H2020 i**P** **O**ver **I**c**N**- the bet**T**er IP (POINT)

HowTo-IoT

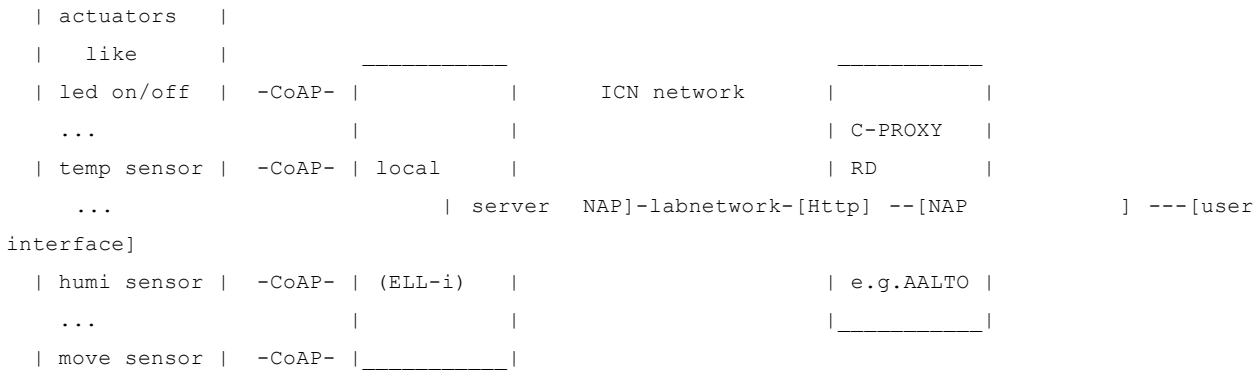
Installation and Configuration of the IoT Devices Operating the RIOT-OS

List of Authors:

Eero Hakala

1. Overview

ELL-i arm based embedded devices



The application is based on FIWARE framework (https://www.fiware.org) with the following pieces:
CoAP server == iot agent == context broker (orion) == freeboard

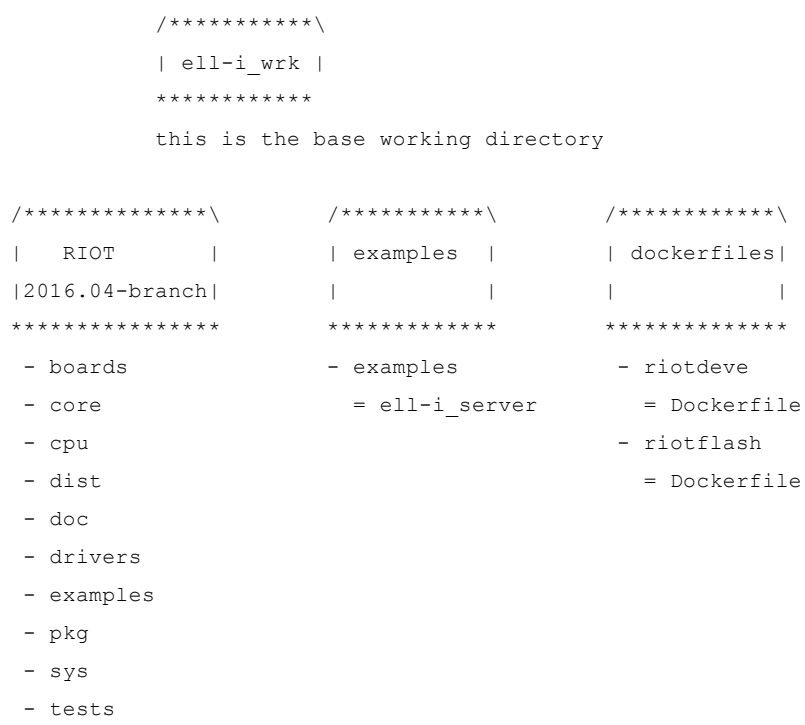
2. Getting started with CoAP

Experimenting with CoAP using a single (linux) host and libcoap. Description of installing and using libcoap (coap-client and coap-server)

http://wiki.point-h2020.eu/pointwiki/index.php?title=Using_libCoAP

2.1 Development environment for ELL-i leaf nodes

RIOT version is fixed to 2016.10-branch and a new board is added for nucleo-f401 that has A/D defined: adc_f401



The idea to arrange the folders like this is to keep the application (in the ell-i_wrk/examples directory) separate from operating system (resides in ell-i_wrk/RIOT)!

The process is quite straightforward:
- create a working directory and get the needed files
- create development environment
- Compile, flash and test

2.1.1 Getting the files

Copy the following files to a suitable directory (ell-i_wrk)

```
~/Blackadder/deployment/IoT/Ell-i_dockerfiles.tar
~/Blackadder/deployment/IoT/Ell-i_examples.tar
~/Blackadder/deployment/IoT/Ell-i_adc_f401.tar
```

and then create the environment

```
$ git clone https://github.com/RIOT-OS/RIOT
$ cd RIOT/
$ git checkout 2016.04-branch
$ cd ..
$ tar xvf ell-i_adc_f401.tar
$ tar xvf ell-i_dockerfiles.tar
$ tar xvf ell-i_examples.tar
```

Connect the embedded device (using both USB and ethernet!), note that you can get power either from USB-connection [U5V] or from ethernet (if the network is POE enabled [E5V]) ==> JP5 (PWR)!
Setup the tty-connection using /dev/ttyACM0 (in ubuntu!!) with e.g. minicom

2.1.2 Create the docker images for development and flashing

```
$ docker build --rm -t rdeve dockerfiles/riotdeve
$ docker build --rm -t rflash dockerfiles/riotflash
```

2.1.3 Compile, flash and test

Compile (all in same row!)

```
$ docker run -it --rm --privileged -v $(pwd):/data/riotbuild rdeve
make -C examples/ellin_server BOARD=adc_f401 QUIET=1
```

Flash (all in same row!)

```
$ docker run -it --rm --privileged -v $(pwd):/data/riotbuild rflash
make -C examples/ellin_server BOARD=adc_f401 QUIET=1 flash
```

monitor & control with USB-serial:

```
------------------------------------------------------------------
--
 main(): This is RIOT! (Version:
2016.10-devel-1763-geca49b-5f268b48a498-2016)
 ELL-i nanocoap example application
 Configured network interfaces:
 Iface  5   HWaddr: d8:80:39:02:c0:e7

           MTU:1500  HL:64  RTR  RTR_ADV
           Source address length: 6
           Link type: wired
           inet6 addr: ff02::1/128  scope: local [multicast]
           inet6 addr: fe80::da80:39ff:fe02:c0e7/64  scope: local
           inet6 addr: ff02::1:ff02:c0e7/128  scope: local
[multicast]
           inet6 addr: ff02::2/128  scope: local [multicast]

 Waiting for incoming UDP packet...

------------------------------------------------------------------
--
```

the test setup
LED (in series with 1k) ON/OFF controlled (digital out, pin PA10) reading status with digital input LOW/HIGH, reading luminosity with LDR (in series with 4.7k) value is read from pin PA0 (analog in), those both can confirm the LED state. Connected pins: PA10, PA0, GND and 3.3V)

test the application with CoAP client (monitor: USB [IPv6 address listed], CoAP: ethernet)

```
echo 01 |coap-client -m put
coap://[fe80::da80:39ff:fe02:c0e7%eth0]/output-digital/PA10 -f -
 coap-client -m get
coap://[fe80::da80:39ff:fe02:c0e7%eth0]/input-analog/PA0
```

```
 coap-client -m get
coap://[fe80::da80:39ff:fe02:c0e7%eth0]/input-digital/PA10
 echo 1 |coap-client -m put
coap://[fe80::da80:39ff:fe02:c0e7%eth0]/output-digital/PA10 -f -
 coap-client -m get
coap://[fe80::da80:39ff:fe02:c0e7%eth0]/input-analog/PA0
 coap-client -m get
coap://[fe80::da80:39ff:fe02:c0e7%eth0]/input-digital/PA10
```

http://wiki.point-h2020.eu/pointwiki/images/thumb/9/9c/Ell-i_server_test-setup_for-digital_in-out_and_analog-in.png/120px-Ell-i_server_test-setup_for-digital_in-out_and_analog-in.png
http://wiki.point-h2020.eu/pointwiki/images/thumb/1/16/Ell-i_server_test-setup_for-digital_in-out_and_analog-in_breadboarding.png/90px-Ell-i_server_test-setup_for-digital_in-out_and_analog-in_breadboarding.png

3. Software structure of ELL-i node

ELL-i CoAP server is based on (RIOT-OS) nanocoap and it is implemented using RIOT-OS 2016.10-branch.

ELL-i leaf node is defined in examples/ellin_server directory:

```
 main.c         initiates the application
  coap_handler.call functionalities defined here (the handlers and
                 corresponding path definitions)
  Makefile
```

RIOT-OS modules, best list in Makefile

The structure of a typical (nano)CoAP-server application is straightforward:

```
main.c               takes care of initialization
```

```
coap_handler.c  contains the server logic
```

Adding functionalities is easy: you need only change coap_handler.c

This database defines supported features

```
/* must be sorted by path (alphabetically) */
const coap_resource_t coap_resources[] = {
    COAP_WELL_KNOWN_CORE_DEFAULT_HANDLER,
    { "/testi2/PA10", COAP_GET, _riot_testi2_PA10_handler },
    { "/testi3/PA10", COAP_GET, _riot_testi3_PA10_handler },
};
```

These again are the corresponding function calls, what should be done is added here!

```
static  ssize_t  _riot_testi2_PA10_handler(coap_pkt_t  *pkt,  uint8_t
*buf, size_t len)
{
    const char *teksti = "testi2";
    return coap_reply_simple(pkt, buf, len,
            COAP_FORMAT_TEXT, (uint8_t*)teksti, strlen(teksti));
}
static  ssize_t  _riot_testi3_PA10_handler(coap_pkt_t  *pkt,  uint8_t
*buf, size_t len)
{
    const char *teksti = "testi3";
    return coap_reply_simple(pkt, buf, len,
            COAP_FORMAT_TEXT, (uint8_t*)teksti, strlen(teksti));
}
```

Supported features are GET, PUT and POST. CoAP observe is an future RIOT feature