Internet-of-Things – Lab Exercise 04 RIOT-OS COAP

Group members: Tim van Dyck

Prerequisites: install Ubuntu 20.04. or higher – a VM/VBox works fine for this exercise. You can use VMware Player or Virtual Box. If you are working in the network lab in the FH Campus Wien, the environment will be provided for you.

Hardware: 2x SAMR21 board, 3x male/female jumper cable, 1x 4.7kohm resistor, 1x DS18S20 temperature sensor, 1x breadboard

You find your assignment here:
https://its.fh-campuswien.ac.at/hackmd/S9yE-TvERemQFduogx3KTw

You have to complete step one below to get all points on the lab!

I Configure both SAMR21, the COAP-Server and the Client, and get the temperature value to the client by asking the server.

Setup your hardware and read out the temperature from the DS18S20 temperature sensor. Add the necessary function to the COAP-Server.

Introduction

What is CoAP?

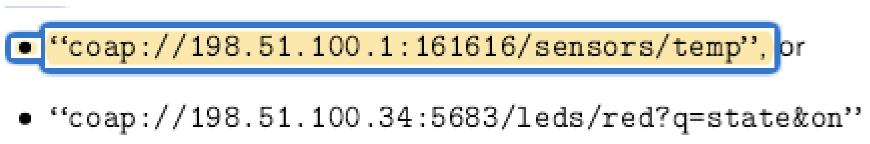
Protocol: 1-1 Communication Protocol, inspired by HTTP(GET,POST,PUT,DELETE), based on UDP.

Observe: Find if a new state is on a variable. No need for polling! **Get + Observe;** everytime the other device sees a change on a variable, it will push a notification to the original device.
Discovery: Discover devices around us. **Server List:** Stores a list of devices around them + the resources and media types

QOS: two modes -> **Confirmable message(acknowledgement)(get conf).** Requests can be carried in confirmable and non-confirmable messages, the same holds true for responses, but they can also be piggybacked in ACK messages.
Non Confirmable -> fire and forget!

Based on REST Architecture, but the things can act as both a client and server.
Responses contain a status code, like in http.

COAP URLS:



- "coap://198.51.100.1:161616/sensors/temp", or
- "coap://198.51.100.34:5683/leds/red?q=state&on"

Reference: IOT Application Protocol Lecture Notes

Actual setup

CoAP Server: Reads the temperature values. SAMR21-xpro board + DS18S20 sensor. Runs nanocoap.

CoAp Client: SAMR21-xpro board, runs gcoap.

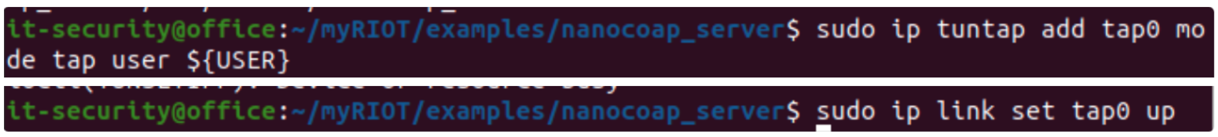
Lab part 1 - example coap server and client

Setting up the CoAP server

In the first step of the exercise, we should use the RIOT OS's nanocoap_server example. This example utilizes the nano_coap library, for hosting a coap server.

At first need to set up RIOT OS on our machine. As these steps were already covered in the second lab protocol, I won't go into detail here.

Then we connect our board, dedicated to host the CoAp Server, to the machine. Once it is connected, we must navigate to the dedicated directory of the example, which is located at ***"~/myRIOT/example/nanocoap_server".***



```
it-security@office:~/myRIOT/examples/nanocoap_server$ sudo ip tuntap add tap0 mo
de tap user ${USER}
it-security@office:~/myRIOT/examples/nanocoap_server$ sudo ip link set tap0 up
```

In this folder, we then issue following command:
sudo make BOARD=samr21-xpro
which compiles the example project.

Afterwards, we flash it to our board via this command:
sudo make BOARD=samr21-xpro flash

The application is now flashed to the board. We disconnect the board from our pc, in order to flash the CoAP client.

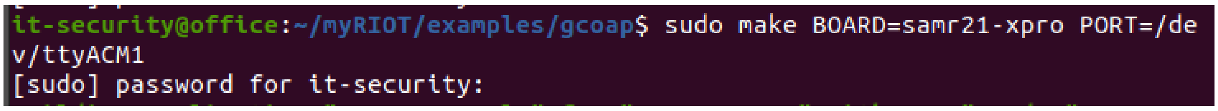
Setting of the CoAP client

The second board acts as the CoAP Client. This board will be used to issue requests to our CoAP server.

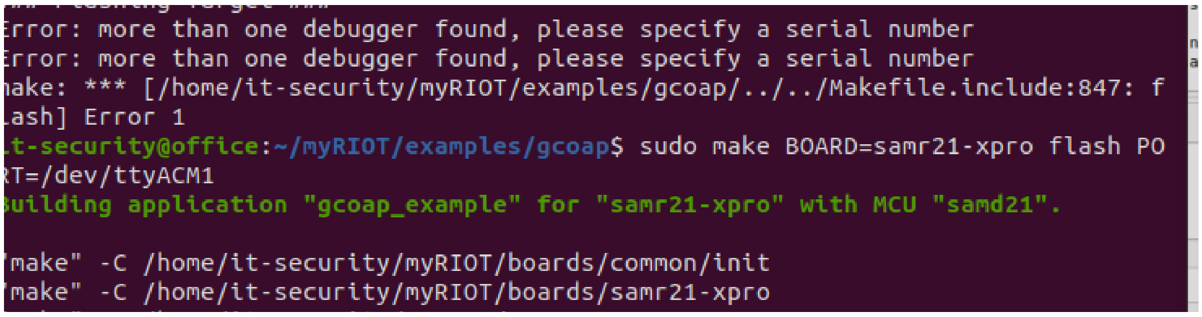
The RIOT OS example project, for the CoAP Client is located in the ***"~/myRIOT/examples/gcoap"*** directory.

First, we have to navigate to this said directory.

Afterwards we issue the following two commands, to compile and flash the project to our board.



```
it-security@office:~/myRIOT/examples/gcoap$ sudo make BOARD=samr21-xpro PORT=/de
v/ttyACM1
[sudo] password for it-security:
```



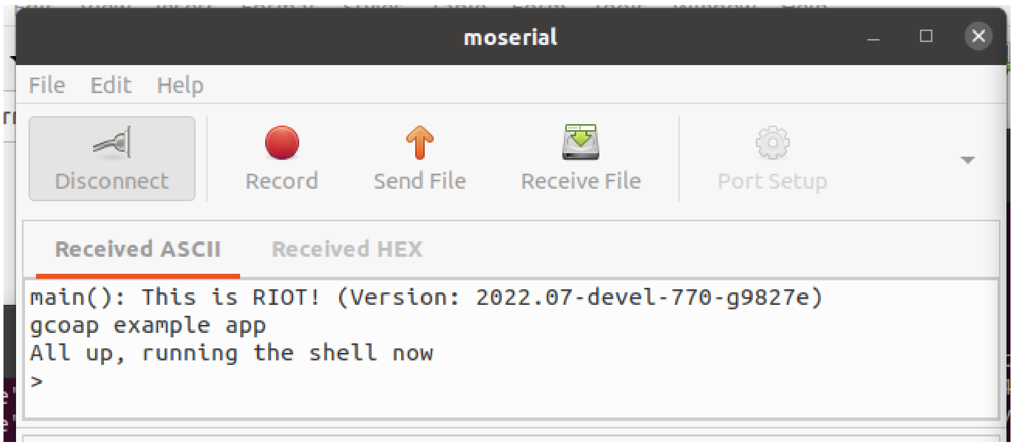
```
Error: more than one debugger found, please specify a serial number
Error: more than one debugger found, please specify a serial number
nake: *** [/home/it-security/myRIOT/examples/gcoap/../../Makefile.include:847: f
lash] Error 1
it-security@office:~/myRIOT/examples/gcoap$ sudo make BOARD=samr21-xpro flash PO
RT=/dev/ttyACM1
Building application "gcoap_example" for "samr21-xpro" with MCU "samd21".

"make" -C /home/it-security/myRIOT/boards/common/init
"make" -C /home/it-security/myRIOT/boards/samr21-xpro
```

Testing the Client and Server

Now, we are ready to test the CoAP client and server. First, we must reconnect the CoAP Server board to our machine.

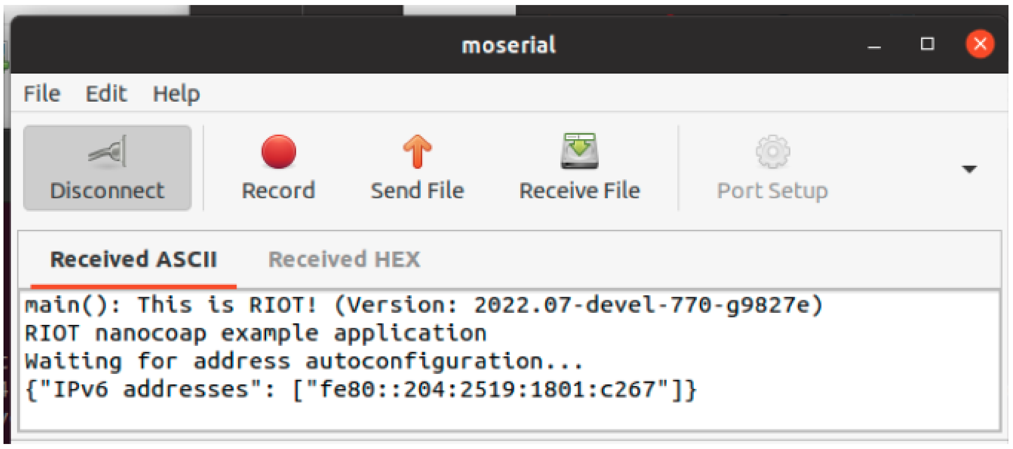
Afterwards, we open Moserial, so that we can read the messages sent by both boards.

Client:



```
main(): This is RIOT! (Version: 2022.07-devel-770-g9827e)
gcoap example app
All up, running the shell now
>
```

Server:

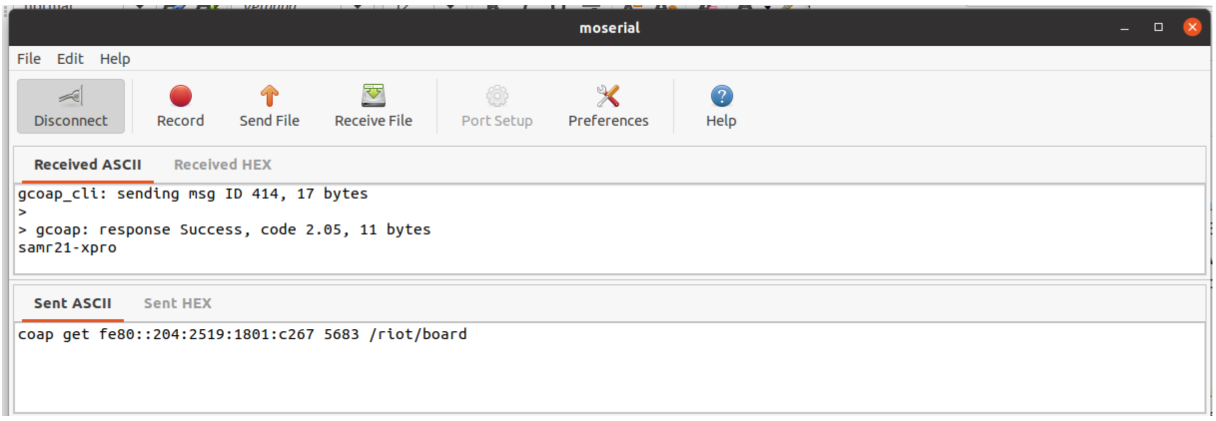


```
main(): This is RIOT! (Version: 2022.07-devel-770-g9827e)
RIOT nanocoap example application
Waiting for address autoconfiguration...
{"IPv6 addresses": ["fe80::204:2519:1801:c267"]}
```

As can be seen, the server opened it's connection at the ***fe80::204:2519:1801:c267*** address. The port is CoAP's default port 5683 (source:https://www.netify.ai/resources/protocols/coap).

GET requests can be sent to the CoAP Server from the client in following format via Moserial(the serial connection):
coap get {ipv6 address} {destination port} {resource path}

In order to test things out, we request the CoAP server ***/riot/board*** resource, via a GET request.



```
gcoap_cli: sending msg ID 414, 17 bytes
>
> gcoap: response Success, code 2.05, 11 bytes
samr21-xpro
```

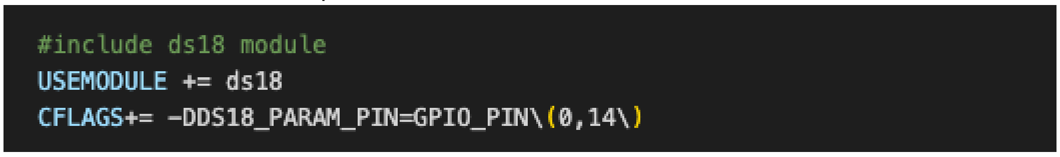
```
coap get fe80::204:2519:1801:c267 5683 /riot/board
```

And as we can see, it worked, and we got the information, on the CoAP client, that the CoAP server uses a samr21-xpro board.

Part 2: Configuring the CoAP Server to send back temperature values

Makefile

To get started, we have to include the ds18 module inside the Makefile, and configure the correct GPIO Pin to which our temperature sensor is connected.



```
#include ds18 module
USEMODULE += ds18
CFLAGS+= -DDS18_PARAM_PIN=GPIO_PIN\(0,14\)
```

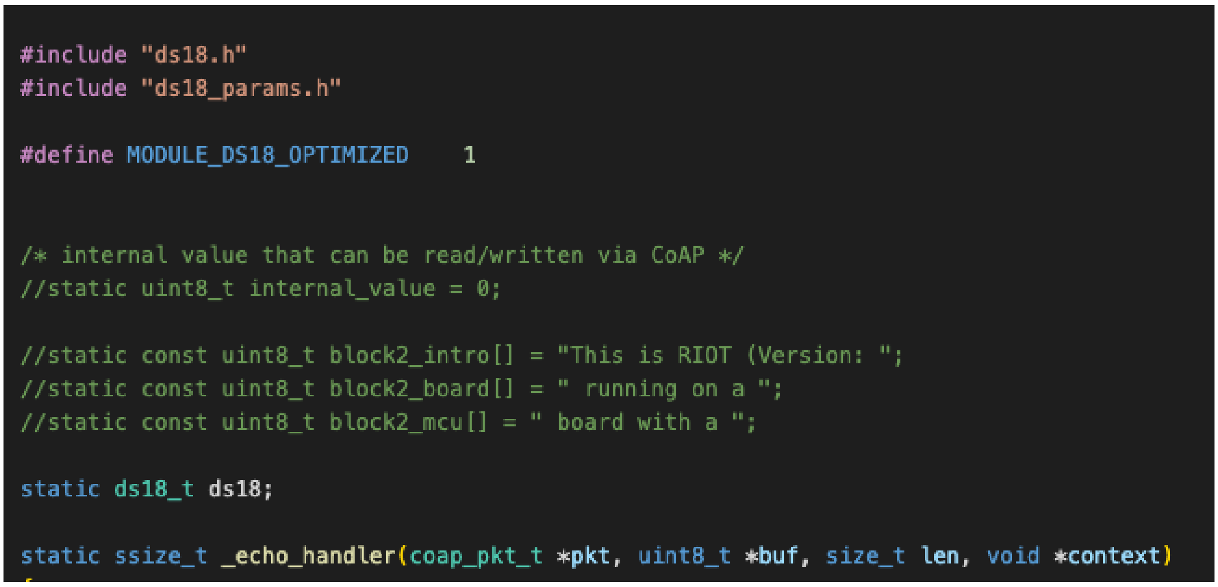
Defining the Resource

We want to create a Resource on our CoAP Server which provides us with the ability to read the temperature from the temperature sensor.

As described previously, CoAP supports multiple http-like request methods. In this case, we will use the GET method, as we want to get the temperature reading from our CoAP server.

Resources can be configured inside the *coap_handler.c* file.

Add imports:
To actually read the temperature from the ds18, we have to import some librarys:



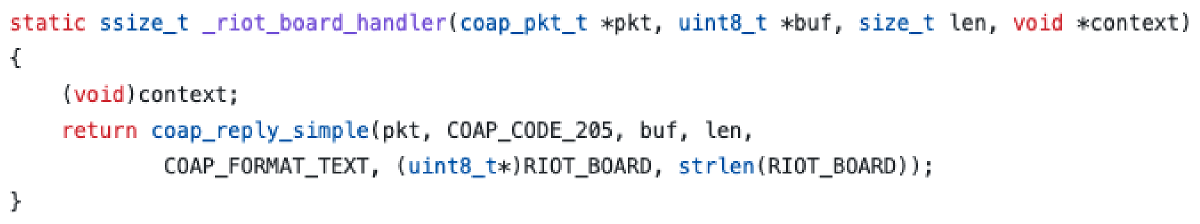
```
#include "ds18.h"
#include "ds18_params.h"

#define MODULE_DS18_OPTIMIZED    1


/* internal value that can be read/written via CoAP */
//static uint8_t internal_value = 0;

//static const uint8_t block2_intro[] = "This is RIOT (Version: ";
//static const uint8_t block2_board[] = " running on a ";
//static const uint8_t block2_mcu[] = " board with a ";

static ds18_t ds18;

static ssize_t _echo_handler(coap_pkt_t *pkt, uint8_t *buf, size_t len, void *context)
```

A ds18_t global variable is also defined. For details, please refer to the second lab protocol.

<u>Resource definition:</u>
Inside our coap_handler.c file, multiple handlers are present. Handlers are needed to provide resources. They are called when a resource is requested via a CoAP method.

Handlers are defined in the format:
typedef ssize_t(* coap_handler_t) (<u>coap_pkt_t</u> *pkt, uint8_t *buf, size_t len, void *context)
source(**https://doc.riot-os.org/group__net__nanocoap.html#ga687f23858a4a5036b60aedee24fb6b90)**
coap_pkt_t:Identifies the incoming request(https://doc.riot-os.org/structcoap__pkt__t.html).

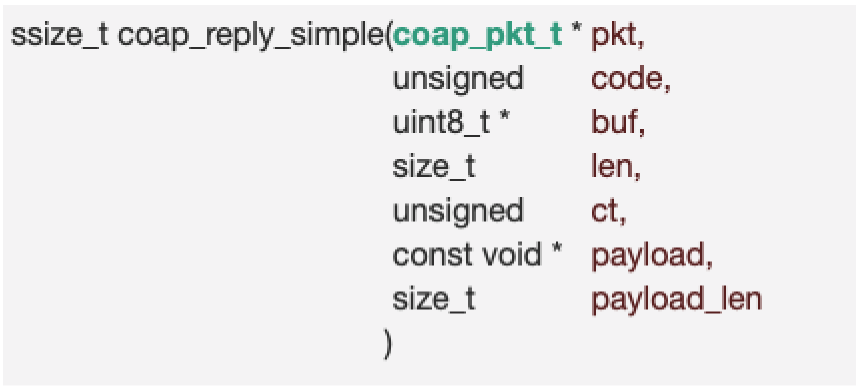
<u>Example handler:</u>
To gain a better understanding of how handlers work, let's look at an example handler:



```
static ssize_t _riot_board_handler(coap_pkt_t *pkt, uint8_t *buf, size_t len, void *context)
{
    (void)context;
    return coap_reply_simple(pkt, COAP_CODE_205, buf, len,
            COAP_FORMAT_TEXT, (uint8_t*)RIOT_BOARD, strlen(RIOT_BOARD));
}
```

The ***_riot_board_handler*** replys to GET requests with the riot_board used by the server.

Here we can observe the usage of **coap_reply_simple**.



```
ssize_t coap_reply_simple(coap_pkt_t * pkt,
                          unsigned     code,
                          uint8_t *    buf,
                          size_t       len,
                          unsigned     ct,
                          const void * payload,
                          size_t       payload_len
                         )
```

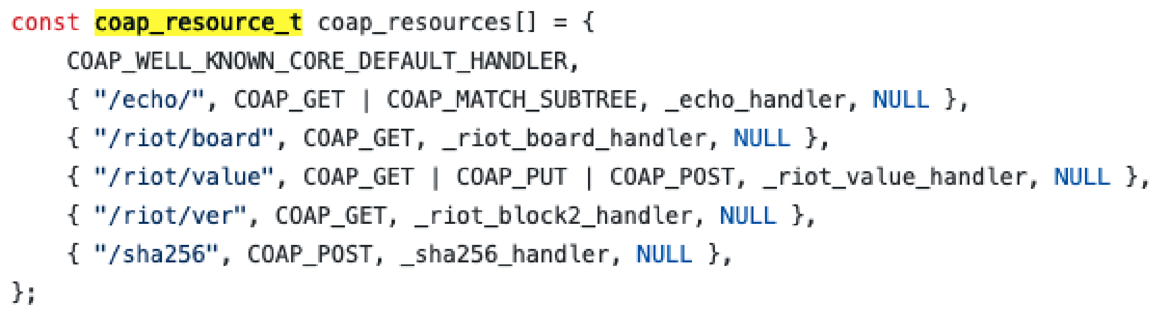
- pkt
  - the packet to reply to
- code
  - coap reply status code
- buf
  - the buffer to which the reply is written to
- len
  - the size of the buffer
- ct
  - content type of the payload, for example: COAP_FORMAT_TEXT
- payload
  - pointer to the payload!
- payload_len
  - length of the payload!

Source:https://doc.riot-os.org/group__net__nanocoap.html#ga82578bcdaeadcb10bdc3ecc52b0b6888

Providing the resource routes:
To actually use our handlers, we must define the routes and methods for our resources/handlers.

This is done inside the **coap_resource_t** structure.

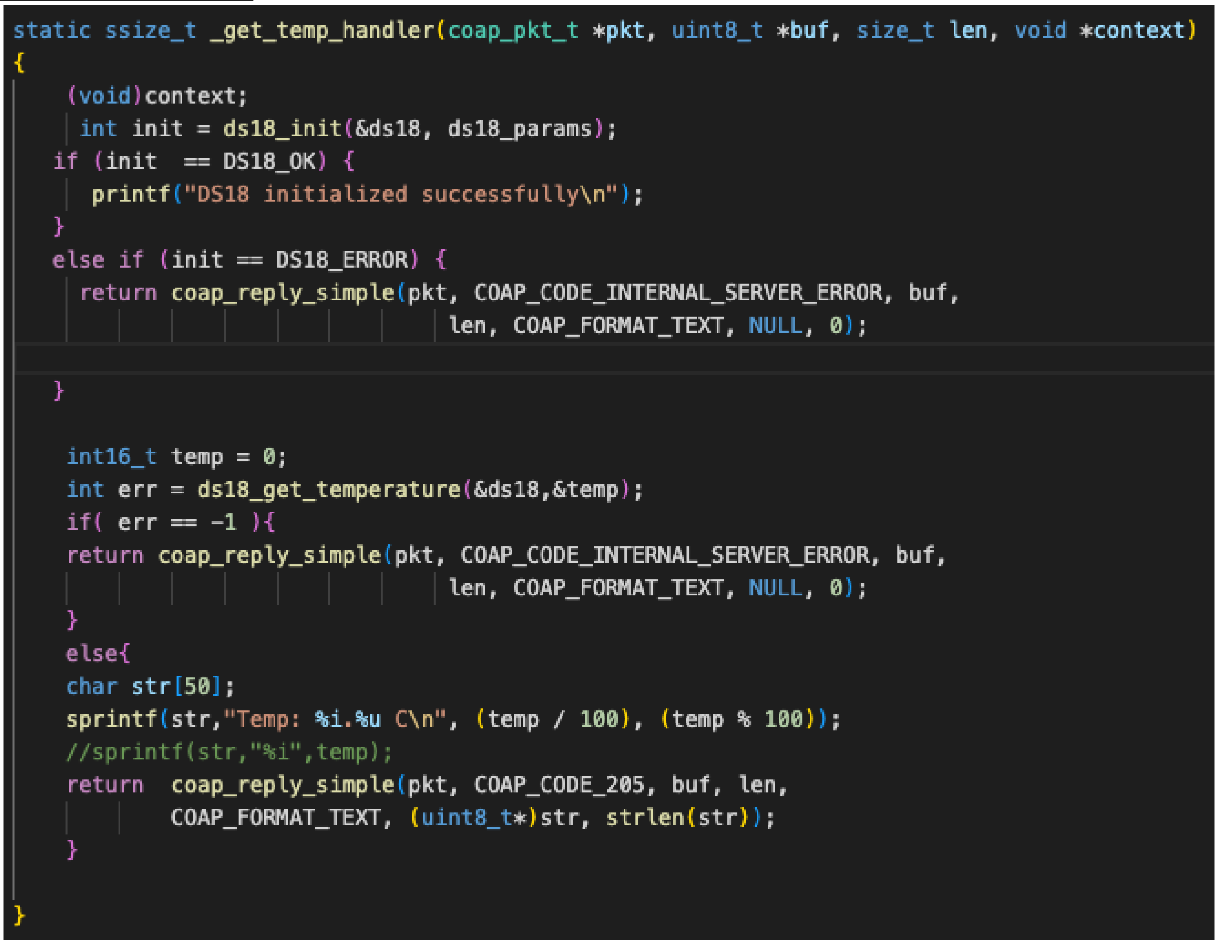
It’s fields contain a path, the methods, a handler and a pointer to user defined context data. (Source: https://doc.riot-os.org/structcoap__resource__t.html)



```
const coap_resource_t coap_resources[] = {
    COAP_WELL_KNOWN_CORE_DEFAULT_HANDLER,
    { "/echo/", COAP_GET | COAP_MATCH_SUBTREE, _echo_handler, NULL },
    { "/riot/board", COAP_GET, _riot_board_handler, NULL },
    { "/riot/value", COAP_GET | COAP_PUT | COAP_POST, _riot_value_handler, NULL },
    { "/riot/ver", COAP_GET, _riot_block2_handler, NULL },
    { "/sha256", COAP_POST, _sha256_handler, NULL },
};
```

Defining our own resource/handler/path

<u>Handler definition:</u>



```
static ssize_t _get_temp_handler(coap_pkt_t *pkt, uint8_t *buf, size_t len, void *context)
{
    (void)context;
     int init = ds18_init(&ds18, ds18_params);
   if (init  == DS18_OK) {
      printf("DS18 initialized successfully\n");
   }
   else if (init == DS18_ERROR) {
     return coap_reply_simple(pkt, COAP_CODE_INTERNAL_SERVER_ERROR, buf,
                                len, COAP_FORMAT_TEXT, NULL, 0);

   }

    int16_t temp = 0;
    int err = ds18_get_temperature(&ds18,&temp);
    if( err == -1 ){
    return coap_reply_simple(pkt, COAP_CODE_INTERNAL_SERVER_ERROR, buf,
                                len, COAP_FORMAT_TEXT, NULL, 0);
    }
    else{
    char str[50];
    sprintf(str,"Temp: %i.%u C\n", (temp / 100), (temp % 100));
    //sprintf(str,"%i",temp);
    return  coap_reply_simple(pkt, COAP_CODE_205, buf, len,
            COAP_FORMAT_TEXT, (uint8_t*)str, strlen(str));
    }

}
```

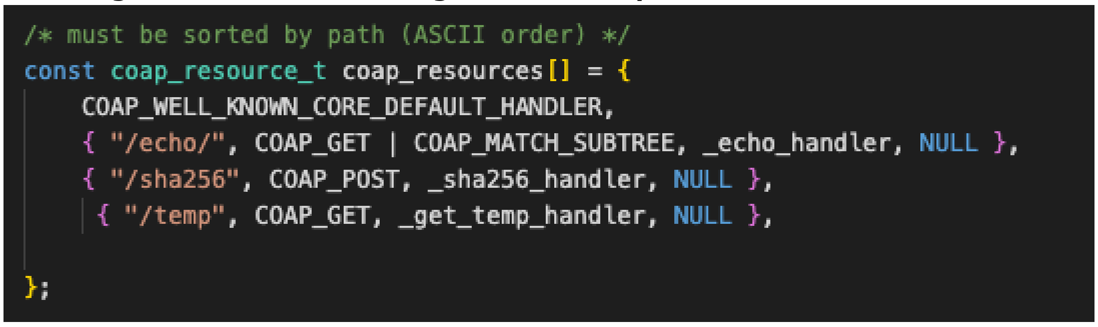
I defined my own handler, for reading temperatures, in the same format as the predefined handlers.

The body of the handler largely conforms to the code for reading temperature values in the second lab.

Some differences can be observed tough:

- Instead of reading every x seconds from the temperature sensor, we simply read from the sensor when the handler is called -> thus when a request for this resource comes in
- Errors are not printed, we instead return a reply with the Inter Server Error status code
- to create our response message, we initiate a character array of length 50, and use sprintf to create our formatted and converted temperature string
- If everything went well, we return a reply with Status code 205
    - This reply contains our formated temperature string **str**

<u>Including the handler and adding the resource path</u>



```
/* must be sorted by path (ASCII order) */
const coap_resource_t coap_resources[] = {
    COAP_WELL_KNOWN_CORE_DEFAULT_HANDLER,
    { "/echo/", COAP_GET | COAP_MATCH_SUBTREE, _echo_handler, NULL },
    { "/sha256", COAP_POST, _sha256_handler, NULL },
     { "/temp", COAP_GET, _get_temp_handler, NULL },

};
```

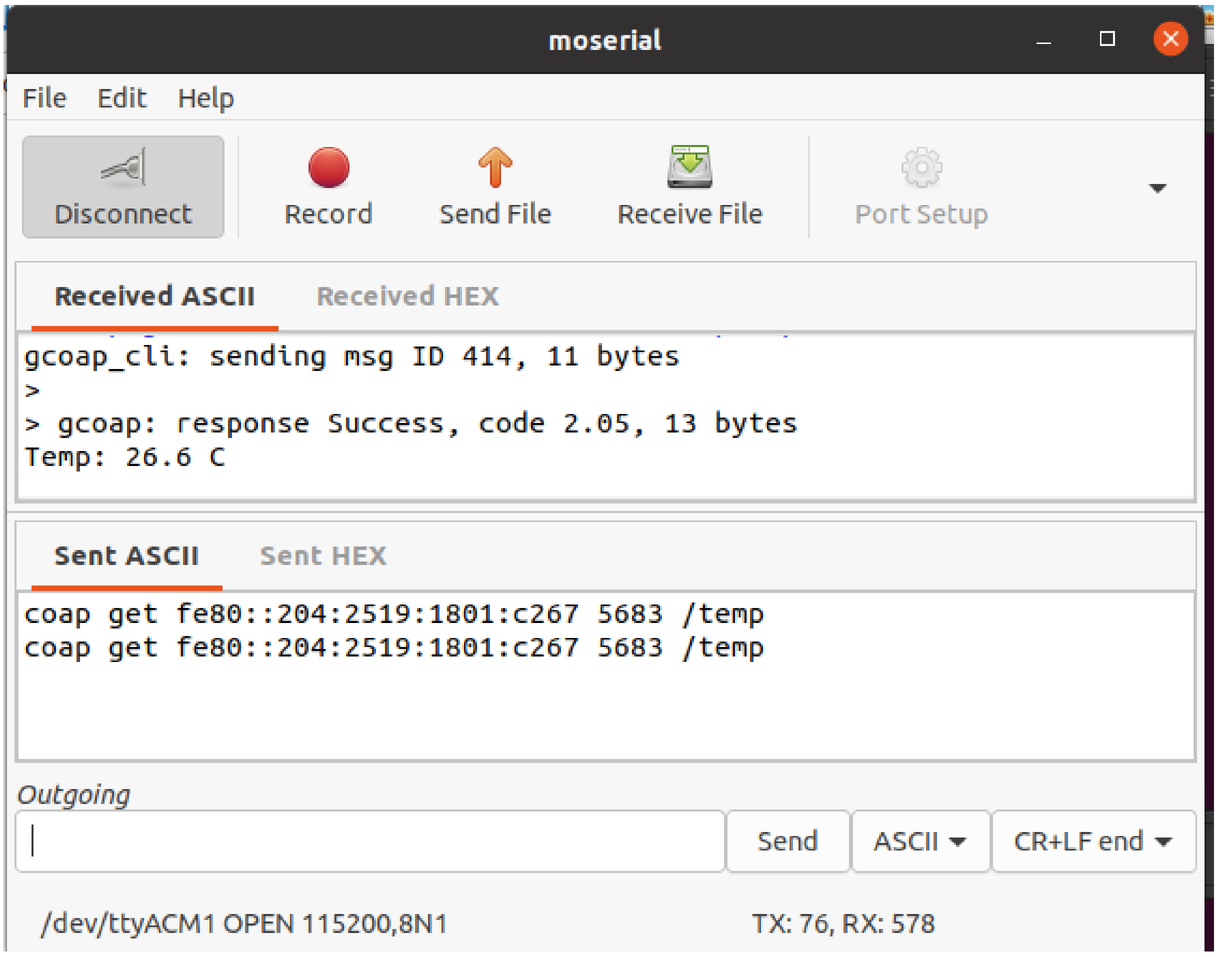
We added a path to our handler. The temperature can now be requested, via calling the **/temp** relative path, via a GET method.

Testing things out

I flashed the new project onto the CoAP server board, as described previously.

Afterwards, I connected to both the CoAP server and client via Moserial.

After the CoAP Server finished initializing, i issued following get request from the coap client:

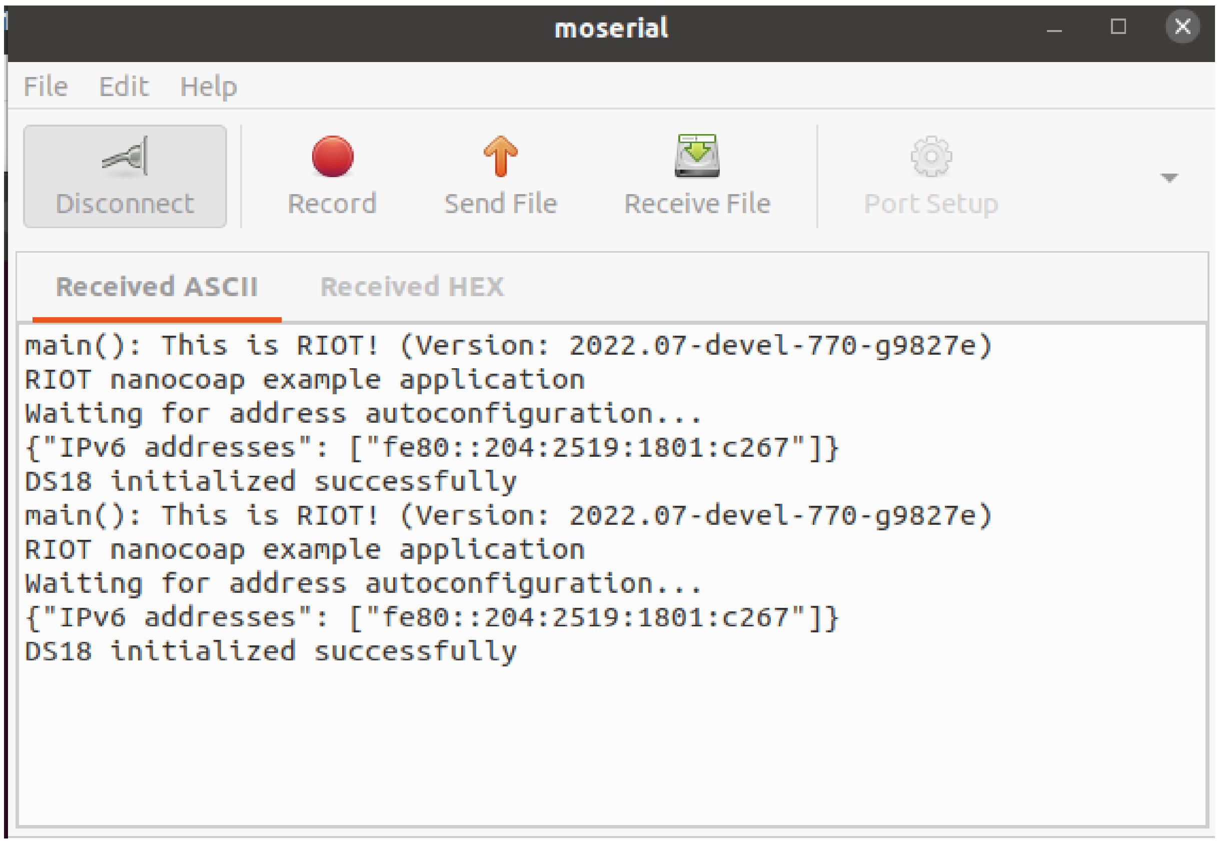


```
gcoap_cli: sending msg ID 414, 11 bytes
>
> gcoap: response Success, code 2.05, 13 bytes
Temp: 26.6 C
```

```
coap get fe80::204:2519:1801:c267 5683 /temp
coap get fe80::204:2519:1801:c267 5683 /temp
```

```
/dev/ttyACM1 OPEN 115200,8N1                TX: 76, RX: 578
```

Coap server output:



```
main(): This is RIOT! (Version: 2022.07-devel-770-g9827e)
RIOT nanocoap example application
Waiting for address autoconfiguration...
{"IPv6 addresses": ["fe80::204:2519:1801:c267"]}
DS18 initialized successfully
main(): This is RIOT! (Version: 2022.07-devel-770-g9827e)
RIOT nanocoap example application
Waiting for address autoconfiguration...
{"IPv6 addresses": ["fe80::204:2519:1801:c267"]}
DS18 initialized successfully
```

Everything worked! We successfully read the temperature value of the ds18 sensor connected to the server, via issuing a request from the client.