Test Documentation

Release test

Test

November 28, 2015

Contents

1 FEATURES 3

2 GETTING STARTED 5
2.1 USING THE NATIVE PORT WITH NETWORKING 5

3 CONTRIBUTE 7

4 MAILING LISTS 9

5 LICENSE 11

6 CONTRIBUTE 13

7 Platform configurations for RIOT-OS 15

8 Mulle OpenOCD configuration files 17

9 Zolertia Re-Mote platform 19

10 Port Features 21

11 Requirements 23
11.1 Install a Toolchain 23
11.2 Drivers 23

12 More Reading 25

13 K60 tools 27
13.1 Watchdog disable 27

14 Valgrind Support 29

15 Network Support 31

16 Setting Up A Tap Network 33

17 Daemonization 35

18 Compile Time Options 37

19 RIOT integration into IoT-LAB **39**
19.1 Control IoT-LAB via Make . . . 39

20 About **41**

21 Example usage **43**

22 Default options **45**

23 What to do about the findings **47**

24 cmdline2xml.sh **49**
24.1 Instrucions . . . 49

25 About **51**

26 Usage **53**

27 RIOT Sniffer Application **55**
27.1 About . . . 55
27.2 Dependencies . . . 55
27.3 Usage . . . 55

28 Creating a SLIP network interface **59**

29 Installation **61**

30 Usage **63**

31 USB to serial adapter tools **65**
31.1 Usage . . . 65
31.2 Exit codes . . . 65
31.3 Makefile example usage . . . 65
31.4 Limitations . . . 66

32 Getting started {#getting-started} **67**

33 Downloading RIOT code {#downloading-riot-code} **69**

34 Compiling RIOT {#compiling-riot} **71**
34.1 Setting up a toolchain {#setting-up-a-toolchain} . . . 71
34.2 The build system {#the-build-system} . . . 71
34.3 Building and executing an examples {#building-and-executing-and-example} . . . 72

35 RIOT Documentation {#mainpage} **73**

36 RIOT in a nutshell {#riot-in-a-nutshell} **75**

37 Contribute to RIOT {#contribute-to-riot} **77**

38 The quickest start {#the-quickest-start} **79**

39 Structure {#structure} **81**
39.1 core . . . 81
39.2 boards . . . 81
39.3 cpu . . . 82
39.4 drivers . . . 82

39.5 sys . . . 82
39.6 sys/net . . . 82
39.7 pkg . . . 83
39.8 examples . . . 83
39.9 tests . . . 83
39.10 dist & doc . . . 83

40 examples/arduino_hello-world **85**

41 Arduino and RIOT **87**

42 Usage **89**

43 Example output **91**

44 examples/default **93**

45 Usage **95**

46 Example output **97**

47 RIOT specific **99**

48 Networking **101**

49 gnrc_networking_border_router example **103**
49.1 Requirements . . . 103
49.2 Configuration . . . 103

50 gnrc_networking example **105**
50.1 Connecting RIOT native and the Linux host . . . 105

51 Hello World! **107**

52 IPC Pingpong! **109**

53 examples/posix_sockets **111**

54 Usage **113**

55 Example output **115**

56 Using C++ and C in a program with RIOT **117**
56.1 Makefile Options . . . 117

57 Creating a patch with git **119**

58 OpenWSN on RIOT **121**

59 Usage **123**

60 About **125**

61 Usage **127**

62 About **129**

63 Usage **131**

64 About **133**

65 Usage **135**

66 About **137**

67 Usage **139**

68 About **141**

69 Usage **143**

70 About **145**

71 Usage **147**

72 About **149**

73 Usage **151**

74 About **153**

75 Usage **155**

76 About **157**

77 Usage **159**

78 About **161**

79 Usage **163**

80 About **165**

81 Usage **167**

82 About **169**

83 Usage **171**

84 About **173**

85 Usage **175**

86 About **177**

87 Usage **179**

88 About **181**

89 Usage **183**

90 Test for nrf24l01p lowlevel functions **185**
90.1 About 185
90.2 Predefined pin mapping 185
90.3 Usage 185
90.4 Expected Results 186

91 Expected result **189**

92 Background **191**

93 About **193**

94 Usage **195**

95 About **197**

96 Usage **199**

97 About **201**

98 Usage **203**
98.1 Interrupt driven 203
98.2 Polling Mode 203

99 Background **205**

100Expected result **207**

101About **209**

102Usage **211**

103About **213**

104Usage **215**

105About **217**

106Usage **219**

107About **221**

108Usage **223**

109About **225**

110Usage **227**

111Expected result **229**

112Background **231**

113Expected result **233**

114Background **235**

115About **237**

116Usage **239**

117Expected result **241**

118Background **243**

119Expected result **245**

120Background **247**

121Expected result **249**

122Background **251**

123Expected result **253**

124Background **255**

125Expected result **257**

126Background **259**

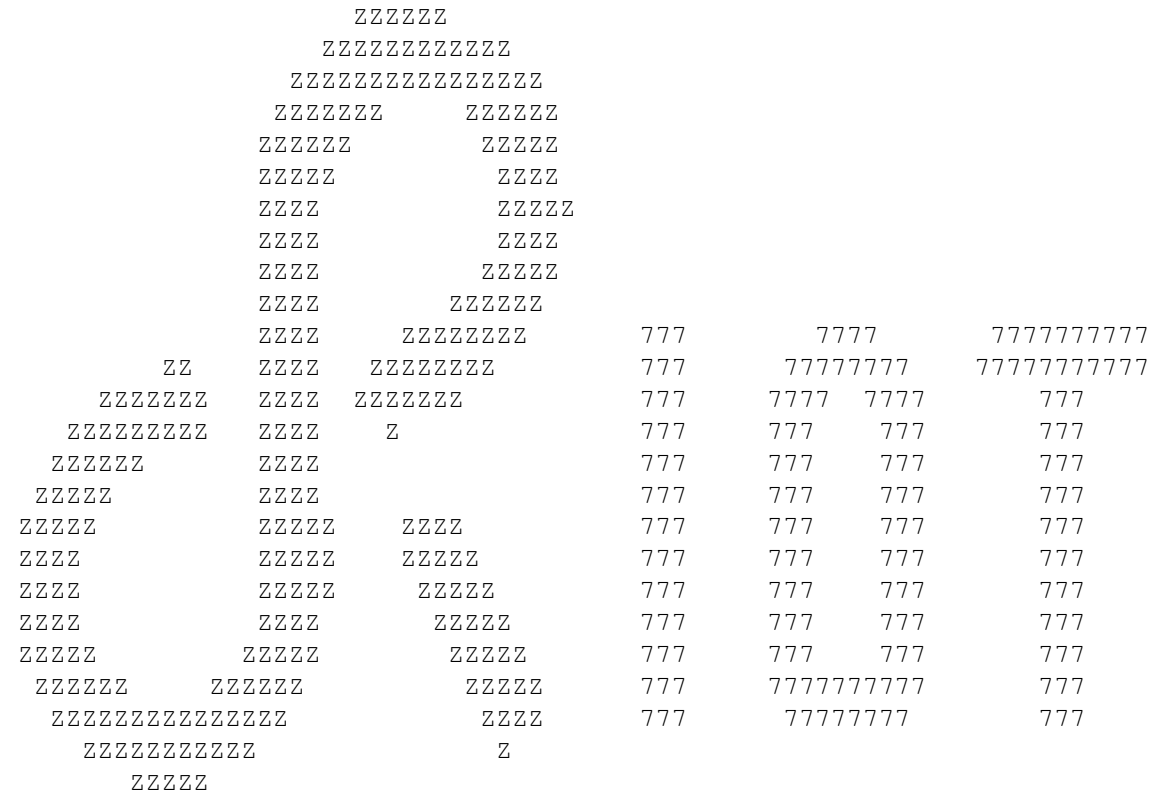
127Expected result **261**

128Background **263**

129Unittests **265**
129.1 Building and running tests 265
129.2 Writing unit tests 266

130Test warning on conflicting features **271**

Markdown files:



The friendly Operating System for IoT!

CHAPTER 1

FEATURES

RIOT OS is an operating system for Internet of Things (IoT) devices. It is based on a microkernel and designed for

- energy efficiency
- hardware independent development
- a high degree of modularity

Its features comprise

- a preemptive, tickless scheduler with priorities
- flexible memory management
- high resolution timers
- virtual, long-term timers
- the native port allows to run RIOT as-is on Linux, BSD, and MacOS. Multiple instances of RIOT running on a single machine can also be interconnected via a simple virtual Ethernet bridge
- Wiselib support (C++ algorithm library, including routing, clustering, timesync, localization, security and more algorithms)
- IPv6
- UDP
- 6LoWPAN
- NHDP

CHAPTER 2

GETTING STARTED

- You want to start the RIOT? Just follow our Getting started documentation
- The RIOT API itself can be built from the code using doxygen. The latest version is uploaded daily to http://riot-os.org/api.

2.1 USING THE NATIVE PORT WITH NETWORKING

If you compile RIOT for the native cpu and include the nativenet module, you can specify a network interface like this: `PORT=tap0 make term`

2.1.1 SETTING UP A TAP NETWORK

There is a shellscript in `RIOT/dist/tools/tapsetup` called `tapsetup` which you can use to create a network of tap interfaces.

USAGE To create a bridge and two (or count at your option) tap interfaces:

```
./dist/tools/tapsetup/tapsetup [-c [<count>]]
```

CHAPTER 3

CONTRIBUTE

To contribute something to RIOT, please refer to the development procedures and read all notes for best practice.

CHAPTER 4

MAILING LISTS

- RIOT OS kernel developers list
- devel@riot-os.org (http://lists.riot-os.org/mailman/listinfo/devel)
- RIOT OS users list
- users@riot-os.org (http://lists.riot-os.org/mailman/listinfo/users)
- RIOT commits
- commits@riot-os.org (http://lists.riot-os.org/mailman/listinfo/commits)
- Github notifications
- notifications@riot-os.org (http://lists.riot-os.org/mailman/listinfo/notifications)

CHAPTER 5

LICENSE

- All sources and binaries that have been developed at Freie Universität Berlin are licensed under the GNU Lesser General Public License version 2.1 as published by the Free Software Foundation.
- Some external sources, especially files developed by SICS are published under a separate license.

All code files contain licensing information.

For more information, see the RIOT website:

http://www.riot-os.org

CHAPTER 6

CONTRIBUTE

This is a short version of the Development Procedures.

1. Check if your code follows the coding conventions. If the code does not comply these style rules, your code will not be merged.
2. The master branch should always be in a working state. The RIOT maintainers will create release tags based on this branch, whenever a milestone is completed.
3. Comments on a pull request should be added to the request itself, and *not* to the commit.
4. Keep commits to the point, e.g., don't add whitespace/typo fixes to other code changes. If changes are layered, layer the patches.
5. Describe the technical detail of the change(s) as specific as possible.
6. Use Labels to help classify pull requests and issues.

CHAPTER 7

Platform configurations for RIOT-OS

This directory contains existing configuration and initialization files for platforms supported by RIOT-OS.

CHAPTER 8

Mulle OpenOCD configuration files

The configuration file in this directory has been tested with OpenOCD v0.7.0. The interface used is ftdi, OpenOCD must be built with –enable-ftdi

To start the OpenOCD GDB server:

openocd -f mulle.cfg

CHAPTER 9

Zolertia Re-Mote platform

The Re-Mote platform is a IoT Hardware development platform based on TI's CC2538 system on chip (SoC), featuring an ARM Cortex-M3 with 512KB flash, 32Kb RAM, double RF interface, and the following goodies:

- ISM 2.4-GHz IEEE 802.15.4 & Zigbee compliant.
- ISM 868-, 915-, 920-, 950-MHz ISM/SRD Band.
- AES-128/256, SHA2 Hardware Encryption Engine.
- ECC-128/256, RSA Hardware Acceleration Engine for Secure Key Exchange.
- Power consumption down to 3uA using our shutdown mode.
- Co-Processor to allow peripheral management, programming over BSL without requiring to press any button to enter bootloader mode.
- Built-in battery charger (500mA), Energy Harvesting and Solar Panels to be connected to standards LiPo batteries.
- Power input with wide range 2-26VDC.
- Built-in TMP102 temperature sensor
- Small form-factor (as the Z1 mote, half the size of an Arduino) 57x35 mm.

CHAPTER 10

Port Features

In terms of hardware support, the following drivers have been implemented:

- CC2538 System-on-Chip:
  - UART
  - Random number generator
  - Low Power Modes
  - General-Purpose Timers.
  - ADC
  - LEDs
  - Buttons
  - Internal/external 2.4GHz antenna switch controllable by SW.

And under work or pending at cc2538 base cpu:

```
* RF 2.4GHz built-in in CC2538 (PR #2198)
* SPI/I2C library
* Built-in core temperature and battery sensor.
* TMP102 temperature sensor driver.
* CC1120 sub-1GHz radio interface.
* Micro-SD external storage.
* USB (in CDC-ACM).
* uDMA Controller.
```

CHAPTER 11

Requirements

- Toolchain to compile RIOT for the CC2538
- Drivers to enable your host to communicate with the platform
- Built-in BSL programming over USB using cc2538-bsl (included)

11.1 Install a Toolchain

The toolchain used to build is arm-gcc, to check if it is currently installed run:

```
$ arm-none-eabi-gcc -v
Using built-in specs.
Target: arm-none-eabi
Configured with: /scratch/julian/lite-respin/eabi/src/gcc-4.3/configure
...
(skip)
...
Thread model: single
gcc version 4.3.2 (Sourcery G++ Lite 2008q3-66)
```

Else install from https://launchpad.net/gcc-arm-embedded

11.2 Drivers

The Re-Mote features a FTDI serial-to-USB module, the driver is commonly found in most OS, but if required it can be downloaded from http://www.ftdichip.com/Drivers/VCP.htm

11.2.1 For the CC2538EM (USB CDC-ACM)

The Re-Mote has built-in support for USB 2.0 USB, Vendor and Product IDs are the following:

- VID 0x0451
- PID 0x16C8

On Linux and OS X this is straightforward, on windows you need to install the following driver:

https://github.com/alignan/lufa/blob/remote-zongle/LUFA/CodeTemplates/WindowsINF/LUFA%20CDC-ACM.inf

And replace the IDs accordingly.

11.2.2 Device Enumerations

For the UART, serial line settings are 115200 8N1, no flow control.

Once all drivers have been installed correctly:

On windows, devices will appear as a virtual `COM` port.

On Linux and OS X, devices will appear under `/dev/`.

On OS X:

- XDS backchannel: `tty.usbserial-<serial number>`
- EM in CDC-ACM: `tty.usbmodemf<X><ABC>` (X a letter, ABC a number e.g. `tty.usbmodemfd121`)

On Linux:

- Re-Mote over FTDI: `ttyUSB1`
- Re-Mote over USB driver (in CDC-ACM): `ttyACMn` (n=0, 1, ...)

CHAPTER 12

More Reading

1. Zolertia Re-Mote website
2. CC2538 System-on-Chip Solution for 2.4-GHz IEEE 802.15.4 and ZigBee applications (SWRU319B)
3. CC1120 sub-1GHz RF transceiver

CHAPTER 13

K60 tools

This directory contains tools for working with K60 CPUs.

13.1 Watchdog disable

wdog-disable.bin is a location-independent watchdog disable function with a breakpoint instruction at the end. Useful for disabling the watchdog directly from OpenOCD.

Usage:

```
openocd -c `reset halt' \
  -c `load_image wdog-disable.bin 0x20000000 bin' \
  -c `resume 0x20000000' # watchdog is disabled and core halted
```

CHAPTER 14

Valgrind Support

Rebuild your application using the all-valgrind target like this:

```
make -B clean all-valgrind
```

That way native will tell Valgrind about RIOT's stacks and prevent Valgrind from reporting lots of false positives. The debug information flag `-g` is not strictly necessary, but passing it allows Valgrind to tell you precisely which code triggered the error.

To run your application run:

```
make term-valgrind
```

All this does is run your application under Valgrind. Now Valgrind will print some information whenever it detects an invalid memory access.

In order to debug the program when this occurs you can pass the –db-attach parameter to Valgrind. E.g:

```
valgrind --db-attach=yes ./bin/native/default.elf tap0
```

Now, you will be asked whether you would like to attach the running process to gdb whenever a problem occurs.

In order for this to work under Linux 3.4 or newer, you might need to disable the ptrace access restrictions: As root call:

```
echo 0 > /proc/sys/kernel/yama/ptrace_scope
```

CHAPTER 15

Network Support

If you compile RIOT for the native cpu and include the `native_net` module, you need to specify a network interface like this:

```
make term PORT=tap0
```

CHAPTER 16

Setting Up A Tap Network

There is a shellscript in RIOT/dist/tools/tapsetup called `tapsetup` which you can use to create a network of tap interfaces.

Usage: To create a bridge and two (or count at your option) tap interfaces:

```
../../dist/tools/tapsetup/tapsetup [-c [<count>]]
```

On OSX you need to start the RIOT instance at some point during the script's execution. The script will instruct you when to do that.

To delete the bridge and all tap interfaces:

```
../../dist/tools/tapsetup/tapsetup -d
```

For OSX you **have** to run this after killing your RIOT instance and rerun `../../dist/tools/tapsetup [-c [<count>]]` before restarting.

CHAPTER 17

Daemonization

You can daemonize a riot process. This is useful for larger networks. Valgrind will fork along with the riot process and dump its output in the terminal.

Usage:

```
./bin/native/default.elf -d
```

CHAPTER 18

Compile Time Options

Compile with

```
CFLAGS=-DNATIVE_AUTO_EXIT make
```

to exit the riot core after the last thread has exited.

CHAPTER 19

RIOT integration into IoT-LAB

Check the Wiki to see how to build and run RIOT on FIT IoT-LAB: https://github.com/iot-lab/iot-lab/wiki/Riot-support

19.1 Control IoT-LAB via Make

19.1.1 Requirements

This feature requires to have a valid account for the FIT IoT-LAB (registration there is open for everyone) and the iot-lab/cli-tools to be installed.

19.1.2 Description

The folder `dist/testbed-support/` contains a `Makefile.iotlab` that defines some targets to control RIOT experiments on IoT-LAB using the GNU Make build system. In order to use this, one has to include this Makefile at the end of the application's Makefile, like this:

```
include $(RIOTBASE)/dist/testbed-support/Makefile.iotlab
```

19.1.3 Variables

This Makefile introduces some additional variables (default values in brackets):

- IOTLAB_NODES (5)
- IOTLAB_DURATION (30 minutes)
- IOTLAB_SITE (grenoble.iot-lab.info)
- IOTLAB_TYPE (m3:at86rf231)
- IOTLAB_AUTH ($HOME/.iotlabrc)
- IOTLAB_USER (taken from $IOTLAB_AUTH)
- IOTLAB_EXP_ID (taken from first experiment in running state)
- IOTLAB_EXP_NAME (RIOT_EXP)
- IOTLAB_PHY_NODES
- IOTLAB_EXCLUDE_NODES

19.1.4 Format of a Resource ID

Both variables `IOTLAB_PHY_NODES` and `IOTLAB_EXCLUDE_NODES` use the resource id string format as specified in the output of `experiment-cli submit --help`. An example would be: 1-3+7+10-13

19.1.5 Targets

It defines the following targets:

- iotlab-exp
- iotlab-flash
- iotlab-reset
- iotlab-term

Please note: All targets that require an already running experiment will use the first experiment of the user that has already entered state "Running" if `IOTLAB_EXP_ID` is not set.

iotlab-exp

This schedules a new experiment on the FIT IoT-LAB and waits until it enters "Running" state. It will request `IOTLAB_NODES` nodes of type `IOTLAB_TYPE` for `IOTLAB_DURATION` minutes at site `IOTLAB_SITE`. With `IOTLAB_PHY_NODES` it is possible to choose specific nodes for this experiment by using the resource id string format as described above. Note that the usage of `IOTLAB_PHY_NODES` ignores `IOTLAB_NODES`. It will also flash the binary of the current application to all registered nodes. The name of the experiment is set to "RIOT_EXP" or "RIOT_EXP_$(IOTLAB_EXP_NAME)" if `IOTLAB_EXP_NAME` is defined.

iotlab-flash

This target updates the application on all registered nodes of the given experiment to the current version of the application. Certain nodes can be excluded by listing them in the `IOTLAB_EXCLUDE_NODES` variable using the resource id string format as described above. If you do not use the default site, then you must specify the site with `IOTLAB_SITE`.

iotlab-reset

This target resets all registered nodes of the given experiment. Certain nodes can be excluded by listing them in the `IOTLAB_EXCLUDE_NODES` variable using the resource id string format as described above. If you do not use the default site, then you must specify the site with `IOTLAB_SITE`.

iotlab-term

Uses ssh to login the user on the IoT-LAB server of the specified site and start the `serial_aggregator` to communication with all registered nodes.

CHAPTER 20

About

This is a cppcheck wrapper script with appropriate parameters for checking RIOT. It accepts a branch name as an argument which is used to limit the scope of the check. Other parameters will be passed to cppcheck, so you can further modify its behavior.

CHAPTER 21

Example usage

Check all files changed in the current branch against the branch named ‘master’:

```
./dist/tools/cppcheck/check.sh master
```

Check all files but ignore warnings about unused struct members:

```
./dist/tools/cppcheck/check.sh --suppress=unassignedVariable
```

Check all files changed in the current branch against the branch named ‘master’, ignoring warnings about unassigned variables:

```
./dist/tools/cppcheck/check.sh master  --suppress=unassignedVariable
```

CHAPTER 22

Default options

This script suppresses warnings of the type “unusedStructMember” by default. If you want to get warnings about “unusedStructMembers” run the script with the –show-unused-struct option: ./dist/tools/cppcheck/check.sh –show-unused-struct [BRANCH] [options to be passed]

CHAPTER 23

What to do about the findings

You should read the code carefully. While cppcheck certainly produces valuable information, it can also warn about code that is actually OK. If this happens, you can add an "inline suppression" like this:

```
/* cppcheck-suppress passedByValue */
timex_t timex_add(const timex_t a, const timex_t b);
```

CHAPTER 24

cmdline2xml.sh

Export all command line include paths and macro definitions to an XML file suitable for import in Eclipse CDT.

24.1 Instrucions

The Eclipse project must be located at "/RIOT" inside your Eclipse workspace, otherwise change cmdline2xml.sh accordingly (ECLIPSE_PROJECT_NAME=RIOT).

In the shell:

```
cd to application directory (e.g. examples/hello-world)
make eclipsesym
```

In Eclipse:

1. Open the project properties, menu Project->Properties
2. Select C/C++ General->Paths and Symbols
3. (optional) Click Restore Defaults to delete any existing macros and include paths
4. Click Import Settings...
5. Select `eclipsesym.xml` in your application directory and press Finish
6. Rebuild C/C++ index, menu Project->C/C++ Index->Rebuild

All conditional compilation and all include paths should now resolve properly for your application.

The file `eclipsesym.xml` is specific to the application being built and may differ depending on what modules are enabled and which platform is being built. Make sure that everything is set up properly in your shell and that regular `make all` works before running `make eclipsesym`

CHAPTER 25

About

This script checks if a Pull Request needs squashing or if it is waiting for another Pull Request.

CHAPTER 26

Usage

```
./pr_check.sh [<master branch>]
```

The optional `<master branch>` parameter refers to the branch the pull request's branch branched from. The script will output all commits marked as squashable from `HEAD` to the merge-base with `<master branch>`. The default for `<master branch>` is `master`.

A commit is marked as squashable if it contains the keywords SQUASH or FIX (case insensitive) within the first five characters of it's subject title.

CHAPTER 27

RIOT Sniffer Application

27.1 About

This sniffer script can be used to sniff network traffic using RIOT based nodes. It is primarily designed for sniffing wireless data traffic, but can also well be used for wired network traffic, as long as the used network devices support promiscuous mode and output of raw data.

The sniffer is based on a RIOT node running the sniffer application application located in RIOTs application repository. This node outputs received network traffic via a serial port or a network socket in the Wireshark pcap format. This output is then parsed by the `sniffer.py` script included in this folder run on a host computer.

The `sniffer.py` script is a modified version of malvira's script for the Redbee Ecotag (https://github.com/malvira/libmc1322x/wiki/wireshark).

27.2 Dependencies

The `sniffer.py` script needs pyserial.

Installing the dependencies:

27.2.1 Debuntu

```
apt-get install python-serial
```

27.2.2 PIP

```
pip install pyserial
```

27.3 Usage

General usage:

1. Flash an applicable RIOT node with the sniffer application from (https://github.com/RIOT-OS/applications/tree/master/sniffer)
2. Run the `sniffer.py` script For serial port:

```
$ ./sniffer.py serial <tty> <baudrate> <channel> [outfile]
```

For network socket:

```
$ ./sniffer.py socket <host> <port> <channel> [outfile]
```

The script has the following parameters:

- **connType:** The type of connection to use. Either `serial` for serial ports or `socket` for network sockets.
- **host:** The host if the `socket` connection type is in use.
- **port:** The port of the host if the `socket` connection type is in use.
- **tty:** The serial port the RIOT board is connected to. Under Linux, this is typically something like /dev/ttyUSB0 or /dev/ttyACM0. Under Windows, this is typically something like COM0 or COM1. This option is used for the `serial` connection type.
- **baudrate:** The baudrate the serial port is configured to. The default in RIOT is 115200, though this is defined per board and some boards have some other value defined per default. NOTE: when sniffing networks where the on-air bitrate is > baudrate, it makes sense to increase the baudrate so no data is skipped when sniffing. This option is used for the `serial` connection type.
- **channel:** The radio channel to use when sniffing. Possible values vary and depend on the link-layer that is sniffed. This parameter is ignored when sniffing wired networks.
- **[outfile]:** When this parameter is specified, the sniffer output is saved into this file. See the examples below for alternatives to specifying this parameter. (optional)

27.3.1 Examples

The following examples are made when using the sniffer application together with an `iotlab-m3` node that is connected to /dev/ttyUSB1 (or COM1) (`serial` connection type) and runs per default with a baudrate of 500000. For the `socket` connection type port 20000 is used.

Linux (serial)

Dump packets to a file:

```
$ ./sniffer.py serial /dev/ttyUSB1 500000 17 > foo.pcap
```

This .pcap can then be opened in wireshark.

Alternatively for live captures, you can pipe directly into wireshark with:

```
$ ./sniffer.py serial /dev/ttyUSB1 500000 17 | wireshark -k -i -
```

Windows (serial)

For windows you can use the optional third argument to output to a .pcap:

```
$ ./sniffer.py serial COM1 500000 17 foo.pcap
```

IoT-Lab Testbed (socket)

Start an experiment either via the website provided by the IoT-Lab testbed or by using the RIOT specific iotlab Makefile with 3 neighboring `iotlab-m3` nodes, where one of them runs the sniffer application and the others run the `gnrc_networking` application.

Now you can bind the sniffer node to localhost: ssh -L 20000:*node-id*:20000 *user@site*.iot-lab.info

Then you can dump or observe the traffic generated by the other nodes running the `gnrc_networking` application via one of the following commands:

```
$ ./sniffer.py socket localhost 20000 26 > foo.pcap
$ ./sniffer.py socket localhost 20000 26 | wireshark -k -i -
```

CHAPTER 28

Creating a SLIP network interface

The module `gnrc_slip` (Serial line IP) enables the RIOT network stack to communicate IP packets over the serial interface. This collection of tools originally from Contiki [1] enables Linux to interpret this data. Though there is a tool for such operations on Linux (`slattach`) it is only able to handle IPv4 packages and is unnessarily complicated.

CHAPTER 29

Installation

Just install them using

```
make
sudo make install
```

By default they are installed to the `/usr/local/bin` directory, you can however change that by setting the PREFIX environment variable

```
export PREFIX=${HOME}/.local
make
sudo make install
```

CHAPTER 30

Usage

`tapslip6` allows you to open a TAP interface (includes link-layer data) for a serial interace handling IPv6 data, `tunslip` allows you to open a TUN interface (includes only network-layer data) for a serial interace handling IPv4 data, and `tunslip6` allows you to open a TUN interface (includes only network-layer data) for a serial interace handling IPv6 data.

For more information use the help feature of the tools

```
tapslip -h
tunslip -h
tunslip6 -h
```

[1] https://github.com/contiki-os/contiki/tree/a4206273a5a491949f9e565e343f31908173c998/tools

CHAPTER 31

USB to serial adapter tools

Tools for finding connected USB to serial adapter devices.

31.1 Usage

`./list-ttys.sh`

List all currently connected USB to serial adapters by searching through `/sys/bus/usb/devices/`.

`./find-tty.sh [serial_regex1] [serial_regex2] ... [serial_regexZ]`

Write to `stdout` the first tty connected to the chosen programmer. `serial_regexN` are extended regular expressions (as understood by `egrep`) containing a pattern matched against the USB device serial number. Each of the given expressions are tested, against each serial number until a match has been found.

In order to search for an exact match against the device serial, use '^serialnumber$' as the pattern. If no pattern is given, `find-tty.sh` returns the first found USB tty (in an arbitrary order, this is not guaranteed to be the `/dev/ttyUSBX` with the lowest number).

Serial strings from all connected USB ttys can be found from the list generated by `list-ttys.sh`.

31.2 Exit codes

`find-tty.sh` returns 0 if a match is found, 1 otherwise.

31.3 Makefile example usage

The script `find-tty.sh` is designed for use from within a board `Makefile.include`. An example section is shown below (for an OpenOCD based solution):

```
# Add serial matching command
ifneq ($(PROGRAMMER_SERIAL),)
  OOCD_BOARD_FLAGS += -c `ftdi_serial $(PROGRAMMER_SERIAL)'

  ifeq ($(PORT),)
    # try to find tty name by serial number, only works on Linux currently.
    ifeq ($(OS),Linux)
      PORT := $(shell $(RIOTBASE)/dist/tools/usb-serial/find-tty.sh ``^$(PROGRAMMER_SERIAL)
```

```
    endif
  endif
endif

# Fallback PORT if no serial was specified or if the specified serial was not found
ifeq ($(PORT),)
    ifeq ($(OS),Linux)
      PORT := $(shell $(RIOTBASE)/dist/tools/usb-serial/find-tty.sh)
    else ifeq ($(OS),Darwin)
      PORT := $(shell ls -1 /dev/tty.SLAB_USBtoUART* | head -n 1)
    endif
endif

# TODO: add support for windows as host platform
ifeq ($(PORT),)
  $(info CAUTION: No terminal port for your host system found!)
endif
export PORT
```

31.4 Limitations

Only tested on Linux, and probably only works on Linux.

Getting started {#getting-started}

[TOC]

CHAPTER 33

Downloading RIOT code {#downloading-riot-code}

You can obtain the latest RIOT code from our Github repository either by downloading the latest tarball or by cloning the git repository.

In order to clone the RIOT repository, you need the Git revision control system and run the following command:

```
git clone git://github.com/RIOT-OS/RIOT.git
```

CHAPTER 34

Compiling RIOT {#compiling-riot}

34.1 Setting up a toolchain {#setting-up-a-toolchain}

Depending on the hardware you want to use, you need to first install a corresponding toolchain. The Wiki on RIOT's Github page contains a lot of information that can help you with your platform:

- ARM-based platforms
- TI MSP430
- Atmel ATmega
- native

34.2 The build system {#the-build-system}

RIOT uses GNU make as build system. The simplest way to compile and link an application with RIOT, is to set up a Makefile providing at least the following variables:

- `APPLICATION`: should contain the (unique) name of your application
- `BOARD`: specifies the platform the application should be build for by default
- `RIOTBASE`: specifies the path to your copy of the RIOT repository (note, that you may want to use `$(CURDIR)` here, to give a relative path)

Additionally it has to include the `Makefile.include`, located in RIOT's root directory:

```
# a minimal application Makefile
APPLICATION = mini-makefile
BOARD ?= native
RIOTBASE ?= $(CURDIR)/../RIOT

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

You can use Make's `?=` operator in order to allow overwriting variables from the command line. For example, you can easily specify the target platform, using the sample Makefile, by invoking make like this:

```
make BOARD=iotlab-m3
```

Besides typical targets like `clean`, `all`, or `doc`, RIOT provides the special targets `flash` and `term` to invoke the configured flashing and terminal tools for the specified platform. These targets use the variable `PORT` for the serial communication to the device. Neither this variable nor the targets `flash` and `term` are mandatory for the native port.

For the native port, PORT has a special meaning: it is used to identify the tap interface if the netdev2_tap module is used. The target debug can be used to invoke a debugger on some platforms. For the native port the additional targets such as all-valgrind and valgrind exist. Refer to cpu/native/README.md for additional information

Some RIOT directories contain special Makefiles like Makefile.base, Makefile.include or Makefile.dep. The first one can be included into other Makefiles to define some standard targets. The files called Makefile.include are used in boards and cpu to append target specific information to variables like INCLUDES, setting the include paths. Makefile.dep serves to define dependencies.

Unless specified otherwise, make will create an elf-file as well as an Intel hex file in the bin folder of your application directory.

Learn more about the build system in the Wiki

34.3 Building and executing an examples {#building-and-executing-and-example}

RIOT provides a number of examples in the examples/ directory. Every example has a README that documents its usage and its purpose. You can build them by typing

```
make BOARD=samr21-xpro
```

or

```
make all BOARD=samr21-xpro
```

into your shell.

To flash the application to a board just type

```
make flash BOARD=samr21-xpro
```

You can then access the board via the serial interface:

```
make term BOARD=samr21-xpro
```

If you are using multiple boards you can use the PORT macro to specify the serial interface:

```
make term BOARD=samr21-xpro PORT=/dev/ttyACM1
```

Note that the PORT macro has a slightly different semantic in native. Here it is used to provide the name of the TAP interface you want to use for the virtualized networking capabilities of RIOT.

We use pyterm as the default terminal application. It is shipped with RIOT in the dist/tools/pyterm/ directory. If you choose to use another terminal program you can set TERMPROG (and if need be the TERMFLAGS) macros:

```
make -C examples/gnrc_networking/ term \
    BOARD=samr21-xpro \
    TERMPROG=gtkterm \
    TERMFLAGS=''-s 115200 -p /dev/ttyACM0 -e''
```

CHAPTER 35

RIOT Documentation {#mainpage}

[TOC]

CHAPTER 36

RIOT in a nutshell {#riot-in-a-nutshell}

RIOT is an open-source microkernel-based operating system, designed to match the requirements of Internet of Things (IoT) devices and other embedded devices. These requirements include a very low memory footprint (on the order of a few kilobytes), high energy efficiency, real-time capabilities, communication stacks for both wireless and wired networks, and support for a wide range of low-power hardware.

RIOT provides a microkernel, multiple network stacks, and utilities which include cryptographic libraries, data structures (bloom filters, hash tables, priority queues), a shell and more. RIOT supports a wide range of microcontroller architectures, radio drivers, sensors, and configurations for entire platforms, e.g. Atmel SAM R21 Xplained Pro, Zolertia Z1, STM32 Discovery Boards etc. (see the list of supported hardware. Across all supported hardware (32-bit, 16-bit, and 8-bit platforms). RIOT provides a consistent API and enables ANSI C and C++ application programming, with multithreading, IPC, system timers, mutexes etc.

CHAPTER 37

Contribute to RIOT {#contribute-to-riot}

RIOT is developed by an open community that anyone is welcome to join:

- Download and contribute your code on GitHub. You can read about how to contribute in our GitHub Wiki.
- Subscribe to users@riot-os.org to ask for help using RIOT or writing an application for RIOT (or to just stay in the loop). A searchable archive of this list is available at the RIOT user Gmane newsgroup
- [Subscribe]((http://lists.riot-os.org/mailman/listinfo/devel) to devel@riot-os.org to follow and discuss kernel and network stack developement, or hardware support. A searchable archive of this list is available at the RIOT devel Gmane newsgroup
- Follow us on Twitter for news from the RIOT community.
- Contact us on IRC for live support and discussions: irc.freenode.org #riot-os

CHAPTER 38

The quickest start {#the-quickest-start}

You can run RIOT on most IoT devices, on open-access testbed hardware (e.g. IoT-lab), and also directly as a process on your Linux/FreeBSD/OSX machine (we call this the `native` port). Try it right now in your terminal window:

```
git clone git://github.com/RIOT-OS/RIOT.git # assumption: git is pre-installed
git checkout <LATEST_RELEASE>
cd RIOT
./dist/tools/tapsetup/tapsetup              # create virtual Ethernet
                                            # interfaces to connect to RIOT
cd examples/default/
make all
make term
```

... and you are in the RIOT shell! Type `help` to discover available commands. For further information see the README of the `default` example.

To use RIOT directly on your embedded platform, and for more hands-on details with RIOT, see @ref getting-started.

Before that, skimming through the next section is recommended (but not mandatory).

CHAPTER 39

Structure {#structure}

This section walks you through RIOT's structure. Once you understand this structure, you will easily find your way around in RIOT's code base.

RIOT's code base is structured into five groups.

- The kernel (`core`)
- Platform specific code (`cpu`; `boards`)
- Device drivers (`drivers`)
- Libraries and network code (`sys`; `pkg`)
- Applications for demonstrating features and for testing (`examples`; `tests`)

In addition RIOT includes a collection of scripts for various tasks (`dist`) as well as a predefined environment for generating this documentation (`doc`).

The structural groups are projected onto the directory structure of RIOT, where each of these groups resides in one or two directories in the main RIOT directory.

The following list gives a more detailed description of each of RIOT's top-level directories:

39.1 core

This directory contains the actual kernel. The kernel consists of the scheduler, inter-process-communication (messaging), threading, thread synchronization, and supporting data-structures and type definitions.

See @ref core for further information and API documentations.

39.2 boards

The platform dependent code is split into two logic elements: CPUs and boards, while maintaining a strict 1-to-n relationship, a board has exactly one CPU, while a CPU can be part of n boards. The CPU part contains all generic, CPU specific code (see below).

The board part contains the specific configuration for the CPU it contains. This configuration mainly includes the peripheral configuration and pin-mapping, the configuration of on-board devices, and the CPU's clock configuration.

On top of the source and header files needed for each board, this directory additionally may include some script and configuration files needed for interfacing with the board. These are typically custom flash/debug scripts or e.g. OpenOCD configuration files. For most boards, these files are located in a `dist` sub-directory of the board.

See here @ref boards for further information.

39.3 cpu

For each supported CPU this directory contains a sub-directory with the name of the CPU. These directories then contain all CPU specific configurations, such as implementations of power management (LPM), interrupt handling and vectors, startup code, clock initialization code and thread handling (e.g. context switching) code. For most CPUs you will also find the linker scripts in the `ldscripts` sub-directory.

In the `periph` sub-directory of each CPU you can find the implementations of the CPU's peripheral drivers like SPI, UART, GPIO, etc. See @ref drivers_periph for their API documentation.

Many CPUs share a certain amount of their code (e.g. all ARM Cortex-M based CPUs share the same code for task switching and interrupt handling). This shared code is put in its own directories, following a `xxxxx_common` naming scheme. Examples for this is code shared across architectures (e.g. `cortexm_common`, `msp430_comon`) or code shared among vendors (e.g. `kinetis_common`).

See @ref cpu for more detailed informtation.

39.4 drivers

This directory contains the drivers for external devices such as network interfaces, sensors and actuators. Each device driver is put into its own sub-directory with the name of that device.

All of RIOT's device drivers are based on the peripheral driver API (e.g. SPI, GPIO, etc.) and other RIOT modules like the `xtimer`. This way the drivers are completely platform agnostic and they don't have any dependencies into the CPU and board code.

See @ref drivers for more details.

39.5 sys

RIOT follows the micro-kernel design paradigm where everything is supposed to be a module. All of these modules that are not part of the hardware abstraction nor device drivers can be found in this directory. The libraries include data structures (e.g. bloom, color), crypto libraries (e.g. hashes, AES) , high-level APIs (e.g. Posix implementations), memory management (e.g. malloc), the RIOT shell and many more.

See @ref sys for a complete list of available libraries

39.6 sys/net

The `sys/net` sub-directory needs to be explicitly mentioned, as this is where all the networking code in RIOT resides. Here you can find the network stack implementations (e.g. the GNRC stack) as well as network stack agnostic code as header definitions or network types.

See @ref net for more details on networking code.

39.7 pkg

RIOT comes with support for a number of external libraries (e.g. OpenWSN, microcoap). The way they are included is that RIOT ships with a custom Makefile for each supported library that downloads the library and optionally applies a number of patches to make it work with RIOT. These Makefiles and patches can be found in the `pkg` directory.

See @ref pkg for a detailed description on how this works.

39.8 examples

Here you find a number of example applications that demonstrate certain features of RIOT. The default example found in this directory is a good starting point for anyone who is new to RIOT.

For more information best browse that directory and have a look at the `README.md` files that ship with each example.

39.9 tests

Many features/modules in RIOT come with their own test application, which are located in this directory. In contrary to the examples these tests are mostly focusing on a single aspect than on a set of features. Despite for testing, you might consider these tests also for insights on understanding RIOT.

39.10 dist & doc

All the tooling around RIOT can be found in these two folders.

`doc` contains the doxygen configuration and also contains the compiled doxygen output after running `make doc`.

Lastly, the `dist` directory contains tools to help you with RIOT. These include the serial terminal application `pyterm`, generic scripts for flashing, debugging, reseting (e.g. support for OpenOCD, Jlink), as well as code enabling easy integration to open testbeds such as the IoT-LAB. Furthermore you can find here scripts to do all kind of code and style checks.

Idea for this section: just name each of RIOT's main features/concepts and link to an appropriate page with further information: - Create an application - Networking - The `main()` function - Make system - Include modules - Threading - Choose the right stack size - IPC - Auto initialization –>

Examples:

CHAPTER 40

examples/arduino_hello-world

This application demonstrates the usage of Arduino sketches in RIOT.

The sketch itself is fairly simple. On startup, it initializes the LED pin to output mode, starts the serial port with a baudrate of 115200 and prints "Hello Arduino!" on the serial port. When running, the application echoes any newline terminated string that was received on the serial port, while toggling the default LED with a 1Hz frequency.

The sketch just uses some very primitive Arduino API elements for demonstration purposes:

- control of digital pins (pinMode(), digital read and write)
- the delay() function
- reading and writing the serial port using the Serial class

CHAPTER 41

Arduino and RIOT

For information on the Arduino support in RIOT please refer to the API documentation at http://doc.riot-os.org/group__sys__arduino.html

CHAPTER 42

Usage

Just send any newline terminated string to the board's serial port, and the board will echo this string.

CHAPTER 43

Example output

When using pyterm, the output will look similar to this:

```
2015-11-26 14:04:58,307 - INFO # main(): This is RIOT! (Version: xxx)
2015-11-26 14:04:58,308 - INFO # Hello Arduino!
hello again
2015-11-26 14:06:29,800 - INFO # Echo: hello again
are you still there?
2015-11-26 14:06:48,301 - INFO # Echo: are you still there?
```

If your board is equipped with an on-board LED, you should see this LED toggling every half a second.

NOTE: if your board's STDIO baudrate is not configured to be 115200 (see your board's `board.h`), the first line of the output may not be shown or scrambled.

CHAPTER 44

examples/default

This application is a showcase for RIOT's hardware support. Using it for your board, you should be able to interactively use any hardware that is supported.

To do this, the application uses the `shell` and `shell_commands` modules and all the driver modules each board supports.

`shell` is a very simple interactive command interpreter that can be used to call functions. Many of RIOT's modules define some generic shell commands. These are included via the `shell_commands` module.

Additionally, the `ps` module which provides the `ps` shell command is included.

CHAPTER 45

Usage

Build, flash and start the application:

```
export BOARD=your_board
make
make flash
make term
```

The `term` make target starts a terminal emulator for your board. It connects to a default port so you can interact with the shell, usually that is `/dev/ttyUSB0`. If your port is named differently, the `PORT=/dev/yourport` variable can be used to override this.

CHAPTER 46

Example output

The shell commands come with online help. Call `help` to see which commands exist and what they do.

Running the `help` command on an iotlab-m3:

```
2015-09-16 16:57:17,723 - INFO # help
2015-09-16 16:57:17,725 - INFO # Command              Description
2015-09-16 16:57:17,726 - INFO # ---------------------------------------
2015-09-16 16:57:17,727 - INFO # reboot               Reboot the node
2015-09-16 16:57:17,729 - INFO # ps                   Prints information about running thre
2015-09-16 16:57:17,731 - INFO # isl29020_init        Initializes the isl29020 sensor drive
2015-09-16 16:57:17,733 - INFO # isl29020_read        Prints data from the isl29020 sensor
2015-09-16 16:57:17,735 - INFO # lps331ap_init        Initializes the lps331ap sensor drive
2015-09-16 16:57:17,737 - INFO # lps331ap_read        Prints data from the lps331ap sensor
2015-09-16 16:57:17,739 - INFO # l3g4200d_init        Initializes the l3g4200d sensor drive
2015-09-16 16:57:17,740 - INFO # l3g4200d_read        Prints data from the l3g4200d sensor
2015-09-16 16:57:17,742 - INFO # lsm303dlhc_init      Initializes the lsm303dlhc sensor dri
2015-09-16 16:57:17,744 - INFO # lsm303dlhc_read      Prints data from the lsm303dlhc senso
2015-09-16 16:57:17,746 - INFO # ifconfig             Configure network interfaces
2015-09-16 16:57:17,747 - INFO # txtsnd               send raw data
```

Running the `ps` command on an iotlab-m3:

```
2015-09-16 16:57:57,634 - INFO # ps
2015-09-16 16:57:57,637 - INFO #    pid | name                | state    Q | pri | stack
2015-09-16 16:57:57,640 - INFO #      1 | idle                | pending  Q |  15 |   256
2015-09-16 16:57:57,642 - INFO #      2 | main                | pending  Q |   7 |  1536
2015-09-16 16:57:57,645 - INFO #      3 | pktdump             | bl rx    _ |   6 |  1536
2015-09-16 16:57:57,647 - INFO #      4 | at86rfxx            | bl rx    _ |   3 |  1024
2015-09-16 16:57:57,649 - INFO #        | SUM                 |            |     |  4352
```

CHAPTER 47

RIOT specific

The `ps` command is used to analyze the thread's state and memory status.

CHAPTER 48

Networking

The `ifconfig` command will help you to configure all available network interfaces. On an iolab-m3 it will print something like:

```
2015-09-16 16:58:37,762 - INFO # ifconfig
2015-09-16 16:58:37,766 - INFO # Iface  4   HWaddr: 9e:72  Channel: 26  NID: 0x23  TX-Powe
2015-09-16 16:58:37,768 - INFO #            Long HWaddr: 36:32:48:33:46:da:9e:72
2015-09-16 16:58:37,769 - INFO #            AUTOACK  CSMA
2015-09-16 16:58:37,770 - INFO #            Source address length: 2
```

Type `ifconfig help` to get an online help for all available options (e.g. setting the radio channel via `ifconfig 4 set chan 12`).

The `txtsnd` command allows you to send a simple string directly over the link layer using unicast or broadcast. The application will also automatically print information about any received packet over the serial. This will look like:

```
2015-09-16 16:59:29,187 - INFO # PKTDUMP: data received:
2015-09-16 16:59:29,189 - INFO # ~~ SNIP  0 - size:  28 byte, type:
NETTYPE_UNDEF (0)
2015-09-16 16:59:29,190 - INFO # 000000 7b 3b 3a 02 85 00 e7 fb 00 00 00 00 01
02 5a 55
2015-09-16 16:59:29,192 - INFO # 000010 40 42 3e 62 f2 1a 00 00 00 00 00 00
2015-09-16 16:59:29,194 - INFO # ~~ SNIP  1 - size:  18 byte, type:
NETTYPE_NETIF (-1)
2015-09-16 16:59:29,195 - INFO # if_pid: 4  rssi: 49  lqi: 78
2015-09-16 16:59:29,196 - INFO # src_l2addr: 5a:55:40:42:3e:62:f2:1a
2015-09-16 16:59:29,197 - INFO # dst_l2addr: ff:ff
2015-09-16 16:59:29,198 - INFO # ~~ PKT    -  2 snips, total size:  46 byte
```

CHAPTER 49

gnrc_networking_border_router example

49.1 Requirements

In order to setup a 6LoWPAN border router on RIOT, you need either a board that offers an IPv6 capable network interface (e.g. the `encx24j600` Ethernet chip) or connect it over the serial interface to a Linux host and use the SLIP standard [1]. The example application in this folder assumes as a default to be run on an Atmel SAM R21 Xplained Pro evaluation board using an external UART adapter for the second serial interface. However, it is feasible to run the example on any RIOT supported platform that offers either more than one UART or be equipped with an IPv6 capable network device. In this case only the Makefile of this application has to be slightly modified, e.g. by replacing the line

```
USEMODULE += gnrc_slip
```

with something like

```
USEMODULE += encx24j600
```

and specify the target platform as `BOARD = myplatform`. In order to use the border router over SLIP, please check the `periph_conf.h` of the corresponding board and look out for the `UART_NUMOF` parameter. Its value has to be bigger than 1.

49.2 Configuration

In order to connect a RIOT 6LoWPAN border router over SLIP you run a small program called tunslip (imported from Contiki) [2] on the Linux host. The program can be found in the `dist/tools/tunslip` folder and has to be compiled before first use (simple calling `make` should be enough). Now, one can start the program by calling something like:

```
cd dist/tools/tunslip
make
sudo ./tunslip6 affe::1/64 -t tun0 -s /dev/ttyUSB0
```

Assuming that `/dev/ttyUSB0` is the device descriptor for the (additional) UART interface of your RIOT board.

On the RIOT side you have to configure the SLIP interface by configuring a corresponding IPv6 address, e.g.

```
ifconfig 6 add affe::2
```

and adding the SLIP interface to the neighbor cache (because Linux won't respond to neighbor solicitations on an interface without a link-layer address) by calling

```
ncache add 6 affe::1
```

After this you're basically done and should be able to ping between the border router and the outside world (assuming that the Linux host is properly forwarding your traffic).

Additionally, you can configure IPv6 addresses on the 6LoWPAN interface for communication with other 6LoWPAN nodes. See also the `gnrc_networking` example for further help.

[1] https://tools.ietf.org/html/rfc1055

[2] https://github.com/contiki-os/contiki/blob/master/tools/tunslip.c

CHAPTER 50

gnrc_networking example

50.1 Connecting RIOT native and the Linux host

> **Note:** RIOT does not support IPv4, so you need to stick to IPv6 anytime. To establish a connection between RIOT and the Linux host, you will need `netcat` (with IPv6 support). Ubuntu 14.04 comes with netcat IPv6 support pre-installed. On Debian it's available in the package `netcat-openbsd`. Be aware that many programs require you to add an option such as -6 to tell them to use IPv6, otherwise they will fail. If you're using a *Raspberry Pi*, run `sudo modprobe ipv6` before trying this example, because raspbian does not load the IPv6 module automatically. On some systems (openSUSE for example), the *firewall* may interfere, and prevent some packets to arrive at the application (they will however show up in Wireshark, which can be confusing). So be sure to adjust your firewall rules, or turn it off (who needs security anyway).

First, create a tap interface (to which RIOT will connect) and a bridge (to which Linux will connect):

```
sudo ip tuntap add tap0 mode tap user ${USER}
sudo ip link set tap0 up
```

Now you can start the `gnrc_networking` example by invoking `make term`. This should automatically connect to the `tap0` interface. If this doesn't work for some reason, run `make` without any arguments, and then run the binary manually like so (assuming you are in the `examples/gnrc_networking` directory):

To verify that there is connectivity between RIOT and Linux, go to the RIOT console and run `ifconfig`:

```
> ifconfig
Iface  7   HWaddr: ce:f5:e1:c5:f7:5a
inet6 addr: ff02::1/128  scope: local [multicast]
inet6 addr: fe80::ccf5:e1ff:fec5:f75a/64  scope: local
inet6 addr: ff02::1:ffc5:f75a/128  scope: local [multicast]
```

Copy the link-local address of the RIOT node (prefixed with `fe80`) and try to ping it **from the Linux node**:

```
ping6 fe80::ccf5:e1ff:fec5:f75a%tap0
```

Note that the interface on which to send the ping needs to be appended to the IPv6 address, `%tap0` in the above example. When talking to the RIOT node, you always want to send to/receive from the `tap0` interface.

If the pings succeed you can go on to send UDP packets. To do that, first start a UDP server on the RIOT node:

```
> udp server start 8808
Success: started UDP server on port 8808
```

Now, on the Linux host, you can run netcat to connect with RIOT's UDP server:

```
nc -6uv fe80::ccf5:e1ff:fec5:f75a%tap0 8808
```

The `-6` option is necessary to tell netcat to use IPv6 only, the `-u` option tells it to use UDP only, and the `-v` option makes it give more verbose output (this one is optional).

You should now see that UDP messages are received on the RIOT side. Opening a UDP server on the Linux side is also possible. Do do that, write down the IP address of the host (run on Linux):

```
ifconfig tap0
tap0     Link encap:Ethernet  HWaddr ce:f5:e1:c5:f7:59
         inet6 addr: fe80::4049:5fff:fe17:b3ae/64 Scope:Link
         UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
         RX packets:6 errors:0 dropped:0 overruns:0 frame:0
         TX packets:36 errors:0 dropped:0 overruns:0 carrier:0
         collisions:0 txqueuelen:0
         RX bytes:488 (488.0 B)  TX bytes:3517 (3.5 KB)
```

Then open a UDP server on Linux (the `-l` option makes netcat listen for incoming connections):

```
nc -6ul 8808
```

Now, on the RIOT side, send a UDP packet using:

```
udp send fe80::4049:5fff:fe17:b3ae 8808 testmessage
```

You should see `testmessage` appear in netcat. Instead of using netcat, you can of course write your own software, but you may have to bind the socket to a specific interface (tap0 in this case). For an example that shows how to do so, see here.

CHAPTER 51

Hello World!

This is a basic example how to use RIOT in your embedded application. It prints out the famous text `Hello World!`.

This example should foremost give you an overview how to use the Makefile system:

- First you must give your application a name, which is commonly the same as the name of the directory it resides in. Then you can define a default BOARD for which the application was written. By using e.g. `make BOARD=msba2` you can override the default board. With `make buildtest` the application gets compiled for all supported boards.
- The variable `RIOTBASE` contains an absolute or relative path to the directory where you have checked out RIOT. If your code resides in a subdirectory of RIOT, then you can use `$(CURDIR)` as it's done in here.
- The variable `QUIET`, which is either `1` or `0`, defines whether to print verbose compile information, or hide them, respectively.
- The last line of your Makefile must be `include $(RIOTBASE)/Makefile.include`.

The code itself may look like your usual *C* beginners hello-world example.

CHAPTER 52

IPC Pingpong!

This example is to illustrate the usage of RIOTs IPC messaging system.

The application starts a second thread (in addition to the main thread) and sends messages between these two threads. The main thread calls `thread_send_receive()` in an endless loop. The second thread receives the message, prints `2nd: got msg from x` to stdout and sends a reply message with an incrementing number back to the main thread. The main thread then prints the number it received from the 2nd thread.

The correct output should look like this:

```
This is RIOT! (Version: xxx)
kernel_init(): jumping into first task...
Starting IPC Ping-pong example...
1st thread started, pid: 1
2nd thread started, pid: 2
2nd: Got msg from 1
1st: Got msg with content 2
2nd: Got msg from 1
1st: Got msg with content 3
2nd: Got msg from 1
1st: Got msg with content 4
2nd: Got msg from 1
1st: Got msg with content 5
2nd: Got msg from 1
1st: Got msg with content 6
2nd: Got msg from 1
1st: Got msg with content 7
2nd: Got msg from 1
1st: Got msg with content 8
2nd: Got msg from 1
1st: Got msg with content 9
2nd: Got msg from 1
1st: Got msg with content 10
[...]
```

CHAPTER 53

examples/posix_sockets

This application is a showcase for RIOT's POSIX socket support. To keep things simple this application has only one-hop support and no routing capabilities.

CHAPTER 54

Usage

Build, flash and start the application:

```
export BOARD=your_board
make
make flash
make term
```

The `term` make target starts a terminal emulator for your board. It connects to a default port so you can interact with the shell, usually that is `/dev/ttyUSB0`. If your port is named differently, the `PORT=/dev/yourport` (not to be confused with the UDP port) variable can be used to override this.

CHAPTER 55

Example output

The shell commands come with online help. Call `help` to see which commands exist and what they do.

udp send fe80::1 1337 uiaeue 2015-09-22 14:55:30,686 - INFO # > udp send fe80::1 1337 uiaeue 2015-09-22 14:55:30,690 - INFO # Success: send 6 byte to fe80::1:1337

Running the `help` command on an iotlab-m3:

```
2015-09-22 14:54:54,442 - INFO # help
2015-09-22 14:54:54,443 - INFO # Command              Description
2015-09-22 14:54:54,444 - INFO # ---------------------------------------
2015-09-22 14:54:54,446 - INFO # udp                  send data over UDP and listen on UDP
2015-09-22 14:54:54,447 - INFO # reboot               Reboot the node
2015-09-22 14:54:54,449 - INFO # ps                   Prints information about running thre
2015-09-22 14:54:54,451 - INFO # mersenne_init        initializes the PRNG
2015-09-22 14:54:54,453 - INFO # mersenne_get         returns 32 bit of pseudo randomness
2015-09-22 14:54:54,454 - INFO # ifconfig             Configure network interfaces
2015-09-22 14:54:54,455 - INFO # txtsnd               send raw data
2015-09-22 14:54:54,457 - INFO # ncache               manage neighbor cache by hand
2015-09-22 14:54:54,459 - INFO # routers              IPv6 default router list
```

Running the `ps` command on an iotlab-m3:

```
2015-09-22 14:54:57,134 - INFO # > ps
2015-09-22 14:54:57,139 - INFO #    pid | name                 | state    Q | pri | stack
2015-09-22 14:54:57,143 - INFO #      1 | idle                 | pending  Q |  15 |   256
2015-09-22 14:54:57,157 - INFO #      2 | main                 | pending  Q |   7 |  1536
2015-09-22 14:54:57,164 - INFO #      3 | 6lo                  | bl rx    _ |   3 |  1024
2015-09-22 14:54:57,169 - INFO #      4 | ipv6                 | bl rx    _ |   4 |  1024
2015-09-22 14:54:57,172 - INFO #      5 | udp                  | bl rx    _ |   5 |  1024
2015-09-22 14:54:57,177 - INFO #      6 | at86rfxx             | bl rx    _ |   3 |  1024
2015-09-22 14:54:57,183 - INFO #        | SUM                  |            |     |  5888
```

Start a UDP server with `udp server start <udp_port>`:

```
2015-09-22 14:55:09,563 - INFO # > udp server start 1337
2015-09-22 14:55:09,564 - INFO # Success: started UDP server on port 1337
```

Send a UDP package with `udp send <dst_addr> <dst_port> <data>`:

```
2015-09-22 14:55:30,686 - INFO # > udp send fe80::3432:4833:46d4:9e06 1337 test
2015-09-22 14:55:30,690 - INFO # Success: send 4 byte to [fe80::3432:4833:46d4:9e06]:1337
```

You can get the IPv6 address of the destination by using the `ifconfig` command on the receiver:

```
2015-09-22 14:58:10,394 - INFO # ifconfig
2015-09-22 14:58:10,397 - INFO # Iface  6   HWaddr: 9e:06  Channel: 26  NID: 0x23  TX-Powe
2015-09-22 14:58:10,399 - INFO #            Long HWaddr: 36:32:48:33:46:d4:9e:06
2015-09-22 14:58:10,400 - INFO #            AUTOACK  CSMA  MTU:1280  6LO  IPHC
2015-09-22 14:58:10,402 - INFO #            Source address length: 8
2015-09-22 14:58:10,404 - INFO #            Link type: wireless
2015-09-22 14:58:10,407 - INFO #            inet6 addr: ff02::1/128  scope: local [multicas
2015-09-22 14:58:10,415 - INFO #            inet6 addr: fe80::3432:4833:46d4:9e06/64  scope
2015-09-22 14:58:10,416 - INFO #
```

CHAPTER 56

Using C++ and C in a program with RIOT

This project demonstrates how user can use both C++ and C in their application with RIOT.

56.1 Makefile Options

- CXXEXFLAGS : user's extra flags used to build c++ files should be defined here (e.g -std=gnu++11).

This directory provides some porting information for libraries and programs to use with RIOT (to build an external module). If you'd like to add a package to RIOT you need to add a directory with the name of your package to this directory. Your directory should contain at least two files:

- **One or more patch files** - Your patches of the upstream application of the package to make it build with RIOT.
- **Makefile**- A Makefile describing how to get the upstream application, apply the patch and how to build the package as a RIOT module. A rough template for several methods of acquiring a package is provided in Makefile.git, Makefile.http, and Makefile.svn

CHAPTER 57

Creating a patch with git

Assuming your upstream application resides in a git repository, you can create the patch files as follows:

- checkout the targeted version of the upstream application
- conduct necessary changes (e.g. edit, add, or remove some files)
- commit your changes using `git commit`
- create the patch files using `git format-patch -n HEAD~N` where `N` is the number of commits you did
- move the resulting patch files to the corresponding subfolder of pkg

Packages are included to your application as external modules. Thus you only have to add the following line to your application (and update your INCLUDE path accordingly):

```
USEPKG += <pkg_name>
```

Since there is no official public repository for the CMSIS-DSP library, we are using our own repo.

CHAPTER 58

OpenWSN on RIOT

This port of OpenWSN to RIOT is based on current OpenWSN upstream providing a BSP with RIOT's interfaces. Currently supported are iotlab-m3 and fox. More boards will follow through improvements in netdev radio driver interface.

CHAPTER 59

Usage

A test can be found in tests/openwsn providing a shell command to initialise as root or listening node. And providing a sample Makefile.

Build using

```
$> export BOARD=iotlab-m3
$> export PORT=/dev/ttyTHEPORTOFYOURIOTLAB
$> make -B clean flash term
```

To use OpenWSN with RIOT it has to be added to the used packages variable

```
USEPKG += openwsn
```

On the first build the archive will be fetched, patched and built. **WARNING** A call of `make clean` also cleans the OpenWSN tree right now so changes to the source code will be lost in the next build.

CHAPTER 60

About

This is a manual test application for the ADT7310 temperature sensor driver.

Usage

This test application will initialize the sensor with the following parameters:

- Mode: 1 SPS
- Resolution: 16 bit

After initialization, the sensor reads the acceleration values every second and prints them to the STDOUT.

CHAPTER 62

About

This is a manual test application for the AT86RF2xx radio driver

For running this test, you need to connect/configure the following pins of your radio device:

- SPI MISO
- SPI MOSI
- SPI CLK
- CS (ship select)
- RESET
- SLEEP
- INT (external interrupt)

CHAPTER 63

Usage

For testing the radio driver you can use the netif and txtsnd shell commands that are included in this application.

CHAPTER 64

About

This is a manual test application for the HDC1000 driver.

Usage

This test application will initialize the HDC1000 sensor with the following parameters:

- Temperature and humidity are acquired in sequence
- Temperature measurement resolution 14 bit
- Humidity measurement resolution 14 bit

After initialization, the sensor reads the temperature and humidity values every 1s and prints them to STDOUT.

CHAPTER 66

About

This is a manual test application for the HIH6130 humidity and temperature sensor.

Usage

After initialization, the sensor reads the measurement values every 100ms and prints them to the STDOUT.

CHAPTER 68

About

This is a manual test application for the INA220 current and power monitor driver.

Usage

This test application will initialize the sensor with the following parameters:

- ADC resolution: 12 bit
- Sampling time: 532 us
- Calibration register: 4096

After initialization, the sensor reads the measurement values every 100ms and prints them to the STDOUT.

CHAPTER 70

About

This is a manual test application for the ISL29020 light sensor driver.

Usage

This test application will initialize the list sensor with the following parameters:

- Mode: Ambient light measurement
- Range: 16000LUX

After initialization, the sensor value is read every 250ms and printed to the STDOUT.

To verify the seen value you can hold the sensor into a bright light or shield the sensor to see the value changing.

CHAPTER 72

About

This is a manual test application for the ISL29125 light sensor driver.

CHAPTER 73

Usage

This test application will initialize the list sensor with the following parameters:

- Mode: All modes are tested once, then RGB mode is used continuously
- Range: 16000 lux
- Resolution: 16 bit

After initialization, the sensor value is read every 250ms and printed to the stdout.

Expose the sensor to varying light sources to see changing readouts.

CHAPTER 74

About

This is a manual test application for testing the KW2xrf network device driver.

For running this test, you need to connect/configure the following pins of your radio device:

- SPI DEV
- CS (chip select)
- INT (external interrupt)

CHAPTER 75

Usage

For testing the radio driver you can use the netif and txtsnd shell commands that are included in this application.

CHAPTER 76

About

This is a manual test application for the L3G4200D gyroscope driver.

Usage

This test application will initialize the pressure sensor with the following parameters:

- Sampling Rate: 100Hz
- Bandwidth: 25Hz
- Scale: 500DPS

After initialization, the sensor reads the angular speed values every 10ms and prints them to the STDOUT.

CHAPTER 78

About

This is a manual test application for the LIS3DH accelerometer driver.

Usage

This test application will initialize the accelerometer with the following parameters:

- Sampling Rate: 100Hz
- Scale: 4G
- Temperature sensor: Enabled

After initialization, the sensor reads the acceleration values every 100ms and prints them to the STDOUT.

CHAPTER 80

About

This is a manual test application for the LPS331AP pressure sensor driver.

Usage

This test application will initialize the pressure sensor with the following parameters:

- Sampling Rate: 7Hz

After initialization, the sensor reads the pressure and temperature values every 250ms and prints them to the STDOUT.

CHAPTER 82

About

This is a manual test application for the MAG3110 magnetometer driver.

CHAPTER 83

Usage

This test application will initialize the MAG3110 with the following parameters:

- output rate set to 1.25 Hz
- over sample ratio set to 128

After initialization, the sensor reads the x-, y-, z-axis values every 1 s and prints them to STDOUT.

CHAPTER 84

About

This is a manual test application for the MMA8652 accelerometer driver.

Usage

This test application will initialize the MMA8652 sensor with the following parameters:

- full scale parameter set to +/-2 g
- 6.25 Hz output data-rate

After initialization, the sensor reads the x-, y-, z-axis values every 1 s and prints them to STDOUT.

CHAPTER 86

About

This is a manual test application for the MPL3115A2 driver.

Usage

This test application will initialize the MPL3115A2 sensor with the following parameters:

- oversample ratio 128

After initialization, the sensor reads the pressure and temperature values every 1s and prints them to STDOUT.

CHAPTER 88

About

This is a test application for the MPU-9150 Nine-Axis Driver.

Usage

The application will initialize the MPU-9150 motion sensor with the following parameters:

- Accelerometer: ON
- Gyroscope: ON
- Magnetometer: ON
- Sampling Rate: 200Hz
- Compass Sampling Rate: 100Hz

After initialization, the application reads accel, gyro, compass and temperature values every second and prints them to the STDOUT.

CHAPTER 90

Test for nrf24l01p lowlevel functions

90.1 About

This is a small test application to see how the lowlevel-driver functions of the proprietary nrf24l01p-transceiver work. These functions consist of general SPI and GPIO commands, which abstract the driver-functions from the used board.

90.2 Predefined pin mapping

Please compare the `tests/driver_nrf24l01p_lowlevel/Makefile` for predefined pin-mappings on different boards. (In addition, you also need to connect to 3V and GND)

90.3 Usage

You should be presented with the RIOT shell, providing you with commands to initialize the transceiver (command: `it`), sending one packet (command: `send`) or read out and print all registers of the transceiver as binary values (command: `prgs`).

90.3.1 Procedure

- take two boards and connect a transceiver to each (it should be also possible to use one board with different SPI-ports)
- depending on your board, you'll maybe also need to connect a UART/tty converter
- build and flash the test-program to each
- open a terminal (e.g. pyterm) for each
- if possible, reset the board by using the reset-button. You'll see "*Welcome to RIOT*" etc.
- type `help` to see the description of the commands
- initialize both with `it`
- with one board, send a packet by typing `send`
- in the next step you can also use `send` to send data in the other direction
- now you can use send on both boards/transceivers to send messages between them

90.4 Expected Results

After you did all steps described above, you should see that a 32 Byte sequence (numbers from 32...1) has been transferred from one device to the other. This sequence is printed out from the receiver after the receive interrupt occurred and the receive-procedure has been made.

After initialization (`it`) you should see the following output:

```
 > it

Init Transceiver

Registering nrf24l01p_rx_handler thread...
################## Print Registers ###################
REG_CONFIG:
0x0 returned: 00111111

REG_EN_AA:
0x1 returned: 00000001

REG_EN_RXADDR:
0x2 returned: 00000011

REG_SETUP_AW:
0x3 returned: 00000011

REG_SETUP_RETR:
0x4 returned: 00101111

REG_RF_CH:
0x5 returned: 00000101

REG_RF_SETUP:
0x6 returned: 00100111

REG_STATUS:
0x7 returned: 00001110

REG_OBSERVE_TX:
0x8 returned: 00000000

REG_RPD:
0x9 returned: 00000000

REG_RX_ADDR_P0:
0xa returned: e7 e7 e7 e7 e7

REG_TX_ADDR:
0x10 returned: e7 e7 e7 e7 e7

REG_RX_PW_P0:
0x11 returned: 00100000

REG_FIFO_STATUS:
```

```
0x17 returned: 00010001

REG_DYNPD:
0x1c returned: 00000000

REG_FEATURE:
0x1d returned: 00000000
```

After the data has been sent (`send`), you should see the following output on the receiver terminal:

```
In HW cb
nrf24l01p_rx_handler got a message: Received packet.
32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1
```

Expected result

The test will initialize all basic networking functionality including the minimal NRF51822 radio driver and run the shell providing netif shell commands.

CHAPTER 92

Background

Use the shell commands to test the link layer functionality of the minimal NRF51822 radio driver (nrfmin).

CHAPTER 93

About

This is a manual test application for the SPI NVRAM driver.

Usage

This test application will initialize the SPI bus and NVRAM device with the following parameters:

- Baudrate: 10 MHz (overridable by setting TEST_NVRAM_SPI_SPEED)
- SPI config: SPI_CONF_FIRST_RISING (overridable by setting TEST_NVRAM_SPI_CONF)

The memory will be overwritten by the test application. The original contents will not be restored after the test.

CHAPTER 95

About

This application is a test for the PDC8544 LCD display driver.

CHAPTER 96

Usage

Use the provided shell commands to control your display.

CHAPTER 97

About

This is a manual test application for the PIR motion sensor driver.

In order to build this application, you need to add the board to the Makefile's `WHITELIST` first and define a pin mapping (see below).

CHAPTER 98

Usage

There are two ways to test this. You can either actively poll the sensor state, or you can register a thread which receives messages for state changes.

98.1 Interrupt driven

Connect the sensor's "out" pin to a GPIO of your board that can be configured to create interrupts. Compile and flash this test application like:

```
export BOARD=your_board
export PIR_GPIO=name_of_your_pin
make clean
make all-interrupt
make flash
```

The output should look like:

```
kernel_init(): jumping into first task...

PIR motion sensor test application

Initializing PIR sensor at GPIO_8... [OK]

Registering PIR handler thread...     [OK]
PIR handler got a message: the movement has ceased.
PIR handler got a message: something started moving.
PIR handler got a message: the movement has ceased.
```

98.2 Polling Mode

Connect the sensor's "out" pin to any GPIO pin of you board. Compile and flash this test application like:

```
export BOARD=your_board
export PIR_GPIO=name_of_your_pin
make clean
make all-polling
make flash
```

The output should look like this:

```
kernel_init(): jumping into first task...
PIR motion sensor test application

Initializing PIR sensor at GPIO_10... [OK]

Printing sensor state every second.
Status: lo
...
Status: lo
Status: hi
...
```

CHAPTER 99

Background

Test for the high level servo driver.

CHAPTER 100

Expected result

A servo connected to `PWM_0` channel 0 should move back and forth inside the angle -90 degrees to +90 degrees, approximately.

Using a scope should show a varying pulse length between 1000 us to 2000 us long. The requested frequency is 100 Hz, but due to hardware limitations it might not be possible to achieve the selected frequency. The pulse width should, however, remain the same, only the frequency of pulses (and hence the duty cycle) should differ.

CHAPTER 101

About

This is a manual test application for the SRF02 ultrasonic ranger driver.

CHAPTER 102

Usage

After initialization, the sensor value is read periodically and printed to the STDOUT.

To verify the seen value you can focus the sensor against any reflecting object and vary the distance to see the value changing.

CHAPTER 103

About

This is a manual test application for the SRF08 ultrasonic ranger driver.

CHAPTER 104

Usage

After initialization, the sensor values are read periodically and printed to the STDOUT. Here, three echos are stored.

To verify the seen values you can focus the sensor against any reflecting object and vary the distance to see especially the first measured value changing.

CHAPTER 105

About

This is a manual test application for the TCS37727 driver.

Usage

This test application will initialize the TCS37717 sensor with the following parameters: Gain 4x, RGBC on, Proximity Detection off

After initialization, the sensor reads the RGBC ADC data every 1s and prints them to STDOUT.

CHAPTER 107

About

This is a manual test application for the TMP006 driver.

CHAPTER 108

Usage

This test application will initialize the TMP006 sensor with the following parameters:

- conversion rate 1 per second

After initialization, the sensor reads the temperature values every 1s and prints them to STDOUT.

About

This is a manual test application for testing the Xbee S1 network device driver.

For running this test, you need to connect the following pins of a Xbee S1 module to your board:

- UART RX
- UART TX
- VCC (3V3)
- GND

NOTE: when you use an Arduino Xbee shield, the Xbee module is connected to the same UART as RIOTs standard out. In this case you must redefine the STDIO to another UART interface in the board.h and connect a UART-to-USB adapter to that UART.

CHAPTER 110

Usage

For testing the Xbee driver you can use the netif shell commands that are included in this application.

Expected result

This application should infinitely print ‘-‘ characters. If it prints only a single ‘+’ characters the test must be considered as failed.

CHAPTER 112

Background

This test was introduced due to an error for floating point handling in an older newlib version.

The idea for this test is taken from: http://sourceware.org/ml/newlib/2010/msg00149.html

CHAPTER 113

Expected result

When running this test, you should see the samples of all configured ADC channels continuously streamed to std-out.

CHAPTER 114

Background

This test application will initialize each configured ADC device to sample with 10-bit accuracy. Once configured the application will continuously convert each available channel and print the conversion results to std-out.

For verification of the output connect the ADC pins to known voltage levels and compare the output.

CHAPTER 115

About

This is a test application for a digital to analog converter (DAC).

This test application will initialize each configured DAC and one ADC (ADC_O) device to sample with 10-bit accuracy. The ADC is only initialized if there is one available on your board.

After initialization, values from 0 to 1000 are converted through the DACs in a loop. Shortly after the digital to analog conversion of one number, the ADC_0 samples its input signal. The numbers that are given to the DACs and the numbers that are sampled by the ADC were printed to the STDOUT.

Usage

a) Connect an oscilloscope to the DAC pins and look at the ten iteration signal levels

or

b) Connect the ADC input to the DAC outputs successively and compare if the sampled input value correlates with the printed output value at each DAC port.

Expected result

This test enables you to test all available low-level I2C functions. Consult the ‘help’ shell command for available actions.

CHAPTER 118

Background

Test for the low-level I2C driver.

Expected result

If everything is running as supposed to, you should see a 1KHz PWM with oscillating duty cycle on each channel of the selected PWM device.

CHAPTER 120

Background

Test for the low-level PWM driver.

Expected result

This test outputs a sequence of random bytes, starting with one, then two and so on, until 20 random bytes are printed. Then the application sleeps for a second and starts over.

CHAPTER 122

Background

This test was introduced to test the implementation of the low-level random number generator driver. For most platforms the implementation is based on hardware CPU peripherals.

Expected result

When everything works as expected, you should see a alarm message after 10 seconds from start-up.

CHAPTER 124

Background

Test for the low-level RTC driver.

Expected result

When everything works as expected, you should see a hello message popping up every 10 seconds.

CHAPTER 126

Background

Test for the low-level RTT driver.

Expected result

You should be presented with the RIOT shell, providing you with commands to initialize a board as master or slave, and to send and receive data via SPI.

CHAPTER 128

Background

Test for the low-level SPI driver.

CHAPTER 129

Unittests

129.1 Building and running tests

Tests can be built by calling:

```
cd tests/unittests
make
```

If there are tests for a module you even can build tests specifically for this module:

```
make tests-<module>
# e.g.
make tests-core
```

You then can run the tests by calling

```
make term
```

or flash them to your board as you would flash any RIOT application to the board (see [[board documentation|RIOT-Platforms]]).

129.1.1 Other output formats

Other output formats using *embUnit*'s `textui` library are available by setting the environment variable `OUTPUT`:

- Compiler: `OUTPUT="COMPILER"`
- Text: `OUTPUT="TEXT"`
- XML: `OUTPUT="XML"`
- Color: `OUTPUT="COLOR"` (like default, but with red/green output)
- Colored-Text: `OUTPUT="COLORTEXT"` (like `TEXT`, but with red/green output)

Compile example

```
OUTPUT=''COMPILER'' make tests-core
make term
```

(only outputs in case of test failures)

Text example

```
OUTPUT=''TEXT'' make tests-core
make term
```

```
- core_bitarithm_tests
1) OK test_SETBIT_null_null
2) OK test_SETBIT_null_limit
3) ...
- core_clist_tests
25) ...
- ...

OK (... tests)
```

XML example

```
OUTPUT=''XML'' make tests-core
make term
```

```
<?xml version=''1.0'' encoding='shift_jis' standalone='yes' ?>
<TestRun>
<core_bitarithm_tests>
<Test id=''1''>
<Name>test_SETBIT_null_null</Name>
</Test>
<Test id=''2''>
<Name>test_SETBIT_null_limit</Name>
</Test>
...
</core_bitarithm_tests>
<core_clist_tests>
<Test id=''25''>
<Name>test_clist_add_one</Name>
</Test>
...
</core_clist_tests>
<Statistics>
<Tests>...</Tests>
</Statistics>
</TestRun>
```

129.2 Writing unit tests

129.2.1 File struture

RIOT uses *embUnit* for unit testing. All unit tests are organized in `tests/unittests` and can be build module-wise, if needed. For each module there exists a `tests-<modulename>/tests-<modulename>.h` file, at least one C file in `tests-<modulename>/` and a `tests-<modulename>/Makefile`. It is recommended to add a C file named `tests-<modulename>/tests-<modulename>-<headername>.c` for every header file that defines functions (or macros) implemented in the module. If there is only one such header file `tests-<modulename>/tests-<modulename>.c` should suffice.

Each `*.c` file should implement a function defined in `tests-<modulename>/tests-<modulename>.h`, named like

```
Test *tests_<modulename>_<headername>_tests(void);

/* or respectively */

Test *tests_<modulename>_tests(void);
```

129.2.2 Testing a module

To write new tests for a module you need to do three things:

1. ***Create a Makefile***: add a file `tests-<modulename>/Makefile`
2. ***Define a test header***: add a file `tests-<modulename>/tests-<modulename>.h`
3. ***Implement tests***: for each header file, that defines a function or macro implemented or related to the module, add a file `tests-<modulename>/tests-<modulename>-<headername>.c` or `tests-<modulename>/tests-<modulename>.c` if there is only one header.

Create a Makefile

The Makefile should have the following content:

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

Define a test header.

The test header `tests-<modulename>/tests-<module>.h` of a module you add to `tests/unittests/` should have the following structure

```
/*
 * Copyright (C) <year> <author>
 *
 * This file is subject to the terms and conditions of the GNU Lesser
 * General Public License v2.1. See the file LICENSE in the top level
 * directory for more details.
 */

/**
 * @addtogroup  unittests
 * @{
 *
 * @file
 * @brief       Unittests for the ``module`` module
 *
 * @author      <author>
 */
#ifndef TESTS_<MODULE>_H_
#define TESTS_<MODULE>_H_
#include ``embUnit/embUnit.h''

#ifdef __cplusplus
extern ``C'' {
```

```
#endif

/**
 * @brief   Generates tests for <header1>.h
 *
 * @return  embUnit tests if successful, NULL if not.
 */
Test *tests_<module>_<header1>_tests(void);

/**
 * @brief   Generates tests for <header2>.h
 *
 * @return  embUnit tests if successful, NULL if not.
 */
Test *tests_<module>_<header2>_tests(void);

/* ... */

#ifdef __cplusplus
}
#endif

#endif /* TESTS_<MODULE>_H_ */
/** @} */
```

Implement tests

Every `tests-<modulename>/tests-<module>*.c` file you add to `tests/unittests/` should have the following structure:

```
/*
 * Copyright (C) <year> <author>
 *
 * This file is subject to the terms and conditions of the GNU Lesser
 * General Public License v2.1. See the file LICENSE in the top level
 * directory for more details.
 */

/* clib includes */

#include ``embUnit/embUnit.h''

#include ``<header>.h''

#include ``tests-<module>.h''

/* your macros */

/* your global variables */

static void set_up(void)
{
    /* omit if not needed */
}
```

```
static void tear_down(void)
{
    /* omit if not needed */
}

static void test_<function1>_<what1>(void) {
    /* ... */

    TEST_ASSERT(/* ... */);
}

static void test_<function1>_<what2>(void) {
    /* ... */

    TEST_ASSERT(/* ... */);
}

/* ... */

static void test_<function2>_<what1>(void) {
    /* ... */

    TEST_ASSERT(/* ... */);
}

static void test_<function2>_<what2>(void) {
    /* ... */

    TEST_ASSERT(/* ... */);
}

/* ... */

Test *tests_<module>_<header>_tests(void)
{
    EMB_UNIT_TESTFIXTURES(fixtures) {
        new_TestFixture(test_<function1>_<what1>),
        new_TestFixture(test_<function1>_<what2>),
        new_TestFixture(test_<function2>_<what1>),
        new_TestFixture(test_<function2>_<what2>),
        /* ... */
    };

    EMB_UNIT_TESTCALLER(<module>_<header>_tests, ``<module>_<header>_tests'',
                        tests_<module>_<header>_set_up,
                        tests_<module>_<header>_tear_down, fixtures);
    /* set up and tear down function can be NULL if omitted */

    return (Test *)&<module>_<header>;
}
```

The following assertion macros are available via *embUnit*

CHAPTER 130

Test warning on conflicting features

Using conflicting features provided by boards was invisible for the user until the used features resulted in a traceable problem or the user was aware of the conflict in advance from documentation ect. Now, existing and known conflicts can be recorded into `FEATURES_CONFLICT` for each board to inform the user on a conflict situation during compile time.

This test requires conflicting features in its `Makefile`, i.e. `FEATURES_REQUIRED = periph_dac periph_spi`. It is expected to be presented with a warning on the conflicts with a short description message during compile time for the stm32f4discovery by now, i.e. :

```
$ make BOARD=stm32f4discovery
The following features may conflict: periph_dac periph_spi
Rationale: On stm32f4discovery boards there are the same pins for the DAC and/or SPI_0.

EXPECT undesired behaviour!
```

The warning presents the conflicting features derived from `FEATURES_CONFLICT` and an optional message derived from `FEATURES_CONFLICT_MSG` provided int the `./RIOT/board/stm32f4discovery/Makefile.features`.

Whenever an application, such as this test, requires board features that match a *conflict group*, e.g. `FEATURES_REQUIRED = periph_dac periph_spi`, a similar warning to the above will be displayed during compile time.

###Usage of *conflict groups*:

- Conflicting features are described in groups separated by a `:` (doublecolon) for each feature, e.g.: `FEATURES_CONFLICT = periph_spi:periph_dac`, which states that `periph_spi` conflicts with `periph_dac`. As seen above, this is the conflict of `SPI_0` pinout is shared with `DAC` on the stm32f4discovery board.
- Distinct groups of conflicts are whitespace separated, e.g.: `featureA:featureB featureC:featureD`, which states that `featureA` conflicts with `featureB`, and `featureC` conflicts with `featureD`. This also means, that e.g. `FEATURES_REQUIRED = featureA featureD` would **not** produce a warning.
- The groups can have an arbitrary number of conflicting features, e.g.: `featureA:featureB:featureC:featureX:featureY:featureZ`
- An optional information can be given using the `FEATURES_CONFLICT_MSG`, e.g.: `FEATURES_CONFLICT_MSG = "featureA uses the same pins as featureB"`
- If the required features match multiple conflict groups, **ALL** conflicting features are provided to the user, e.g.: `FEATURES_CONFLICT = featureA:featureB featureC:featureD` and

`FEATURES_REQUIRED = featureA featureB featureC featureD` would result in: `The following features may conflict: featureA featureB featureC featureD`

RST files: