RIOT Beginner Tutorial

RIOT Summit

https://github.com/riot-os/riot-course

Goal of the tutorial

1. Learn how to write and build a RIOT application

Goal of the tutorial

1. Learn how to write and build a RIOT application

2. Use IoT-LAB to run a RIOT application remotely on real hardware

Goal of the tutorial

1. Learn how to write and build a RIOT application

2. Use IoT-LAB to run a RIOT application remotely on real hardware

3. Learn the basics of security on IoT device

Goal of the tutorial

1. Learn how to write and build a RIOT application

2. Use IoT-LAB to run a RIOT application remotely on real hardware

3. Learn the basics of security on IoT device

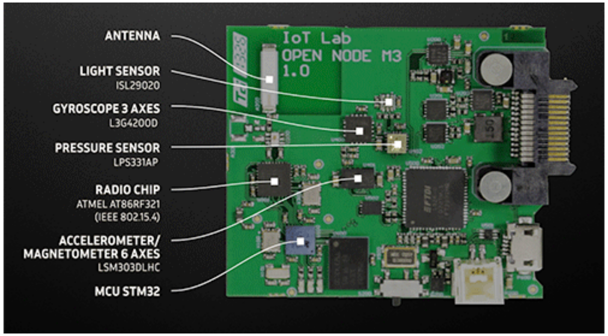
4. Perform a firmware update

Tutorial overview (1)

Tutorial overview (2)

- No setup required, all activities are performed online in Jupyter Notebooks

- Run the RIOT applications on the IoT-LAB testbed
- Use IoT-LAB M3 boards:

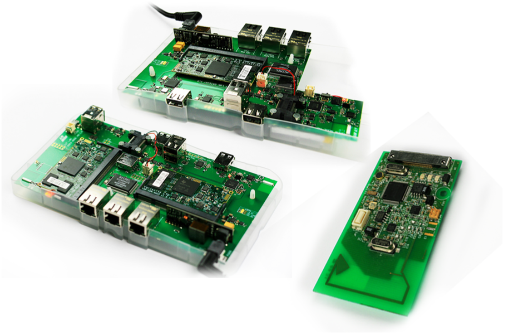


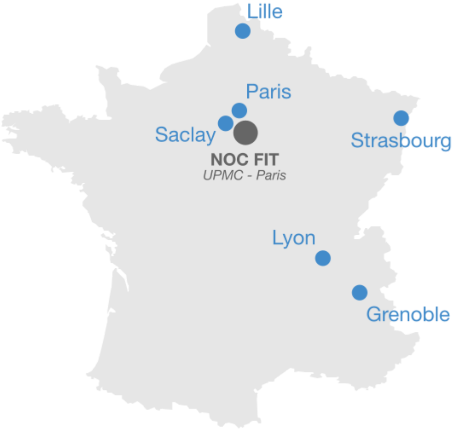
About IoT-LAB

https://www.iot-lab.info

IoT-LAB is a large scale experimentation testbed

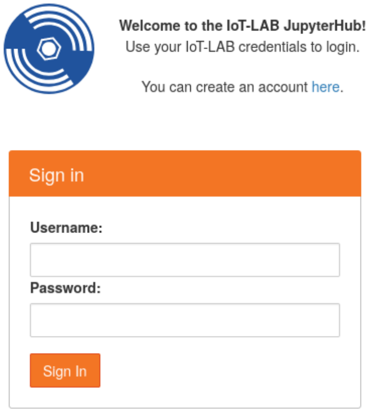
- Can be used for testing wireless communication networks on small devices
- Can be used for learning IoT programming and communication protocols
- Can be used for testing software platforms





About the Jupyter Notebooks

- Available at https://labs.iot-lab.info



Welcome to the IoT-LAB JupyterHub!
Use your IoT-LAB credentials to login.

You can create an account here.

- No setup required!

- Source code of the notebooks is available at https://github.com/iot-lab/iot-lab-training

Short demo: in Jupyterlab, read the notebook start.ipynb**

RIOT Overview

What is RIOT

- **operating system** for microcontrollers
  - **microkernel architecture** ⇒ require very low resources
  - **real-time** and **multi-threaded**
  - comes with **in-house networking stacks**

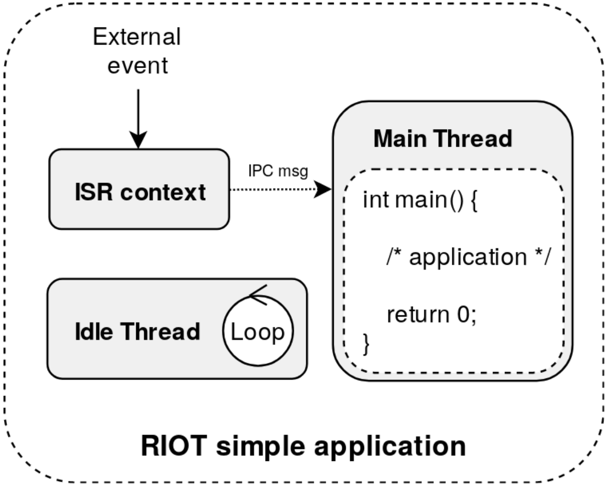
- **open-source:** https://github.com/RIOT-OS/RIOT
  - free software platform
  - **world-wide community** of developers

- **easy to use** and **reuse**
  - Standard programming in C
  - Standard tooling
  - **API is independent** from the hardware

General-Purpose OS for IoT

- **Real-Time** scheduler
  - ⇒ fixed priorities preemption with O(1) operations
  - ⇒ tickless scheduler, i.e. no periodic timer event

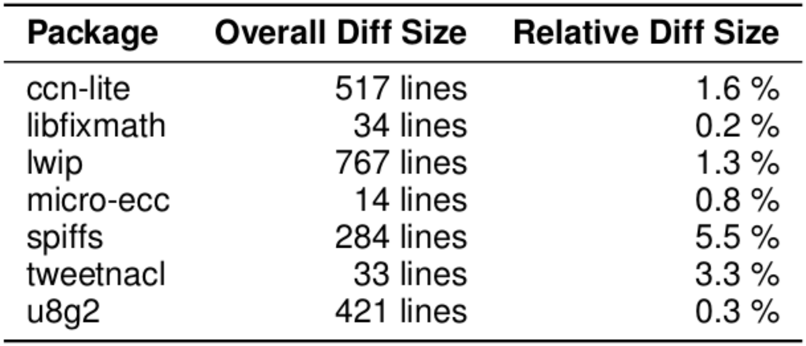
- **Multi-Threading** and IPC:
  - Separate thread contexts with separate thread memory stack
  - Minimal thread control block (TCB)
  - Thread synchronization using mutexes, semaphores and messaging
  - ISR context handles external events and notifies threads using IPC messages
  - *Note:* optional multi-threading



A modular OS

Features are provided as modules ⇒ **only build what's required**

- System libraries: **xtimer**, **shell**, crypto, etc
- Sensors and actuators
- Display drivers, filesystems, etc
- Embedded interpretors: Javascript, LUA, uPython
- High-level network protocols: CoAP, MQTT-SN, etc
- External packages: lwIP, Openthread, u8g2, loramac, etc



Package	Overall Diff Size	Relative Diff Size
ccn-lite	517 lines	1.6 %
libfixmath	34 lines	0.2 %
lwip	767 lines	1.3 %
micro-ecc	14 lines	0.8 %
spiffs	284 lines	5.5 %
tweetnacl	33 lines	3.3 %
u8g2	421 lines	0.3 %

Network stacks

IP oriented stacks ⇒ designed for Ethernet, WiFi, 802.15.4 networks

- **GNRC**: the in-house 802.15.4/6LowPAN/IPv6 stack of RIOT

Network stacks

IP oriented stacks ⇒ designed for Ethernet, WiFi, 802.15.4 networks

- **GNRC**: the in-house 802.15.4/6LowPAN/IPv6 stack of RIOT
- **Thread**: 802.15.4 IPv6 stack provided by the ThreadGroup

openthread

released by Nest

Network stacks

IP oriented stacks ⇒ designed for Ethernet, WiFi, 802.15.4 networks

- **GNRC**: the in-house 802.15.4/6LowPAN/IPv6 stack of RIOT
- **Thread**: 802.15.4 IPv6 stack provided by the ThreadGroup

openthread
released by Nest

- **OpenWSN** : a deterministic MAC layer implementing the IEEE 802.15.4e TSCH protocol

Network stacks

IP oriented stacks ⇒ designed for Ethernet, WiFi, 802.15.4 networks

- **GNRC**: the in-house 802.15.4/6LowPAN/IPv6 stack of RIOT
- **Thread**: 802.15.4 IPv6 stack provided by the ThreadGroup

openthread
released by Nest

- **OpenWSN** : a deterministic MAC layer implementing the IEEE 802.15.4e TSCH protocol

California
OPENWSN
BERKELEY

- Other IPv6 stacks:
  - **lwIP**: full-featured network stack designed for low memory consumption
  - **emb6**: A fork of Contiki network stack that can be used without proto-threads

Other network support

- In-house Controller Area Network (**CAN**)

Other network support

- In-house Controller Area Network (**CAN**)
- **BLE** stack support: NimBLE

Other network support

- In-house Controller Area Network (**CAN**)
- **BLE** stack support: NimBLE
- **LoRaWAN** stack ⇒ Compliant with LoRaWAN 1.0.2

Bluetooth™
4.0
Low Energy

LoRaWAN™

Other network support

- In-house Controller Area Network (**CAN**)
- **BLE** stack support: NimBLE
- **LoRaWAN** stack ⇒ Compliant with LoRaWAN 1.0.2
- **SigFox** support for ATA8520e modules

Other important features

- Full featured USB stack (CDC-ACM, CDC-ECM, etc)

Other important features

- Full featured USB stack (CDC-ACM, CDC-ECM, etc)

- Standard and secure software update implementation

https://datatracker.ietf.org/wg/suit/about/

Other important features

- Full featured USB stack (CDC-ACM, CDC-ECM, etc)

- Standard and secure software update implementation

https://datatracker.ietf.org/wg/suit/about/

Getting started

Structure of a RIOT application

A minimal RIOT application consists in:

- A `Makefile`

```
APPLICATION = example

BOARD ?= native

RIOTBASE ?= $(CURDIR)/../../../RIOT

DEVELHELP ?= 1

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

- A C-file containing the main function

```c
#include <stdio.h>

int main(void)
{
    puts("My first RIOT application");
    return 0;
}
```

Build a RIOT application

- The build system of RIOT is based on **make** build tool

Build a RIOT application

- The build system of RIOT is based on **make** build tool
- To build an application, **make** can be called in 2 ways:
  - From the application directory:

    ```
    $ cd <application_directory>
    $ make
    ```

  - From anywhere, by using the `-C` to specify the application directory:

    ```
    $ make -C <application_directory>
    ```

Build a RIOT application

- The build system of RIOT is based on **make** build tool
- To build an application, **make** can be called in 2 ways:
  - From the application directory:

    ```
    $ cd <application_directory>
    $ make
    ```

  - From anywhere, by using the `-C` to specify the application directory:

    ```
    $ make -C <application_directory>
    ```

- Use the **BOARD** variable to specify the target at build time

  ```
  $ make BOARD=<target> -C <application_directory>
  ```

  `BOARD` can be any board supported by RIOT
  ⇒ see the **RIOT/boards** directory for the complete list

Build a RIOT application

- The build system of RIOT is based on **make** build tool
- To build an application, **make** can be called in 2 ways:
  - From the application directory:

    ```
    $ cd <application_directory>
    $ make
    ```

  - From anywhere, by using the `-C` to specify the application directory:

    ```
    $ make -C <application_directory>
    ```

- Use the **BOARD** variable to specify the target at build time

  ```
  $ make BOARD=<target> -C <application_directory>
  ```

  `BOARD` can be any board supported by RIOT
  ⇒ see the **RIOT/boards** directory for the complete list

- Use the **RIOTBASE** variable to specify the RIOT source base directory

Run a RIOT application

This depends on the target board:

- Running on **native**: the RIOT application executed is a simple Linux process

  ```
  $ make BOARD=native -C <application_dir>
  $ <application_dir>/bin/native/application.elf
  ```

- Running on **hardware**: the RIOT application must be *flashed* first on the board

Run a RIOT application

This depends on the target board:

- Running on **native**: the RIOT application executed is a simple Linux process

  ```
  $ make BOARD=native -C <application_dir>
  $ <application_dir>/bin/native/application.elf
  ```

- Running on **hardware**: the RIOT application must be *flashed* first on the board

⇒ use the **flash** and **term** targets with make

- **flash**: build and write the firmware on the MCU flash memory
- **term**: opens a terminal client connected to the serial port of the target

All this can be done in one command:

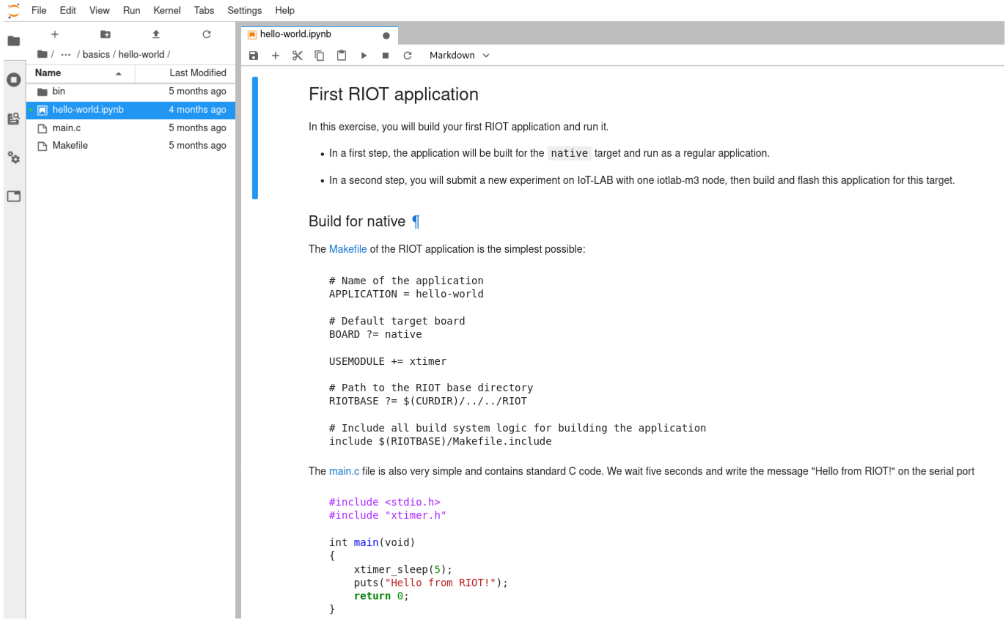
```
$ make BOARD=<target> -C <application_dir> flash term
```

Note: the last command can also be used with **native** target

Exercise: your first RIOT application

Let's build and run our first RIOT application !

In jupyterlab, open the notebook **riot/basics/hello-world/hello-world.ipynb** and follow the instructions.



Name	Last Modified
bin	5 months ago
hello-world.ipynb	4 months ago
main.c	5 months ago
Makefile	5 months ago

First RIOT application

In this exercise, you will build your first RIOT application and run it.

- In a first step, the application will be built for the `native` target and run as a regular application.
- In a second step, you will submit a new experiment on IoT-LAB with one iotlab-m3 node, then build and flash this application for this target.

Build for native ¶

The Makefile of the RIOT application is the simplest possible:

```
# Name of the application
APPLICATION = hello-world

# Default target board
BOARD ?= native

USEMODULE += xtimer

# Path to the RIOT base directory
RIOTBASE ?= $(CURDIR)/../../RIOT

# Include all build system logic for building the application
include $(RIOTBASE)/Makefile.include
```

The main.c file is also very simple and contains standard C code. We wait five seconds and write the message "Hello from RIOT!" on the serial port

```
#include <stdio.h>
#include "xtimer.h"

int main(void)
{
    xtimer_sleep(5);
    puts("Hello from RIOT!");
    return 0;
}
```

How to extend the application

⇒ by adding modules in the application `Makefile` or from the command line:

- Add extra modules with **USEMODULE**
  ⇒ `xtimer`, `fmt`, `shell`, `ps`, etc
- Include external packages with **USEPKG**
  ⇒ `lwip`, `semtech-loramac`, etc
- Use MCU peripherals drivers with **FEATURES_REQUIRED**:
  ⇒ `periph_gpio`, `periph_uart`, `periph_spi`, `periph_i2c`

How to extend the application

⇒ by adding modules in the application `Makefile` or from the command line:

- Add extra modules with **USEMODULE**
  ⇒ `xtimer`, `fmt`, `shell`, `ps`, etc

- Include external packages with **USEPKG**
  ⇒ `lwip`, `semtech-loramac`, etc

- Use MCU peripherals drivers with **FEATURES_REQUIRED**:
  ⇒ `periph_gpio`, `periph_uart`, `periph_spi`, `periph_i2c`

Example in a `Makefile`:

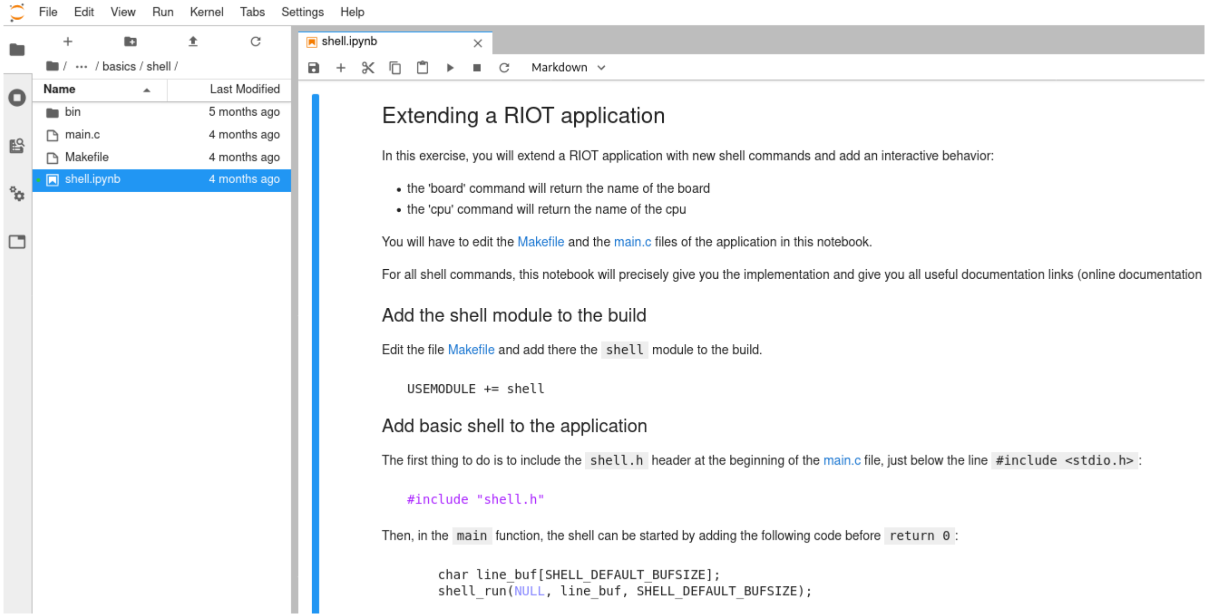
```
USEMODULE += xtimer shell

USEPKG += semtech-loramac

FEATURES_REQUIRED += periph_gpio
```

Example from the command line:

```
$ USEMODULE=xtimer make BOARD=b-l072z-lrwan1
```

Exercise: write an application with a shell

Follow the instructions in the notebook **riot/basics/shell/shell.ipynb**



Extending a RIOT application

In this exercise, you will extend a RIOT application with new shell commands and add an interactive behavior:

- the 'board' command will return the name of the board
- the 'cpu' command will return the name of the cpu

You will have to edit the Makefile and the main.c files of the application in this notebook.

For all shell commands, this notebook will precisely give you the implementation and give you all useful documentation links (online documentation

Add the shell module to the build

Edit the file Makefile and add there the `shell` module to the build.

```
USEMODULE += shell
```

Add basic shell to the application

The first thing to do is to include the `shell.h` header at the beginning of the main.c file, just below the line `#include <stdio.h>` :

```
#include "shell.h"
```

Then, in the `main` function, the shell can be started by adding the following code before `return 0` :

```
    char line_buf[SHELL_DEFAULT_BUFSIZE];
    shell_run(NULL, line_buf, SHELL_DEFAULT_BUFSIZE);
```

Security basics for IoT

Why is security difficult on low-end IoT devices ?

blablabla

Exercise: compute a hash

Follow the instructions in the notebook **riot/security/hash/hash.ipynb**

Exercise: sign and verify signature

Follow the instructions in the notebook **riot/security/signature/signature.ipynb**

Exercise: encrypt and decrypt a message

Follow the instructions in the notebook **riot/security/encyption/encyption.ipynb**

Exercise: secure communication using dtls

Follow the instructions in the notebook **riot/security/dtls/dtls.ipynb**

Exercise: standard and secure firmware update

Follow the instructions in the notebook **riot/security/ota/ota.ipynb**

All complete ? Well done!