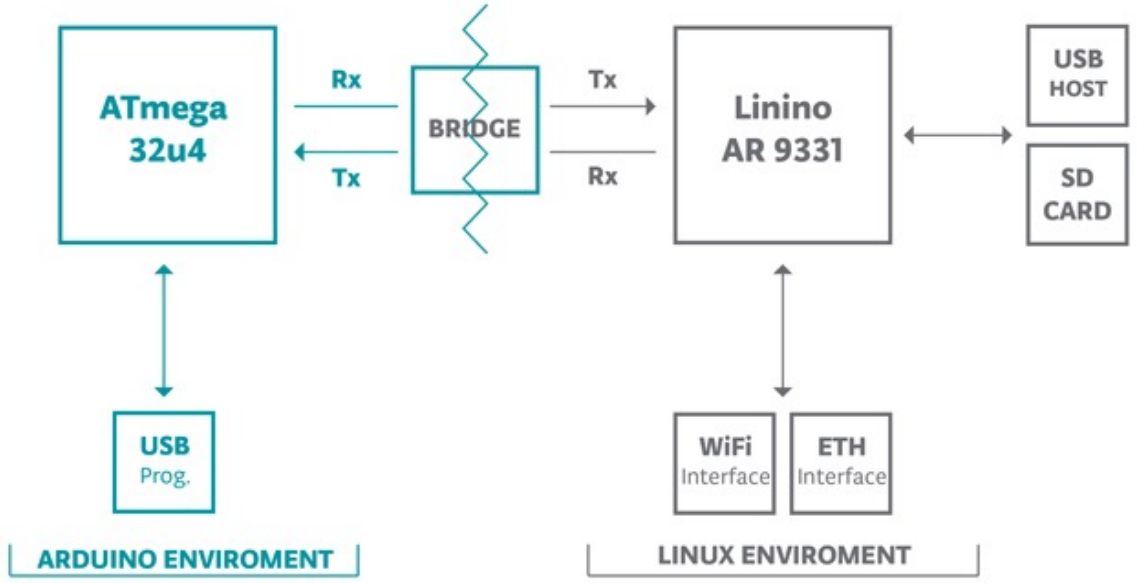
Tecnologie per IoT

Daniele Pagliari
Luca Barbierato

Lab1: Hardware

Overview

- All three exercises in Part3 use the same HW components
  - One of the two LEDs
  - The temperature sensor
  - The Serial interface (for debugging and error reporting)
  - The **WiFi interface** of the Yùn (accessed via the Bridge port)
- The goal is to let the Yùn communicate via REST and MQTT



Overview

- Double-check that your Yùn is connected to your home WLAN (or smartphone hotspot)
  - The same network that your PC is using
  - Check the slides of Part 1 for instruction on setting up the Yùn on-board WiFi.

PART3: EXERCISE 1

The Bridge Library

- An Arduino library which contains all code needed to interface with the Linux processor
- Reference: here

- In this exercise we will use two components of the library
  - BridgeServer (to setup an HTTP server on the Yùn)
  - BridgeClient (to handle individual requests to the server)

New Code Elements

- Include required libraries

```
#include <Bridge.h>
#include <BridgeServer.h>
#include <BridgeClient.h>
```

- Declare global `BridgeServer` instance

```
BridgeServer server;
```

- Setup the server

```
void setup() {
  //...usual setup (pin mode, etc.)...
  pinMode(INT_LED_PIN, OUTPUT);
  digitalWrite(INT_LED_PIN, LOW);
  Bridge.begin();
  digitalWrite(INT_LED_PIN, HIGH);
  server.listenOnLocalhost();
  server.begin();
}
```

Init. bridge conn.

Start server

Use internal LED to understand when the bridge is ready (WiFi connection takes a while on first boot)

New Code Elements

- In `loop()`, process requests using `BridgeClient`

```
void loop() {
  BridgeClient client = server.accept();

  if (client) {
    process(client);
    client.stop();
  }

  delay(50);
}
```

Accept a connection

Close the connection

Avoid overloading with too many connections

New Code Elements

- Read from client as any other `Stream` (exactly as the `Serial` class)

```
void process(BridgeClient client) {

  String command = client.readStringUntil('/');
  command.trim();

  if (command == "led") {
    int val = client.parseInt();
    if (val == 0 || val == 1) {
      digitalWrite(LED_PIN, val);
      printResponse(client, 200, senMlEncode(F("led"), val,F("")));
    } else {

    // etc....
}
```

Parse 1st URL element

Delete \n or similar

Parse 2nd URL element

`F()` around a string constant saves it in FLASH rather than in RAM!!

New Code Elements

- Print the header and body of the HTTP response

```
void printResponse(BridgeClient client, int code, String body) {
  client.println("Status: " + String(code));
  if (code == 200) {
    client.println(F("Content-type: application/json; charset=utf-8"));
    client.println(); //mandatory blank line
    client.println(body); //the response body
  }
}
```

HTTP response code

Here `Content-type` simply helps having a better formatting in the browser (in the next exercise it will be fundamental)

New Code Elements

- The body must be encoded in SenML JSON format

```
{
  “bn”: “Yùn”
  “e”: [
     {
     “n”: <”temperature”>/<”led”>,
     “t”: <timestamp using millis()>,
     “v”: value,
     “u”: “Cel”/null
     }
  ]
}
```

- You can do this “by hand” (it’s just a concatenation of strings) or using the `ArduinoJson` library (reference: here)
  - Solution 1: **smaller memory**
  - Solution 2: **easier** (especially decoding) and more readable

ArduinoJson

- Install `ArduinoJson` from the Library Manager (see part 1 slides)
- Include it in your sketch

```
#include <ArduinoJson.h>
```

- Declare a global dynamic JSON document object with the correct capacity:

```
const int capacity = JSON_OBJECT_SIZE(2) + JSON_ARRAY_SIZE(1) + JSON_OBJECT_SIZE(4) + 40;
DynamicJsonDocument doc_snd(capacity);
```

  - One object with 2 fields (“bn” and “e”), of which “e” is an array with 1 element, and the element contains 4 fields.
  - Plus, 40 additional chars to store string characters (e.g. “temperature”, “led”, “Cel”, etc.)
  - In general, you can use this tool to get the appropriate capacity for a given JSON string

ArduinoJson

- Create your record using a syntax similar to Python dictionaries:

```
String senMlEncode(String res, float v, String unit) {
  doc_snd.clear();
  doc_snd["bn"] = "Yun";
  if (unit != "") {
    doc_snd["e"][0]["u"] = unit;
  } else {
    doc_snd["e"][0]["u"] = (char*)NULL;
  }
  // etc....


  String output;
  serializeJson(doc_snd, output);
  return output;
}
```

Important: clear document memory

Assign JSON fields

Handle NULL fields

Generate JSON string (to be used as body of the HTTP response)

PART3: EXERCISE 2

The Bridge Library (cont'd)

- In this exercise we will use a new component of the Bridge library called `Process`

- `Process` can be used to run Linux shell commands from the Arduino
  - We'll use it to send HTTP requests to a CherryPy server using a Linux tool called `curl`

- In the Bridge library there is also a `HttpClient` class, which however is not flexible enough for our purposes:
  - Does not support `Content-type` specification and other options
  - Still based on `curl`

New Code Elements

- Include the library:

```
#include <Process.h>
```

- Call `Bridge.begin()` in `setup()` as before
- Format the temperature in senML as before
- Send a POST request using the following `curl` command:

```
curl –H "Content-Type: application/json" –X POST
–d <JSON string> <url>
```

- Where:
  - `-H` : extra HTTP header
  - `-X` : specify request type (GET, POST, etc.)
  - `-d` : specify body content (for POST requests)
  - `<url>` : destination (`http://<PC IP Address>:<port>/log` in our example)

New Code Elements

- The previous curl command translates to the following calls to the Process class in Arduino:

```
int postRequest(String data) {
  Process p;
  p.begin("curl");
  p.addParameter("-H");
  p.addParameter("Content-Type: application/json");
  //etc...
  p.addParameter(url);
  p.run();

  return p.exitValue();
}
```

New Code Elements

- You also have to modify the cherrypy servers developed in Exercises 1 and 2 of the SW lab to handle both POST and GET requests to the resource:

  `http://<PC IP Address>:<port>/log`

- Nothing special there, you just need to store all logged data and do some JSON `loads()` and `dumps()`….

PART3: EXERCISE 3

MQTTclient

- In this exercise we will use a contributed library to publish and subscribe to MQTT topics from the Arduino, called `MQTTclient`
- Available in the course “materials” page. Documentation on Github.
- Better features than those available in the Library Manager
- Under the hood, it’s based on the `Process` class seen in the previous exercise
  - In particular, it uses Eclipse Mosquitto utilities (`mosquitto_pub` and `mosquitto_sub`)

Installing a Custom Library

- From the Arduino IDE, select:

  `Sketch/Include Library/Add .ZIP Library`

- Select the .zip archive provided in the materials page

- Then, include the library in your sketch:

```
#include <MQTTclient.h>
```

New Code Elements

- This time we need two JSON objects (one for sending, one for receiving)

```
const int capacity = JSON_OBJECT_SIZE(2) + JSON_ARRAY_SIZE(1) + JSON_OBJECT_SIZE(4) + 40;
DynamicJsonDocument doc_rec(capacity);
DynamicJsonDocument doc_snd(capacity);
```

New Code Elements

- Setup the MQTT client

```
void setup() {
  //usual stuff....


  digitalWrite(LED_PIN, LOW);
  Bridge.begin();
  digitalWrite(INT_LED_PIN, HIGH);
  mqtt.begin("test.mosquitto.org", 1883);
  mqtt.subscribe(my_base_topic + String("/led"), setLedValue);
}
```

Connect to a broker specifying URL and port (we will use Mosquitto's test broker for this lab)

Subscribe to a topic using its name (`tiot/<group-id>/led` in this example). The second argument is a callback function, invoked when new data are available

New Code Elements

- MQTT subscribe callback:

Topic name

Subtopic name
(if you use wildcards)

Message

```
void setLedValue(const String& topic, const String& subtopic, const String& message) {


  DeserializationError err = deserializeJson(doc_rec, message);
  if (err) {
    Serial.print(F("deserializeJson() failed with code "));
    Serial.println(err.c_str());
  }
  if (doc_rec["e"][0]["n"] == "led") {
    //etc...
```

ArduinoJson function
to decode a JSON string

You can access the message fields
again as a Python-stile dictionary

New Code Elements

- In the `loop()` function:

```
void loop() {
  mqtt.monitor();

  //usual stuff...
  String message = senMlEncode("temperature", temp, "Cel");
  mqtt.publish(my_base_topic + String("/temperature"), message);

  delay(1000);
}
```

Get messages on subscribed topics (invoking the callback)

Publish a message to a topic

Mosquitto on the Yùn

- The Mosquitto client should be already installed on the Yùn's Linux processor
- If it isn't follow the instructions on the MQTTclient library's Github page on how to install it
- You'll need to connect to the Yùn's Linux processor using:

```
ssh root@arduino.local
```

- The password is the one set on the web page of the Yùn during the first config.

Testing Exercise 3.3

- To test the exercise, you'll need to install the Mosquitto client also on your PC (instructions can be found here)

- Subscribe command (to read temperature logs):

```
mosquitto_sub -h test.mosquitto.org -t '/tiot/0/temperature'
```

- Publish command (to turn ON the LED):

```
mosquitto_pub -h test.mosquitto.org -t '/tiot/0/led' -m '{"bn":
"Yun", "e": [{"n": "led", "t": null, "v": 1, "u": null}]}'
```