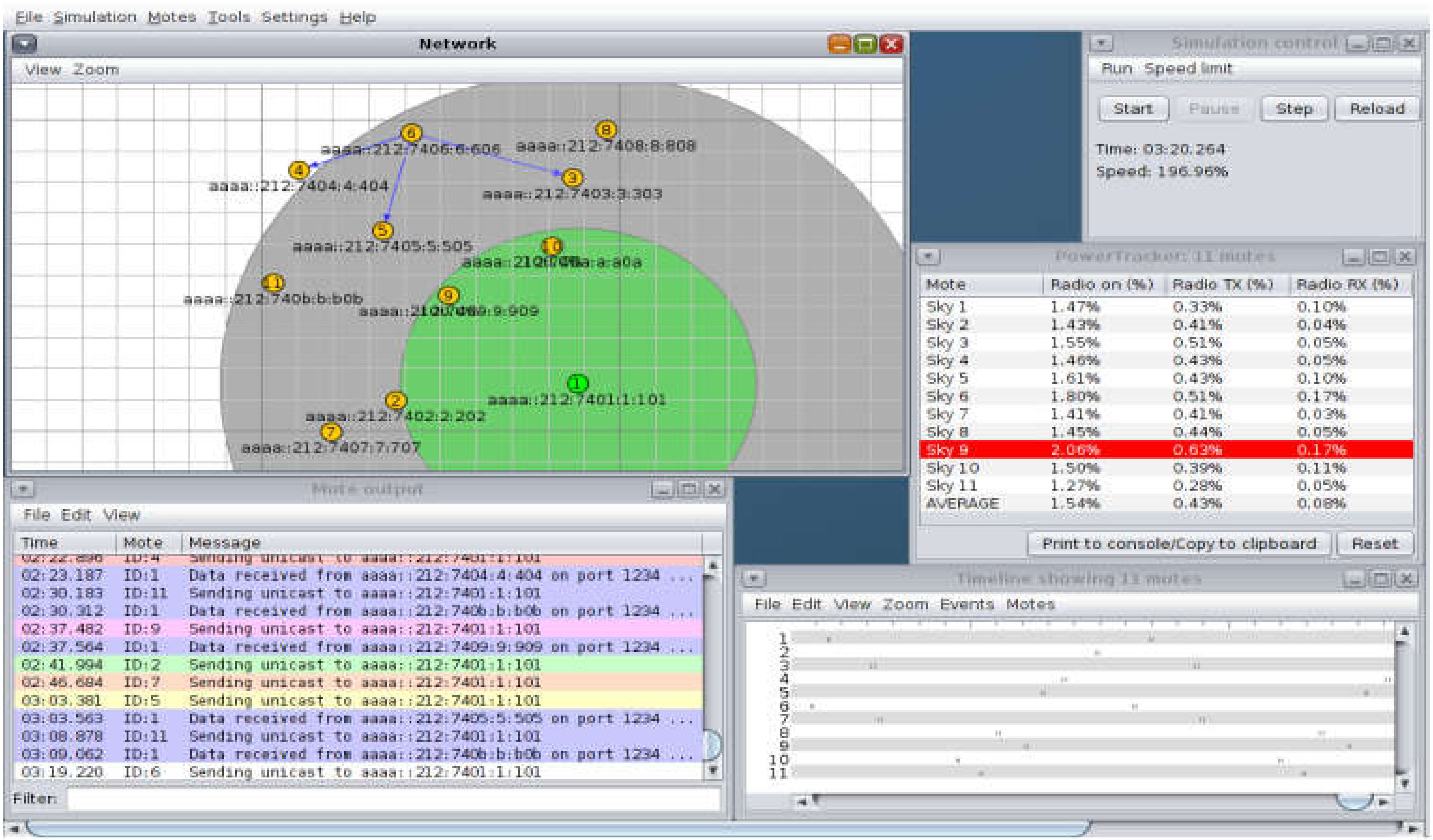
Cooja Emulator

- Cooja is a Contiki network emulator
  - An extensible Java-based simulator capable of emulating Tmote-Sky and Z1
- The code to be executed by the node is the exact same firmware you may upload to physical nodes
- Allows large and small networks of motes to be simulated
- Motes can be emulated at the hardware level
  - Slower but allows for precise inspection of system behavior
- Motes can also be emulated at a less detailed level
  - Faster and allows simulation of larger networks

Cooja Emulator

- Cooja is a highly useful tool for Contiki development
  - It allows developers to test their code and systems long before
    running it on the target hardware
  - Developers regularly set up new simulations to
    - debug their software
    - to verify the behavior of their systems

Cooja GUI



File Simulation Motes Tools Settings Help
Network
View Zoom
aaaa::212:7406:6:606
aaaa::212:7408:8:808
aaaa::212:7404:4:404
aaaa::212:7403:3:303
aaaa::212:7405:5:505
aaaa::212:740b:b:b0b
aaaa::212:7401:1:101
aaaa::212:7402:2:202
aaaa::212:7407:7:707
Simulation control
Run Speed limit
Start Pause Step Reload
Time: 03:20.264
Speed: 196.96%
PowerTracker: 11 motes
Mote Radio on (%) Radio TX (%) Radio RX (%)
Sky 1 1.47% 0.33% 0.10%
Sky 2 1.43% 0.41% 0.04%
Sky 3 1.55% 0.51% 0.05%
Sky 4 1.46% 0.43% 0.05%
Sky 5 1.61% 0.43% 0.10%
Sky 6 1.80% 0.51% 0.17%
Sky 7 1.41% 0.41% 0.03%
Sky 8 1.45% 0.44% 0.05%
Sky 9 2.06% 0.63% 0.17%
Sky 10 1.50% 0.39% 0.11%
Sky 11 1.27% 0.28% 0.05%
AVERAGE 1.54% 0.43% 0.08%
Print to console/Copy to clipboard Reset
Mote output
File Edit View
Time Mote Message
02:23.187 ID:1 Data received from aaaa::212:7404:4:404 on port 1234 ...
02:30.183 ID:11 Sending unicast to aaaa::212:7401:1:101
02:30.312 ID:1 Data received from aaaa::212:740b:b:b0b on port 1234 ...
02:37.482 ID:9 Sending unicast to aaaa::212:7401:1:101
02:37.564 ID:1 Data received from aaaa::212:7409:9:909 on port 1234 ...
02:41.994 ID:2 Sending unicast to aaaa::212:7401:1:101
02:46.684 ID:7 Sending unicast to aaaa::212:7401:1:101
03:03.381 ID:5 Sending unicast to aaaa::212:7401:1:101
03:03.563 ID:1 Data received from aaaa::212:7405:5:505 on port 1234 ...
03:08.878 ID:11 Sending unicast to aaaa::212:7401:1:101
03:09.062 ID:1 Data received from aaaa::212:740b:b:b0b on port 1234 ...
03:19.220 ID:6 Sending unicast to aaaa::212:7401:1:101
Filter:
Timeline showing 11 motes
File Edit View Zoom Events Motes

Examples in Contiki

- Hello-Word
- LED Blinking
- Timers (*Ref. book: IoT in 5 days*)
- Multiple Threads (*Ref. book: IoT in 5 days*)