

# Advanced Networking Architectures and Wireless Systems

---

Carlo Vallati

Assistant Professor@ University of Pisa

[c.vallati@iet.unipi.it](mailto:c.vallati@iet.unipi.it)

# Who's this guy??

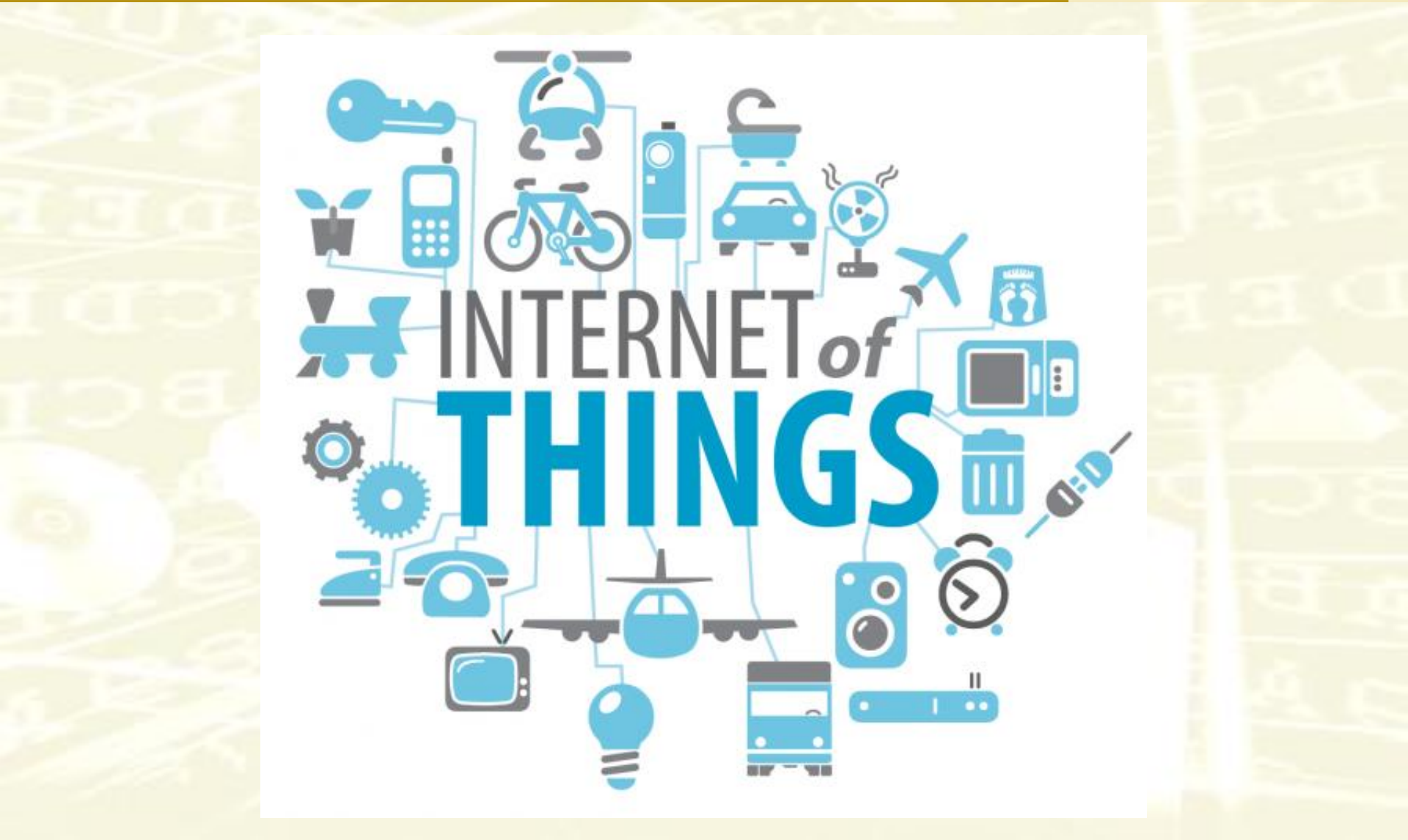


Carlo Vallati  
c.vallati@iet.unipi.it

<http://www.iet.unipi.it/c.vallati/>

# Pointers

- Lab homepage:
  - <http://lab-anaws.github.io/>
- Lab page on github for code and slides:
  - <https://github.com/lab-anaws>



# Outline



- Introduction
- IEEE 802.15.4 refreshment
- Basic network operations: IPv6+6LoWPAN and direct communication
- RPL: multi-hop communications
- CoAP



# Hardware

---

Carlo Vallati

Assistant Professor@ University of Pisa

[c.vallati@iet.unipi.it](mailto:c.vallati@iet.unipi.it)

# How an IoT device look like?



TelosB - SkyMote



Zolertia Z1

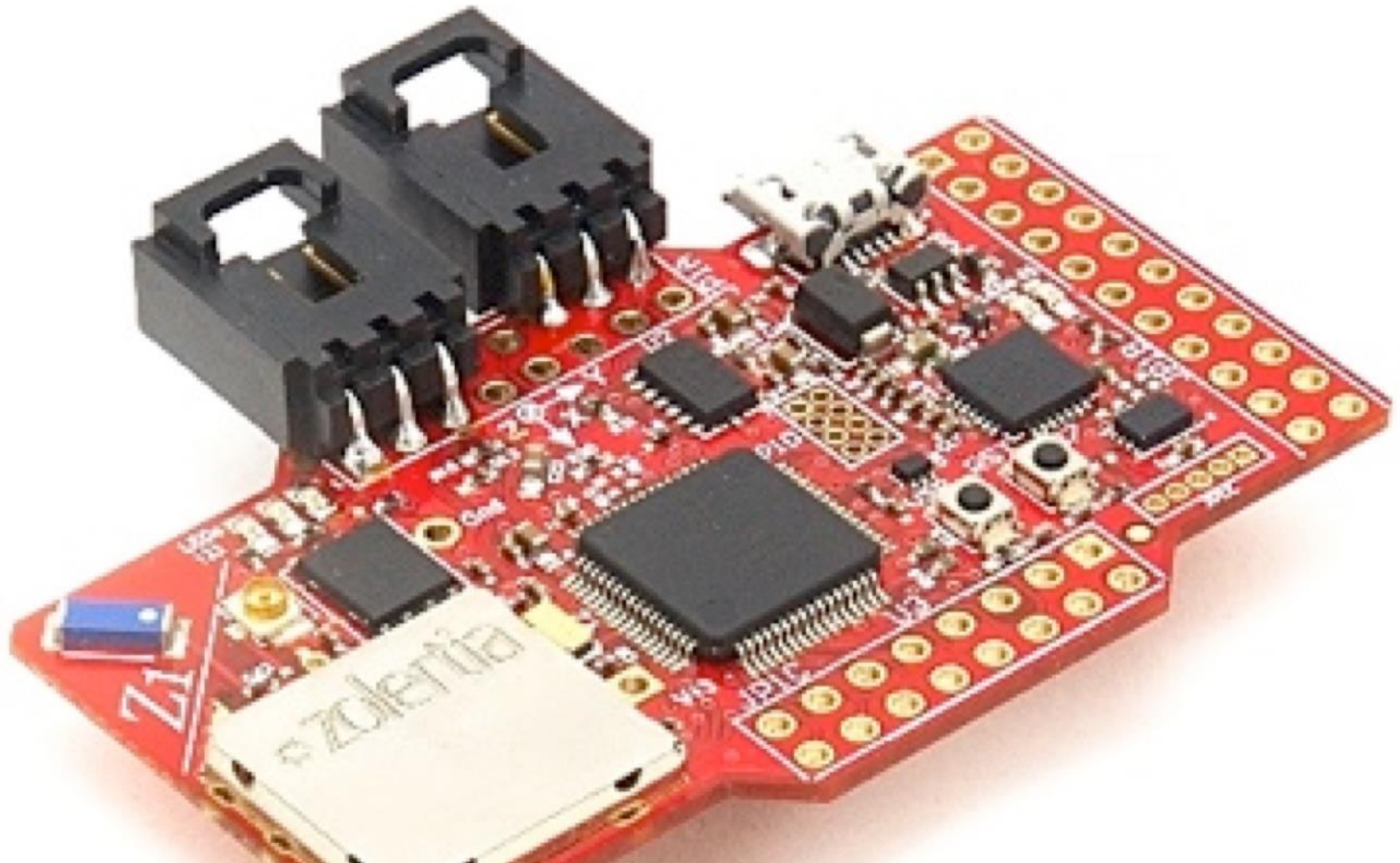


# Z<sup>1</sup>

Low-Power WSN Platform

 zolertia

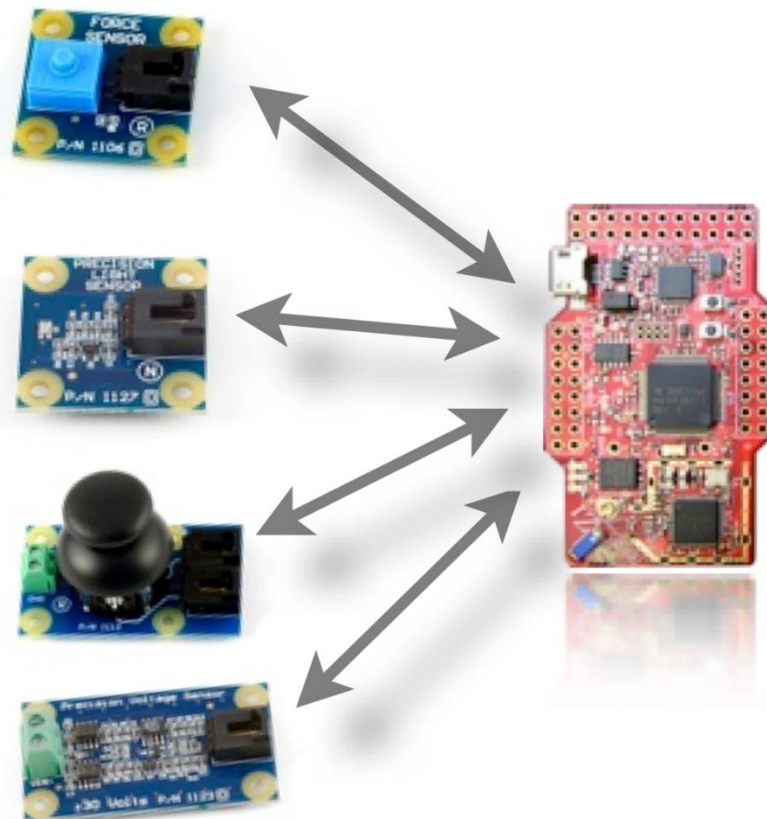
Smarten it up





# Zolertia Z1

★ Out of the box support for Phidgets™



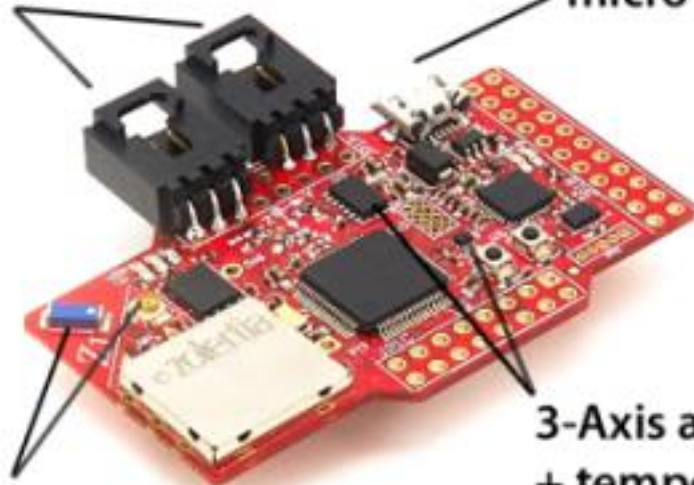
# Zolertia Z1



Similar specs for the SkyMote!

2 x Phidgets sensor ports

micro-USB



Ceramic embedded antenna

U.FL connector for external antenna

3-Axis accelerometer  
+ temperature sensor

## Main Features

- 2.4GHz IEEE® 802.15.4 & 6LoWPAN Compatible
- 2<sup>nd</sup> Generation MSP430(F2617)
- Widely Adopted Radio: CC2420
- On-board Digital Sensors (x2)
- Up to 4x Analog Phidgets™
- 52-pin Expansion Connector
- Embedded or External Antenna
- MicroUSB Connector

# Contiki – OS basics

---

Carlo Vallati

Assistant Professor @ University of Pisa

[c.vallati@iet.unipi.it](mailto:c.vallati@iet.unipi.it)

# WSN Operating Systems

- The OS hides many HW details
  - Simplify the programmer life
- Contains drivers to radio and sensors, scheduling, network stacks, process & power management
- TinyOS, **Contiki**, FreeRTOS, Mantis OS



# Contiki overview

- Contiki is a dynamic operating system for constrained devices
- Event driven kernel
  - Protothreads on top of it
- Support for many platform
  - Tmote Sky, Zolertia Z1, MicaZ ...
- Support for many CPU
- Programmed in C

<http://www.contiki-os.org/start.html>

<https://github.com/contiki-os/contiki/wiki>



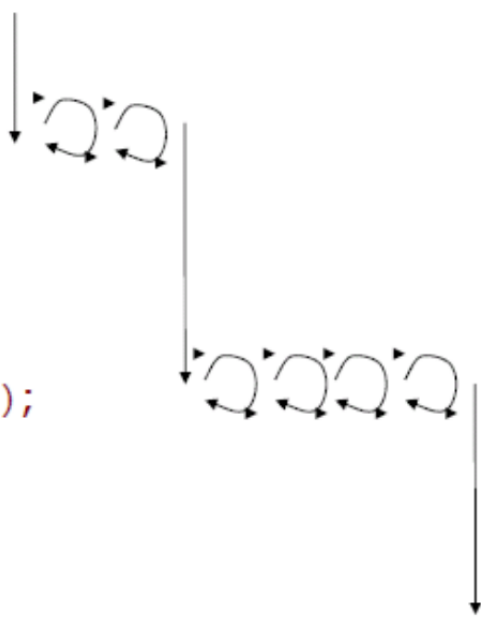
# Event vs Thread

- Event driven kernel only use events
  - Difficult to program
  - No sequential flow of control
  - Low overhead
- **Threads**
  - Easy to program
  - Sequential flow of control
  - High overhead (each thread has its own stack)

# Contiki solution: Protothreads

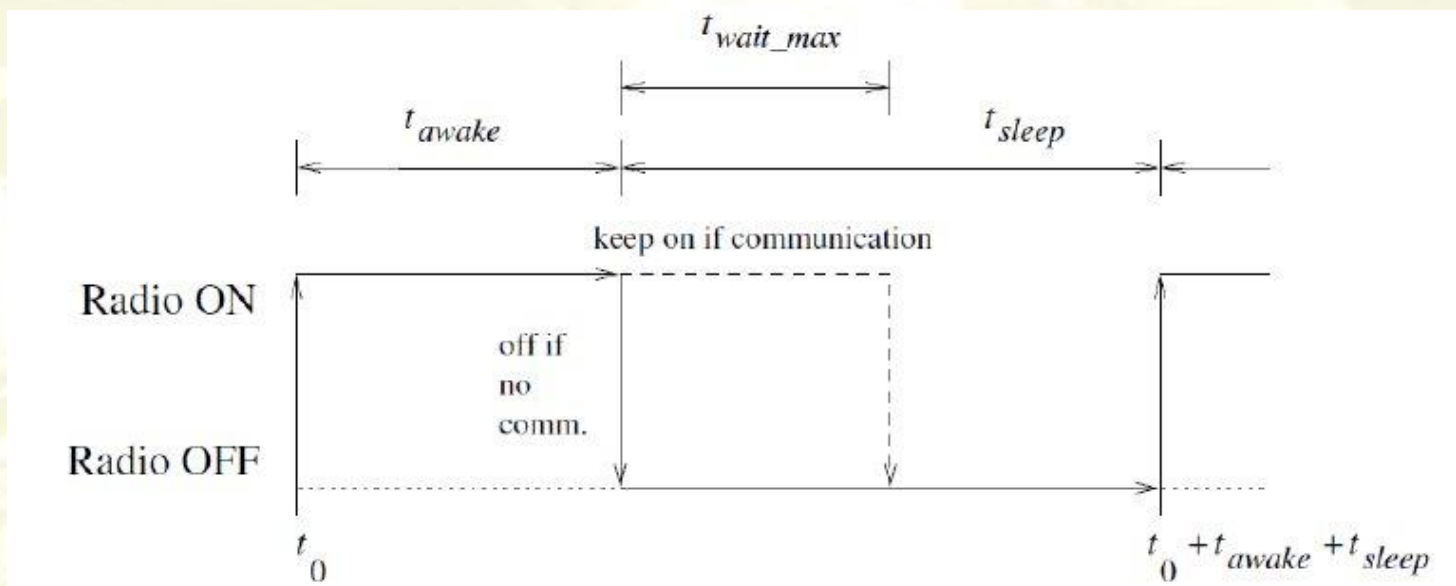
- Contiki adopts Protothreads: threads that have event management support
- Each Protothread has a sequential flow of instructions that can be interrupted to wait for events or conditions

```
int a_protothread(struct pt *pt) {  
    PT_BEGIN(pt);  
  
    PT_WAIT_UNTIL(pt, condition1);  
  
    if(something) {  
        PT_WAIT_UNTIL(pt, condition2);  
    }  
    PT_END(pt);  
}
```



# Protothreads: usage examples

- Simple MAC protocol:
  - Mix of sequential operations and operations triggered by events: Turn ON radio -> Wait for end of transmission or timeout -> Turn OFF radio





# Process Thread

- The code of the thread is called **process thread**
- Each process thread contains the code of a single protothread invoked from the process scheduler and it is declared as follows:

```
PROCESS_THREAD(hello_world_process, ev, data)
{
    PROCESS_BEGIN();
    printf("Hello, world\n");
    PROCESS_END();
}
```

A process thread must start with `PROCESS_BEGIN()`; and terminate with `PROCESS_END()`;

# Process Thread

- Inside the process thread the instructions are defined to manage events or conditions:
  - `PROCESS_WAIT_EVENT();` // Wait for any event.
  - `PROCESS_WAIT_EVENT_UNTIL();` // Wait for an event, but with a condition.
  - `PROCESS_YIELD();` // Wait for any event, equivalent to `PROCESS_WAIT_EVENT()`.
  - `PROCESS_WAIT_UNTIL();` // Wait for a given condition; may not yield the process.
  - `PROCESS_PAUSE();` // Temporarily yield the process.

# Main Process Thread

- Every Contiki image to run on devices must have a main process thread that runs automatically as the device boots up
- In order to make a process starting automatically the following autostart declaration must be added

```
PROCESS(example_process, "Example process");  
AUTOSTART_PROCESSES(&example_process);
```

# Contiki directories

- core
  - System source code
- apps
  - System apps
- platform
  - Platform-specific code
    - Default mote configuration
- cpu
  - CPU-specific code
- example
  - Lots of examples. **USE** it as a starting point.
- tools
  - Cooja and other useful stuff





# Hello World

```
#include "contiki.h"
#include <stdio.h>
/* 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"); /*code goes here*/

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



# Makefile

- The project includes a Makefile that specify how to produce the binary code:

```
CONTIKI_PROJECT = hello-world
```

```
all: $(CONTIKI_PROJECT)
```

```
CONTIKI = /home/user/contiki
```

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

# project-conf.h

- An additional configuration file is usually included to override operating system default configurations
- Add to Makefile

```
CFLAGS += -DPROJECT_CONF_H=\"project-conf.h\"
```
- Example change wireless channel

```
#undef RF_CHANNEL
#define RF_CHANNEL      26
```
- See parameters [platform/z1/contiki-conf.h](#)



# Load Hello-World Program

Go to “contiki/examples/hello-world”

Select the mote, in case you have more than one mote connected

Select the architecture, z1 for Zolertia sky for SkyMote

Compile the program:

**make MOTE=1 TARGET=z1 name-program**

Flash the program:

**make MOTE=1 TARGET=z1 name-program.upload**

name-program.sky is produced as binary

In case of problems in loading the program:

**chmod 777 /dev/ttyUSB0**





# Connect to the mote

To connect to the mote to obtain the log:

**make MOTE=1 login**

**Every mote has a node id which is used by some functions of the operating system.**

To set the node id:

**make burn-nodeid.upload nodeid=158 nodemac=158**

# Mote log



*If you have some issue in connecting to the mote serial port execute the following commands:*

```
wget https://github.com/cmorty/contiki/raw/pull/serialdump/tools/sky/serialdump-linux
```

```
mv serialdump-linux ~/contiki-3.0/tools/sky
```

# Do it!



- Load the hello-world program and grab the mote output

# Introduction to Cooja

- Cooja is a java based emulator for contiki nodes
- The hardware of motes is emulated
- Wireless connection among motes is simulated
  - Go to “contiki/tools/cooja”
  - Launch cooja:
    - **ant run**
  - The first time you need to run the following command to download some unmet dependencies:
    - **git submodule update --init**



# POST and WAIT

- `PROCESS_WAIT_EVENT ();`
  - Waits for an event to be posted to the process
- `PROCESS_WAIT_EVENT_UNTIL(condition c);`
  - Waits for an event to be posted to the process, with an extra condition. Often used: wait until timer has expired
  - `PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&timer));`
- `PROCESS_POST(...)` and `PROCESS_POST_SYNC(...)`
  - Post (a)synchronous event to a process.
  - The other process usually waits with `PROCESS_WAIT_EVENT_UNTIL(ev == EVENTNAME);`

# Timers



- struct timer
    - Passive timer, only keeps track of its expiration time
  - struct etimer
    - Active timer, sends an event when it expires
  - struct ctimer
    - Active timer, calls a function when it expires
  - struct rtimer
    - Real-time timer, calls a function at an exact time.
- Reserved for OS internals

# ETimer



- etimers are usually adopted:

```
#include "sys/etimer.h" // Include etimer
static struct etimer et; // Declare an etimer
...
etimer_set(&et, CLOCK_SECOND*4); // Set the timer
...
// Inside the main loop
PROCESS_WAIT_EVENT(); // Block and wait for any event
// Check if the timer is expired
if(etimer_expired(&et)){
    etimer_reset(&et); // Reset the timer
}
```

# Sensors



- Sensors can wait for external events generated from sensors, e.g. the user press a button:

```
#include "dev/button-sensor.h" // Add sensor library
#include "dev/leds.h" // Add led library
...
// Activate the button
SENSORS_ACTIVATE(button_sensor);
...
// In the main loop
PROCESS_WAIT_EVENT_UNTIL(ev==sensors_event &&
data==&button_sensor); // Wait until the user presses
the button
leds_toggle(LED_ALL); // Turn on all the leds
```

See [example/sky/test-button.c](#)



# Do IT!



- Write a program that loops indefinitely, check if the timer has expired, and if so, toggles LEDs and prints out a message.
- Write a program that loops indefinitely, waits for an event, check if a button has been pressed, toggles LEDs and prints out “Button Press!”. If, instead, the timer has expired toggles LEDs and prints out “Timer!”