RIOT Hands-on Tutorial

Martine Sophie Lenders

Starting the RIOT

Preparations (1)

- Homework:
  - Setup PC for compiling RIOT (+ test your setup)
    `https://github.com/RIOT-OS/Tutorials`
  - Create an IoT-LAB account
- Install `iotlabcli` (in the VM when using Vagrant) for python:

```
pip3 install iotlabcli
```

- Make sure SSH is configured on your system (or VM):

```
ssh-keygen
cat .ssh/id_rsa.pub
# copy to SSH keys at
#   https://www.iot-lab.info/testbed/account
ssh "<iotlab user>"@lille.iot-lab.info
# say "yes" and log out again using `exit`
```

Preparations (2)

- There is a GUI dashboard, but we will use the CLI

  `https://www.iot-lab.info/testbed/dashboard`

- Log into IoT-LAB using `iotlabcli`: `iotlab-auth -u "<iotlab user>"`
- Start a 2 hour experiment on the Testbed

```
iotlab-experiment submit -d 120 \
  --list 1,site=lille+archi=m3:at86rf231
```

```
{
    "id": 234780
}
```

- Get experiment information: `iotlab-experiment get -n -i 234780`
- Note down `network_address` of your node

Running RIOT

- Applications in RIOT consist at minimum of
  - a `Makefile`
  - a C-file, containing a `main()` function
- To see the code go to the task-01 directory:

```
cd task-01
ls
```

Your first application – The Makefile

```makefile
# name of your application
APPLICATION = Task01

# If no BOARD is found in the environment, use this default:
BOARD ?= native

# This has to be the absolute path to the RIOT base directory:
RIOTBASE ?= $(CURDIR)/../../RIOT

# Comment this out to disable code in RIOT that does safety checking
# which is not needed in a production environment but helps in the
# development process:
CFLAGS += -DDEVELHELP

# Change this to 0 show compiler invocation lines by default:
QUIET ?= 1

# Modules to include:
USEMODULE += shell
USEMODULE += shell_commands
USEMODULE += ps

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

Your first application – The C-file

```c
#include <stdio.h>
#include <string.h>

#include "shell.h"

int main(void)
{
    puts("This is Task-01");

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

    return 0;
}
```

Task 1.1: Run your first application as Linux process

1. Compile & run on `native`: `make all term`
2. Type `help`
3. Type `ps`
4. Modify your application:
   - Add a `printf("This application runs on %s", RIOT_BOARD);` *before* `shell_run()`
   - Recompile and restart `make all term`
   - Look at the result

Task 1.2: Run your first application on real hardware

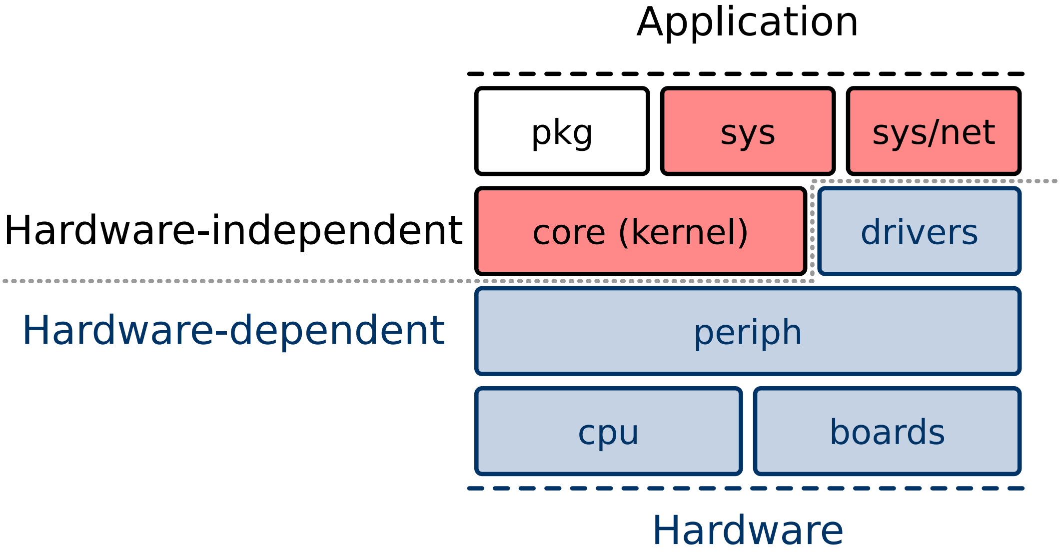
1. Compile, flash and run on `iotlab-m3`

```
BOARD=iotlab-m3 \
IOTLAB_NODE="<your network_address>" \
    make all flash term
```

2. Verify output of `RIOT_BOARD`

RIOT’s architecture

Structural elements of RIOT



- Focus of this tutorial: `core`, `sys`, and `sys/net`

Custom shell commands

Writing a shell handler

- Shell command handlers in RIOT are functions with signature

```
int cmd_handler(int argc, char **argv);
```

- argv: array of strings of arguments to the command

```
print hello world    # argv == {"print", "hello", "world"}
```

- argc: length of argv

Adding a shell handler to the shell

- Shell commands need to be added manually to the shell on initialization

```c
#include "shell.h"

static const shell_command_t shell_commands[] = {
    { "command name", "command description", cmd_handler },
    { NULL, NULL, NULL }
};

/* ... */
    shell_run(commands, line_buf, SHELL_DEFAULT_BUFSIZE)
/* ... */
```

Task 2.1 – A simple echo command handler

- Go to task-02 directory (`cd ../task-02`)
- Write a simple echo command handler in main.c:
  - First argument to the echo command handler shall be printed to output

```
> echo "Hello World"
Hello World
> echo foobar
foobar
```

Task 2.2 – Control the hardware

- `led.h` defines a macro `LED0_TOGGLE` to toggle the primary LED on the board.
- Write a command handler `toggle` in main.c that toggles the primary LED on the board

Multithreading

Threads in RIOT

- Threads in RIOT are functions with signature

```c
void *thread_handler(void *arg);
```

- Use `thread_create()` from `thread.h` to start:

```c
pid = thread_create(stack, sizeof(stack),
                    THREAD_PRIORITY_MAIN - 1,
                    THREAD_CREATE_STACKTEST,
                    thread_handler,
                    NULL, "thread");
```

RIOT kernel primer

Scheduler:

- Tick-less scheduling policy ($O(1)$):
  - Highest priority thread runs until finished or blocked
  - ISR can preempt any thread at all time
  - If all threads are blocked or finished:
    - Special IDLE thread is run
    - Goes into low-power mode

IPC (not important for the following task):

- Synchronous (default) and asynchronous (optional, by IPC queue initialization)

Task 3.1 – Start a thread

- Go to task-03 directory (`cd ../task-03`)
- Open `main.c`
- Reminder:

```c
pid = thread_create(stack, sizeof(stack),
                    THREAD_PRIORITY_MAIN - 1,
                    THREAD_CREATE_STACKTEST,
                    thread_handler,
                    NULL, "thread");
```

- Start the thread `"thread"` from within `main()`
- Run the application on `native`: `make all term`
- Check your output, it should read: `I'm in "thread" now`

Timers

xtimer primer

- `xtimer` is the high level API of RIOT to multiplex hardware timers
- Examples for functionality:
  - `xtimer_now_usec()` to get current system time in microseconds
  - `xtimer_sleep(sec)` to sleep `sec` seconds
  - `xtimer_usleep(usec)` to sleep `usec` microseconds

Task 4.1 – Use `xtimer`

- Reminder: Functions `xtimer_now_usec()`, `xtimer_sleep()`, and `xtimer_usleep()` were introduced
- Go to task-04 directory (`cd ../task-04`)
- Note the inclusion of `xtimer` in Makefile

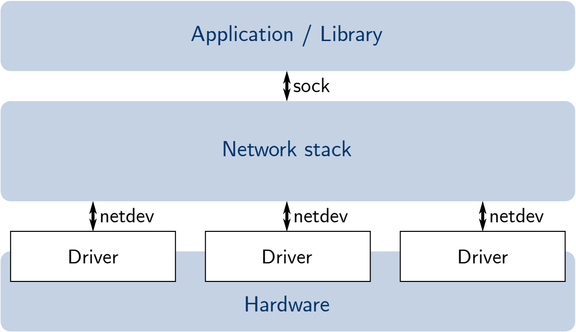
```
USEMODULE += xtimer
```

- Create a thread in `main.c` that prints the current system time every 2 seconds
- Check the existence of the thread with ps shell command

General networking architecture

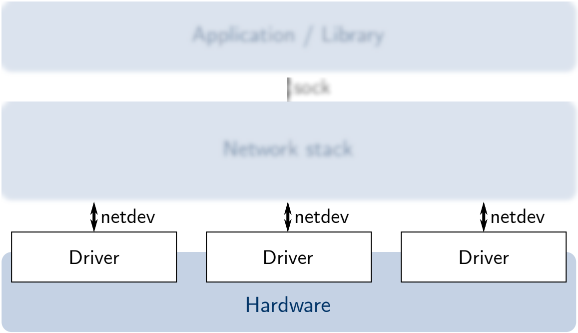
RIOT's Networking architecture

- Designed to integrate any network stack into RIOT



RIOT's Networking architecture

- Designed to integrate any network stack into RIOT



Including the network device driver

- Go to task-05 directory (`cd ../task-05`)
- Note inclusion of `netdev` modules in Makefile

```
USEMODULE += gnrc_netdev_default
USEMODULE += auto_init_gnrc_netif
```

Virtual network interface on `native`

- Use `tapsetup` script in RIOT repository:

```
./../RIOT/dist/tools/tapsetup/tapsetup -c 2
```

- Creates
  - Two TAP interfaces `tap0` and `tap1` and
  - A bridge between them (`tapbr0` on Linux, `bridge0` on OSX)
- Check with `ifconfig` or `ip link`!

Task 5.1 – Your first networking application

- Run the application on `native`: `PORT=tap0 make all term`
- Type `help`
- Run a second instance with `PORT=tap1 make all term`
- Type `ifconfig` on both to get hardware address and interface number
- Use `txtsnd` command to exchange messages between the two instances

Task 5.2 – Use your application on real hardware

- Compile, flash, and run on the board

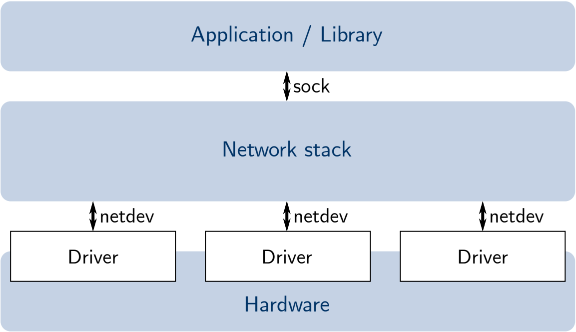
  ```
  BOARD=iotlab-m3 \
  IOTLAB_NODE="<your network_address>" \
    make all flash term
  ```

- Type `ifconfig` to get your hardware addresses
- Use map at

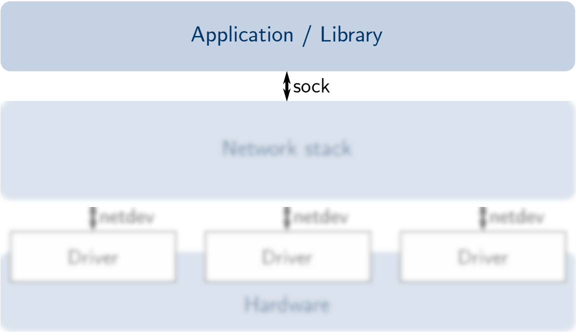
  ```
  https://www.iot-lab.info/testbed/status
  ```

  and talk to each other to find out who your neighbors are
- Use `txtsnd` to send one of your neighbors a friendly message

RIOT's Networking architecture



RIOT's Networking architecture



sock

- collection of unified connectivity APIs to the transport layer
- What's the problem with POSIX sockets?
  - too generic for most use-cases
  - numerical file descriptors (internal storage of state required)
  - in general: too complex for usage, too complex for porting
- protocol-specific APIs:
  - `sock_ip` (raw IP)
  - `sock_udp` (UDP)
  - `sock_tcp` (TCP)
  - ...
- both IPv4 and IPv6 supported

Task 6.1 – Use UDP for messaging

- Go to `task-06` directory `cd ../task-06`
- Note the addition of `gnrc_sock_udp` to Makefile
- `udp.c` utilizes `sock_udp_send()` and `sock_udp_recv()` to exchange UDP packets
- Compile and run on two `native` instances
- Type `help`
- Use `udps 8888` to start a UDP server on port 8888 on first instance (check with `ps`)
- Use `ifconfig` to get link-local IPv6 address of first instance
- Send UDP packet from second instance using `udp` command to first instance

Task 6.2 – Communicate with Linux

- Compile and run a `native` instance
- Start a UDP server on port 8888 (using `udps`)
- Send a packet to RIOT from Linux using `netcat`

```
echo "hello" | nc -6u <RIOT-IPv6-addr>%tapbr0 8888
```

- Start a UDP server on Linux `nc -6lu 8888`
- Send a UDP packet from RIOT to Linux
  `udp <tap0-IPv6-addr> 8888 hello`

Task 6.3 – Exchange UDP packets with your neighbors

- Compile, flash, and run on the board

```
BOARD=iotlab-m3 \
IOTLAB_NODE="<your network_address>" \
  make all flash term
```

- Send and receive UDP messages to and from your neighbors using `udp` and `udps`

SAUL

Better call SAUL!

- The Sensor/Actuator Uber Layer (SAUL) is a sensor/actuator abstraction layer for RIOT
- Device drivers can be registered via the SAUL registry
- Read/write Access via common API:

```c
#include <stdio.h>

#include "saul_reg.h"

int main(void)
{
    saul_reg_t *dev = saul_reg;

    while (dev) {
        int dim;
        phydat_t res;

        dim = saul_reg_read(dev, &res);
        if (dim <= 0) {
            continue;
        }
        puts(dev->name);
        phydat_dump(&res, dim);
        dev = dev->next;
    }
    return 0;
}
```

Task 7.1 – Use SAUL

- Go to `saul` example in RIOT directory `cd ../RIOT/examples/saul`
- The `main.c` does not contain much
- `shell_command` module magic! So have a look at the Makefile:
  - `saul_default` pulls in everything you need
- Compile, flash, and run on the board

```
BOARD=iotlab-m3 \
IOTLAB_NODE="<your network_address>" \
  make all flash term
```

- Command `saul` lists all actuators and sensors
- Read the sensor data using the `saul read` command
- You can also toggle the LEDs again using `saul write`

Task 7.2 – Familiarize yourself with the API

- SAUL example did not contain any API usage
- Go back to the RIOT root directory: `cd ../..`
- Shell commands pulled in by the `shell_commands` module are in `sys/shell/commands`
- Have a look at `sc_saul_reg.c`:
  - `list()` implements saul command
  - `read()` implements `saul read` command
  - `write()` implements `saul write` command
- More functions described at https://doc.riot-os.org/group__sys__saul__reg.html

There is so much more!

Where can I learn more about RIOT?

- Have a look at the examples (and also the tests) in RIOT

```
ls RIOT/examples
ls RIOT/tests
ls RIOT/sys/shell/commands
```

- Read the documentation at https://doc.riot-os.org
- Have questions? Don't hesitate to ask the friendly community at https://forum.riot-os.org or in the #riot-os:matrix.org chat

Now go out and make something!