IFTTT Project- Technical documentation

Group 1: E. DUREL – R. FARGEON – T. GABORIT – B. MEHAT

Table of contents

I. Introduction

This technical document explains how our automation platform solution works. This automation platform supports "if-this-then-that" functionalities, where microcontrollers (with input and output devices/peripherals) send events and receive commands to/from a remote server.

The document first focuses on the design, explaining its specifications. Then, the hardware and software parts are described and usage instructions are provided to the user, to facilitate the setup and use. Moreover, derivations are provided to give key changes and modifications from the previous design version.

II. Design Specification

1. System components

The system is composed of a server hosted on Amazon Web Services (AWS), and several microcontrollers, which communicate with the server using the Message Queuing Telemetry Transport protocol (MQTT). The server is hosting a Node-RED-based web interface where the user can parametrize the microcontrollers for IoT-purpose applications. Node-RED is used both as a Graphical User Interface (GUI) and a controller of the communication between devices.

Microcontrollers can handle a set of peripherals such as Light-Emitting Diodes (LED), switches, potentiometer, or various external I/O devices. Microcontrollers are connected to the internet (and therefore to the server) through ESP-01 Wifi modules. These devices (microcontroller and WIFI module) communicate with each other via UART serial transmission. A microcontroller "template code" has been implemented to support communications (UART, Wifi, MQTT) and low-level interactions with the inputs and outputs. This is the program for the clients.

2. Communication protocol

To make interactions easy between microcontrollers and server, MQTT protocol is used. This is a high-level easy-to-use protocol that allows some "publishers" to send messages to a "broker" (server) via a specific "topic" (subject of the message). Then, the broker receives messages and forwards them to appropriate devices, called "subscribers", which are devices subscribed to the associated topic. The protocol is a well-adapted tool for IoT-purpose applications, as it offers low memory, low consumption, and low processing requirements. The server and microcontrollers are then communicating with each other by sending MQTT messages on various topics, following a specific message format.

3. Design diagram

This diagram is useful to understand the global system organization. Note that for our system, the AWS server is hosting both Node-RED user interface and MQTT broker. A Microcontroller can send information to the server through MQTT messages, using its associated ESP module (left side of the diagram). The server can send information to a microcontroller through MQTT messages.

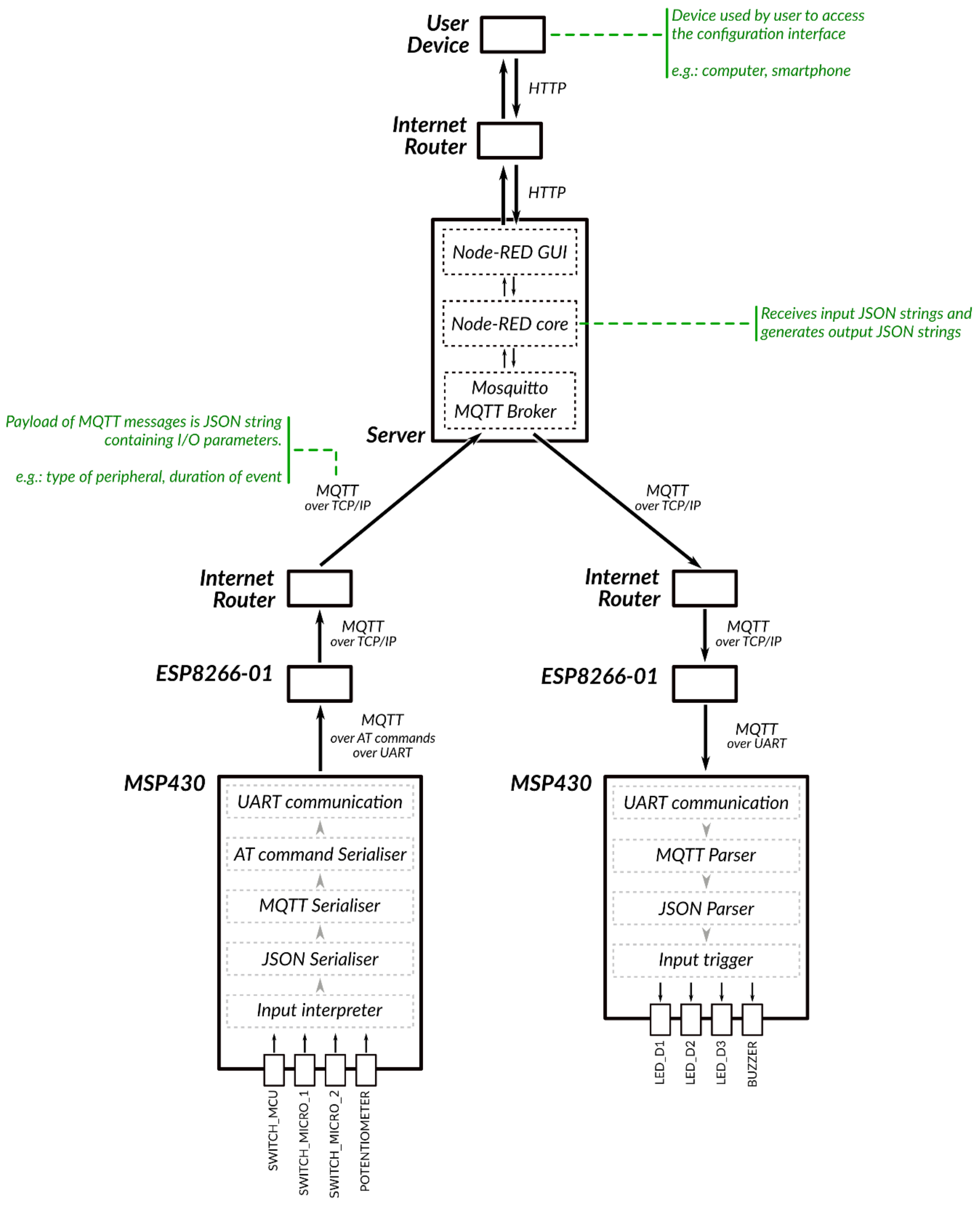


Figure 1 - Design diagram of the whole system

The MQTT broker, when receiving an input message on a specific topic, forwards this message to all subscribed devices. There is one “OUTPUT” topic per microcontroller. The server gives information to a specific microcontroller through its dedicated output topic. These multiple topics prevent a microcontroller from receiving and processing messages dedicated to other devices. However, all microcontrollers publish to the same "INPUT” topic, as the server wants to receive information from all microcontrollers. This simple message organization makes the whole communication process robust.

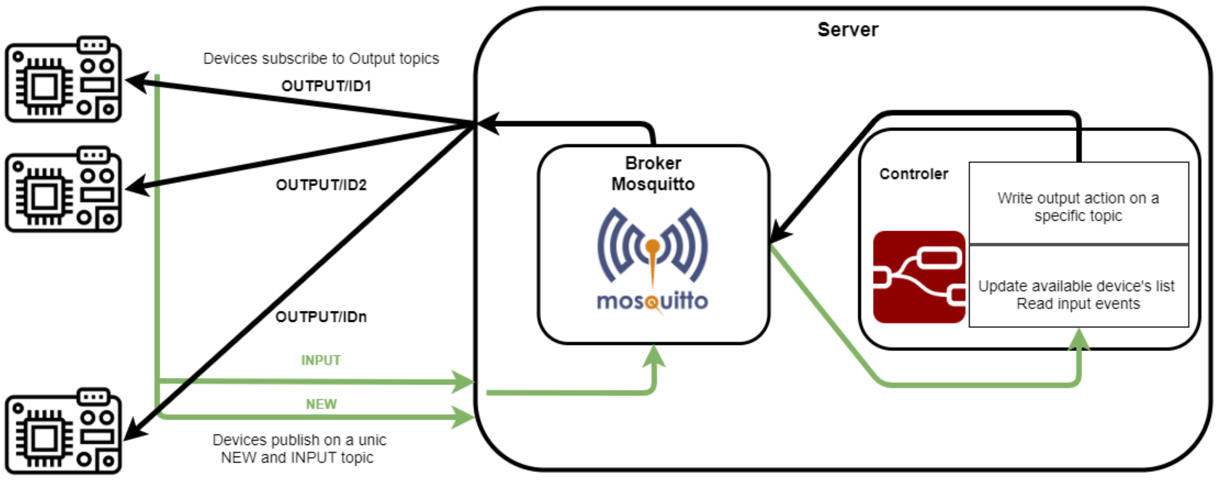

Server
Devices subscribe to Output topics
OUTPUT/ID1
OUTPUT/ID2
OUTPUT/IDn
INPUT
NEW
Devices publish on a unic NEW and INPUT topic
Broker Mosquitto
mosquitto
Controler
Write output action on a specific topic
Update available device's list Read input events

Figure 2 - Communication architecture design

III. Hardware descriptions

1. Circuit Diagram

Each microcontroller has two I/O ports (P1 and P2), each providing 8 pins (from 0 to 7) to which external devices can be connected. However, some pins are reserved and cannot be used by the user. The table below lists all pins and, if they are reserved, their usage. Pins with no specific usage can be used to connect external peripherals to the microcontroller.

Port P1	Usage	Port P2	Usage
P1.0	LED D1	**P2.0**	-
P1.1	UART RX	**P2.1**	LED D3
P1.2	UART TX	**P2.2**	-
P1.3	Switch MCU	**P2.3**	LED D3
P1.4	-	**P2.4**	-
P1.5	-	**P2.5**	LED D3
P1.6	LED D2	**P2.6**	-
P1.7	-	**P2.7**	-

Table 1 - List of pins and their usage

- Transmission between microcontroller and WIFI module is done using a UART serial transmission. UART transmission requires pins P1.1 and P1.2 for the receive and transmit cables.
- LEDs D1, D2 and D3 are embedded on the board and are connected to pins P1.0, P1.6, P2.1, P2.3 and P2.5.
- Embedded switch (Switch MCU) is hard-connected to P1.3.

Devices (switches, LEDs, buzzer, and potentiometer) can be connected to all other I/O pins from ports 1 and 2 (under several restrictions such as ADC link compatibility for the potentiometer port). The employed devices and their pin connection should be specified in a header code as explained in the software description and usage instructions. This process is making the microcontroller usage generic and flexible for the user's requirements.

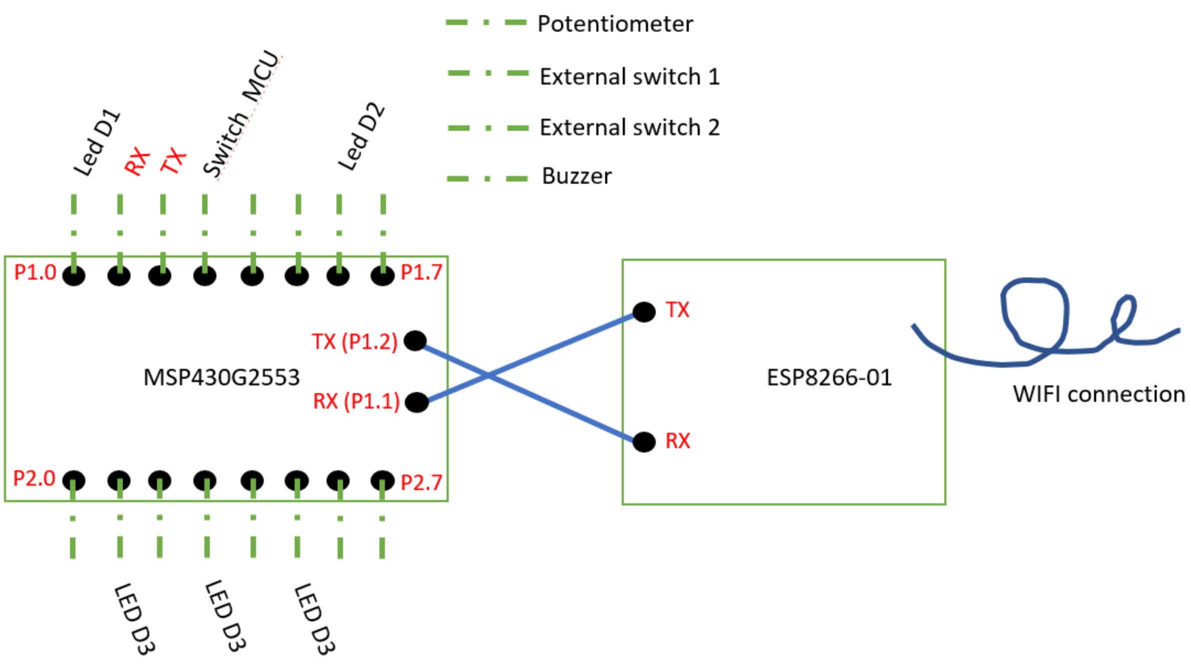


Figure 3 - Circuit diagram

2. Instructions

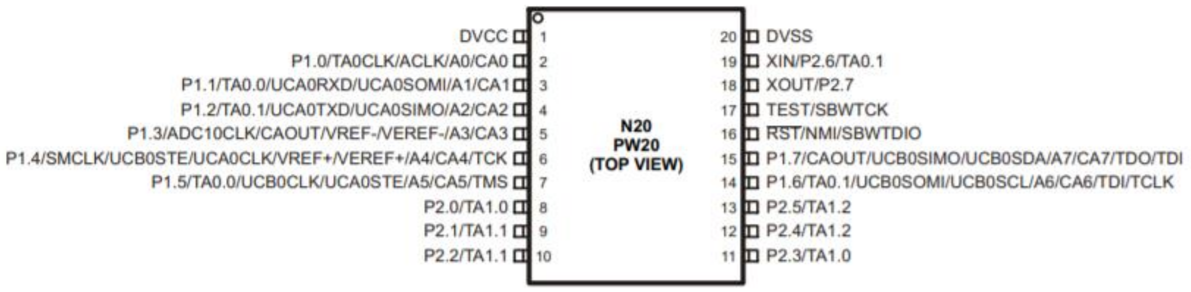

DVCC 1
P1.0/TA0CLK/ACLK/A0/CA0 2
P1.1/TA0.0/UCA0RXD/UCA0SOMI/A1/CA1 3
P1.2/TA0.1/UCA0TXD/UCA0SIMO/A2/CA2 4
P1.3/ADC10CLK/CAOUT/VREF-/VEREF-/A3/CA3 5
P1.4/SMCLK/UCB0STE/UCA0CLK/VREF+/VEREF+/A4/CA4/TCK 6
P1.5/TA0.0/UCB0CLK/UCA0STE/A5/CA5/TMS 7
P2.0/TA1.0 8
P2.1/TA1.1 9
P2.2/TA1.1 10
N20
PW20
(TOP VIEW)
20 DVSS
19 XIN/P2.6/TA0.1
18 XOUT/P2.7
17 TEST/SBWTCK
16 RST/NMI/SBWTDIO
15 P1.7/CAOUT/UCB0SIMO/UCB0SDA/A7/CA7/TDO/TDI
14 P1.6/TA0.1/UCB0SOMI/UCB0SCL/A6/CA6/TDI/TCLK
13 P2.5/TA1.2
12 P2.4/TA1.2
11 P2.3/TA1.0

Figure 4 - MSP430G2553 pins specifications

The figure 4 is extracted from the MSP datasheet and represents the microcontroller pins with the links/properties specific to every pin. Please refer to this figure when choosing a pin for a specific device.

IV. Software description

1. Microcontroller

The microcontroller software is split into two main parts. The first code section is managing the inputs and outputs at a register level. This I/O manager interacts with another higher-level section that interacts with the ESP module to receive and transmit MQTT messages and communicate with the server. Both sections are explained below.

a. I/O manager

The low-level code section is the I/O manager, which implements some basic register functions to set up and drive the inputs and outputs (*board_setup.c*, *outputs.c*). These functions are often generic, such that any input or output device can be parametrized no matter its port and pin connection. The implementation just needs the port and pin information for the connected tools. To easily provide this type of information, a header file (*config.h*) contains a set of *#define* statements to indicate which devices are used, and what their port and pin connections are. These *#define* statements can be modified by the user to a specific usage. The code relative to each tool is generic, so it is adapted to any port/pin configuration - using some pre-processing statements such as *#define* or *#ifdef* to adapt the code to the system at compilation.

The user only needs to modify the *config.h* header file when changing the board hardware configuration, thanks to code genericity. The procedure is described in the usage instruction section of this technical document. This low-level code section also contains the interrupt routines implementation as well as the main function implementation. Each interrupt routine is coded inside an *#ifdef* statement to use it only when an associated input tool (or a timer) is used to trigger an event. They mainly set a software interrupt flag and exit the low-power mode to process the interrupt in the main function. This main function is composed of several *if* statements to handle possible interruptions triggered by all the connected input devices. *Timer0* is used to generate an interrupt every millisecond to handle periodic actions (high-level section).

The low-level code presented above is interacting with a higher-level code section (Communication manager) using two functions. These functions are used to publish a message to the server when an input is triggered and to trigger an output when a message (command) is received from the server. These two functions are presented below, with their parameters as defined in their respective headers (inputs and outputs).

void input(input_type type, input_event event, uint8_t input_intensity, uint8_t input_duration)

Parameters:

input_type	Input component the event comes from
input_event	Event which occurred.
input_intensity	Intensity of the event to be transmitted (e.g. potentiometer value)
input_duration	Duration of the event (e.g. switch hold)

enum input_type:

INPUT_SWITCH_MCU	Switch mounted on the microcontroller PIN1.3
INPUT_SWITCH_MICRO_1	External switch n°1 connected to PINX.X
INPUT_SWITCH_MICRO_2	External switch n°2 connected to PINX.X

INPUT_POTENTIOMETER	Potentiometer connected to PINX.X

enum input_event:

INPUT_PRESS	The switch has been pressed.
INPUT_RELEASE	The switch has been released. The duration of press should be transmitted.
INPUT_POTENTIOMETER_UPDATE	Reading of the potentiometer value.

void output(output_type type, output_trigger trigger, uint8_t intensity uint8_t duration)

Parameters:

output_type	Output component to trigger.
output_trigger	Which trigger should be processed on the output component
output_intensity	Intensity of the event to be triggered (e.g. LED intensity)
output_duration	Duration of the trigger

enum output_type:

OUTPUT_LED_D1	LED D1 mounted on PIN1.0 of MCU
OUTPUT_LED_D2	LED D2 mounted on PIN1.6 of MCU
OUTPUT_LED_D3	LED D3 including all its color components, connected to PIN2.3, PIN2.1, PIN2.5.
OUTPUT_LED_D3_GREEN	LED D3 as a green LED
OUTPUT_LED_D3_RED	LED D3 as a red LED
OUTPUT_LED_D3_BLUE	LED D3 as a blue LED
OUTPUT_LED_D3_PURPLE	LED D3 as a purple LED
OUTPUT_BUZZER	Buzzer connected to PINX.X

enum output_trigger:

OUTPUT_ON	The output component is on.
OUTPUT_OFF	The output component is off.

ERRATA

Custom *Input* API	Custom *Output* API
~~Texas Instruments *DriverLib* API~~ - Handmade code	
Register level	

Note that the DriverLib API mentioned in the interim report to handle the I/O (register level) management is no longer used, as it is not compatible with the MSP430G2553 board used. A handmade code is used instead.

b. Communication manager

This high-level code section is responsible of the communication between the microcontroller and the server through the ESP-01 chip. It can be broken down into four distinct layers (from highest-level - closest to ESP-01 - to lowest-level - closest to I/O manager):

- UART layer
- AT layer
- MQTT layer
- JSON layer

The following paragraphs detail the operations each of these parts performs.

JSON Layer

The lowest-level part is responsible for handling JSON messages which are sent and received to and from the server. JSON messages are used to transmit input and output parameters. This format has been used to permit a compatibility with the server data interpreter. Also, using this format permitted to use the characters '{' and '}' which delimit the JSON message to extract the message easily in the reception buffer.

When used in input mode (i.e., microcontroller sends input event to the server), a JSON serializer generates a JSON message containing input parameters given by the I/O manager (e.g., type of input, type of event, duration of the event, etc.). JSON input messages are then forwarded to the higher-level communication layer (MQTT layer).

When in output mode (i.e., microcontroller receives instructions from the server), a JSON parser interprets messages received from MQTT layer and informs the I/O manager about the output parameters (e.g., type of output peripheral, type of event, intensity of the event, etc.).

MQTT Layer

This second layer is responsible for handling MQTT messages which are sent and received to and from the server. MQTT messages encapsulate JSON messages to allow the use of the MQTT environment.

At initialization (i.e., on system boot), the MQTT layer generates three special MQTT messages to initialize the communication with the server. First, an MQTT "CONNECT" message is generated to connect the MQTT client (i.e., the microcontroller) to the MQTT broker (i.e., the MQTT server). Then, an MQTT "PUBLISH" message is generated. This message holds the unique identifier of the microcontroller and is sent to the server to notify it of the new connection (more details about this feature are given in the Server section of this document). Finally, the last MQTT message generated is a "SUBSCRIBE" message which will notify the server that the microcontroller wants to receive messages from a specific MQTT topic, in this case, the output topic associated with the device. Each of these three MQTT initialization messages is forwarded to the higher-level layer (AT layer) after generation.

When used in input mode, JSON messages generated by the JSON layer are encapsulated into MQTT PUBLISH frames. A PUBLISH frame is a type of MQTT frame used to publish data onto a specific MQTT *topic* (in this case, the "INPUT" topic). Once the MQTT message is generated, it is forwarded to the AT Layer.

When used in output mode, an MQTT parser extracts the JSON message from the received MQTT PUBLISH frame and gives it to the lower-level JSON parser.

AT command Layer

The third layer is responsible for encapsulating MQTT frames into AT commands. The AT command set is a set of commands used to communicate with the ESP-01 Wifi module. These commands enable to easily ask the Wifi module to connect to a Wifi hotspot or to start sending data to a specific host.

At initialization, this layer generates AT commands for asking the ESP-01 Wifi module to connect to Wifi hotspot as well was initiating a TCP connection with the server.

When used in input mode, each time an MQTT frame has to be sent, AT commands are generated to send the MQTT PUBLISH frame to the ESP-01 through UART (highest-level layer, see next section).

UART Layer

Since the microcontroller and the ESP-01 Wifi module are connected by a serial connection, the serial communication protocol UART is used. The highest-level layer of the microcontroller code is responsible for the UART communication between the two entities.

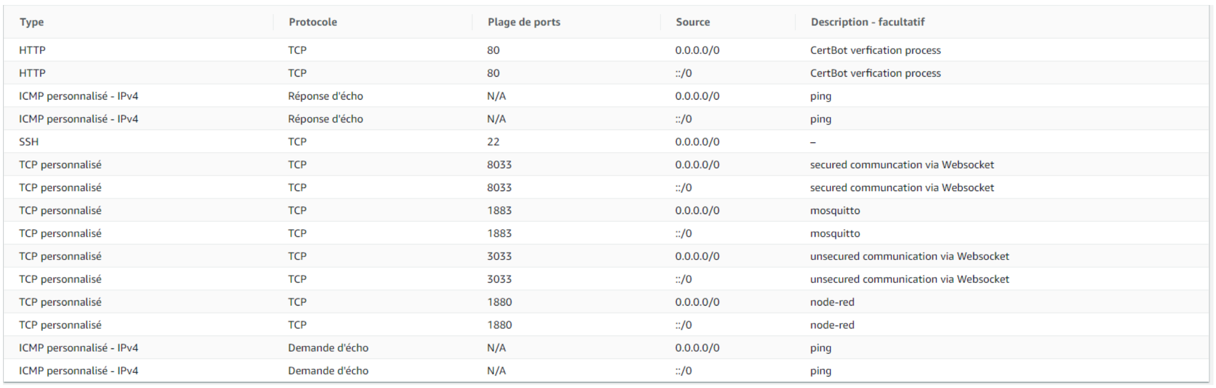
When used in input mode, this layer sends AT commands and MQTT messages to the Wifi module through UART protocol. It also handles return codes sent by the ESP-01 after each action (e.g., "device connected to Wifi hotspot", "message successfully sent").

When used in output mode, the UART layer receives data on the serial communication with the ESP and parses it to extract the MQTT message contained in it. MQTT messages are then forwarded to the MQTT Parser for interpretation.

2. Server

a. Virtual Private Server (VPS) instance

The server used for this project is a virtual machine hosted by Amazon Web Services. This virtual machine hosts a Linux distribution and can be accessed from any internet access using an SSH connection. AWS allows using a free Elastic Compute Cloud (EC2) virtual machine. This Virtual Private Server (VPS) can be accessed through the internet using its Domain Name System (DNS) address. To use the internet applications on our VPS instance we must open some network ports. For this project we opened port 22 to allow SSH communication, port 80 to allow HTTP communication, port 1883 to communicate with *Mosquitto* from the internet, and port 1880 to communicate with Node-RED from the internet. Figure 5 summarises the security rules applied to our server.



Type	Protocole	Plage de ports	Source	Description - facultatif
HTTP	TCP	80	0.0.0.0/0	CertBot verfication process
HTTP	TCP	80	::/0	CertBot verfication process
ICMP personnalisé - IPv4	Réponse d'écho	N/A	0.0.0.0/0	ping
ICMP personnalisé - IPv4	Réponse d'écho	N/A	::/0	ping
SSH	TCP	22	0.0.0.0/0	–
TCP personnalisé	TCP	8033	0.0.0.0/0	secured communcation via Websocket
TCP personnalisé	TCP	8033	::/0	secured communcation via Websocket
TCP personnalisé	TCP	1883	0.0.0.0/0	mosquitto
TCP personnalisé	TCP	1883	::/0	mosquitto
TCP personnalisé	TCP	3033	0.0.0.0/0	unsecured communication via Websocket
TCP personnalisé	TCP	3033	::/0	unsecured communication via Websocket
TCP personnalisé	TCP	1880	0.0.0.0/0	node-red
TCP personnalisé	TCP	1880	::/0	node-red
ICMP personnalisé - IPv4	Demande d'écho	N/A	0.0.0.0/0	ping
ICMP personnalisé - IPv4	Demande d'écho	N/A	::/0	ping

Figure 5 - Summary of the security rules apply to the VPS instance.

b. MQTT broker

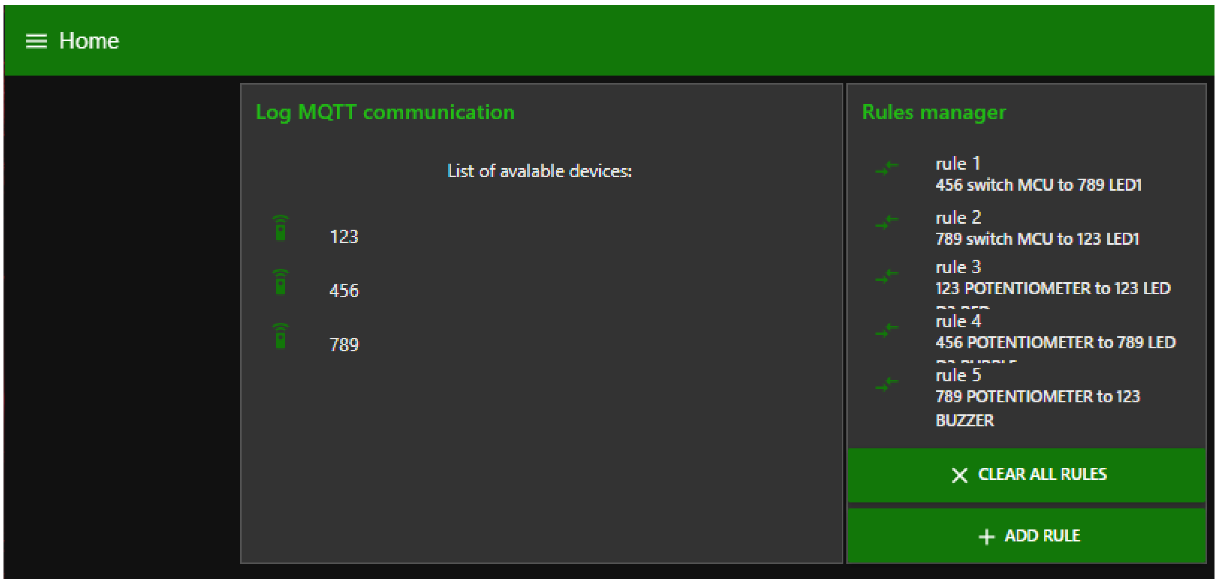
On this VPS it had been installed a *Mosquitto* MQTT broker. This broker uses the protocol MQTT to receive the published messages and forward them to the appropriate subscribed device. The *Mosquitto* broker uses port 1883 of the VPS.

c. Node-RED controller

The controller of the system is based on a Model-View-Controller design. The controller is based on an open-source flow-based development tool tailored for the Internet of Things (IoT) application: Node-RED. Node-RED provides a web-browser programming tool based on visual programming with nodes and offers the possibility to code function using JavaScript. Node-RED also provides the possibility to easily create a "Dashboard" that we can use as a GUI and allows an easy configuration of the input/output rules by the user.

View / control part

The GUI is a web interface generated by Node-RED. Since it is a web interface it is accessible to any user with a web browser and an internet connection. The GUI is composed of two pages: the home page is a global dashboard that displays a list of available devices and the list of rules defined for the system to operate. When clicking on the button add rule, the user is redirected to a rule editing page that allows him to easily associate one input event with a corresponding output action. The control part is also handled by the Node-RED dashboard UI blocks. Those function blocks allow adding text boxes, buttons, sliders, and dropdown lists to the GUI that can be activated by the user with a mouse click. Figure 6 and Figure 7 show two different pages of the GUI as they are displayed to the user.


Home

Log MQTT communication

List of avalable devices:

123

456

789

Rules manager

rule 1
456 switch MCU to 789 LED1

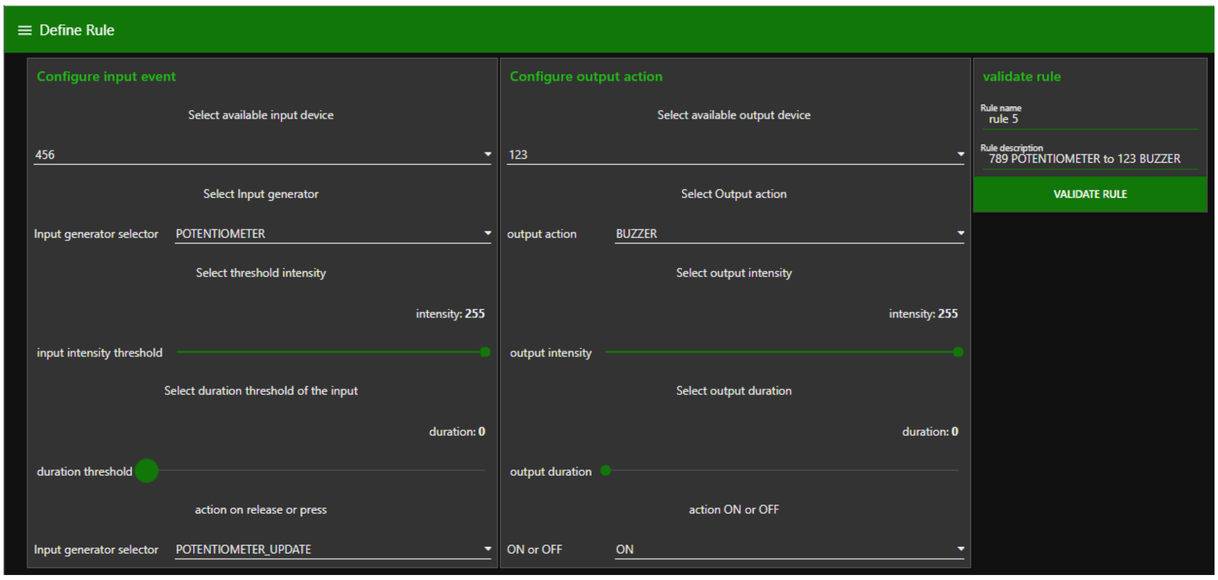
rule 2
789 switch MCU to 123 LED1

rule 3
123 POTENTIOMETER to 123 LED

rule 4
456 POTENTIOMETER to 789 LED

rule 5
789 POTENTIOMETER to 123 BUZZER

Figure 6 - Home page GUI


Define Rule

Configure input event

Select available input device: 456

Select Input generator
Input generator selector: POTENTIOMETER

Select threshold intensity
intensity: 255
input intensity threshold

Select duration threshold of the input
duration: 0
duration threshold

action on release or press
Input generator selector: POTENTIOMETER_UPDATE

Configure output action

Select available output device: 123

Select Output action
output action: BUZZER

Select output intensity
intensity: 255
output intensity

Select output duration
duration: 0
output duration

action ON or OFF
ON or OFF: ON

validate rule

Rule name: rule 5

Rule description: 789 POTENTIOMETER to 123 BUZZER

Figure 7 - Rule editor page GUI

Controller model part

Node-RED uses a flow-based graphical development tool. The development tool is a web browser interface on which the developer can interconnect nodes. The complete model at a node level is displayed in **Erreur ! Source du renvoi introuvable.**. The information is transmitted from a node to another using a JavaScript object "*msg*". The nodes execute their function when they receive this message through their connections.

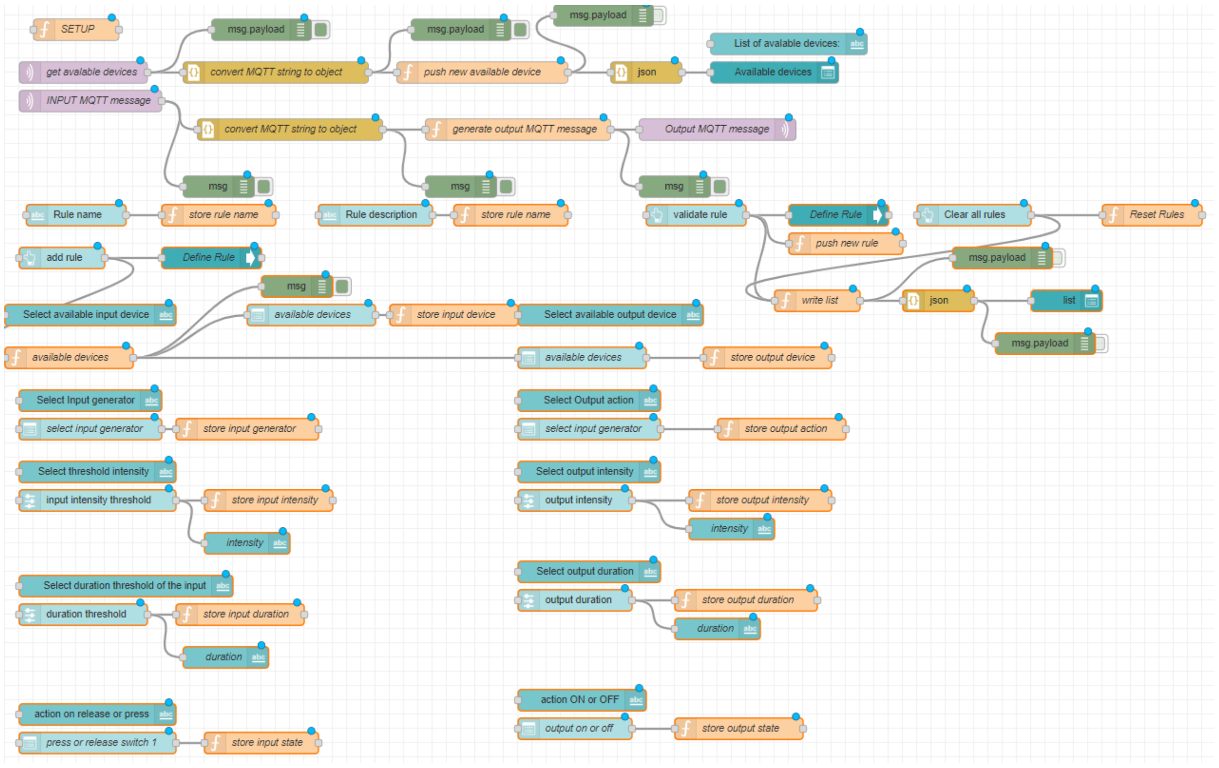


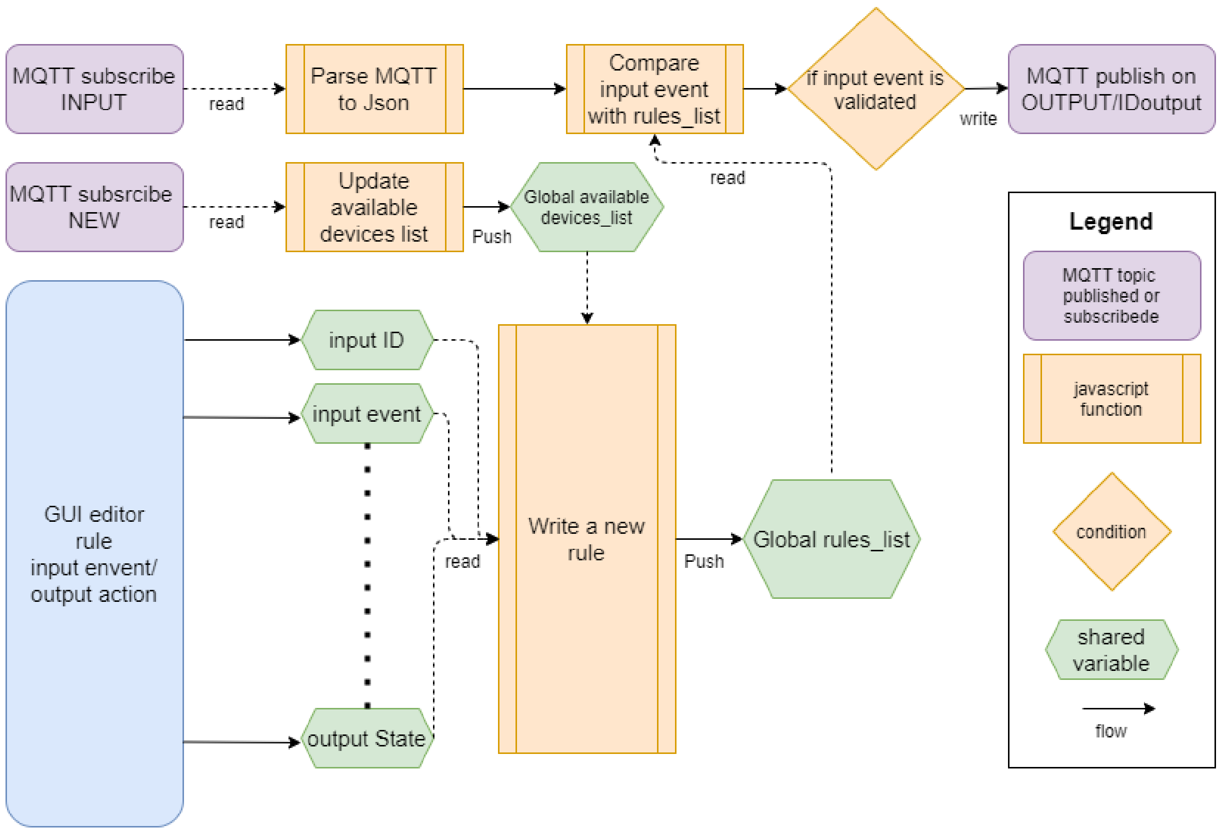
Figure 8 Complete flow-based model developed on Node-RED

The controller is connected to the *Mosquitto* broker through the localhost IP address using port 1883. Two MQTT IN nodes subscribed to the topics *NEW* and *INPUT* listen to the messages emitted by the devices.

Node-RED allows to create local variables on different levels that can be accessed by the nodes. For this project two tables have been defined as global variables. The global variable *devices_list* stores the ID of all devices that have sent a PUBLISH message on the *NEW* topic. The global variable *rules_list* stores an array of structures that store all the information needed to associate an event input with an output action.

When the user edits a new rule, each parameter of the rule is stored into a flow variable that is a local variable with a reduced visibility to a corresponding flow. When the user presses the "*validate rule*" button all the flow variables are aggregated into a rule structure and this structure is pushed in the array *rules_list*.

When a message is received on the *INPUT* topic, the controller parses this message into a JavaScript object. Then the *rules_list* array is browsed and the controller checks if there is a rule associated with the emitting device, the type of event, the intensity, and the duration. If the input event received triggers a rule in the list, an MQTT message containing the API corresponding to the associate output action in JSON is generated and published on the appropriate topic. The following Figure 9 summarizes the overall back-end design of the controller.



MQTT subscribe INPUT
read
Parse MQTT to Json
Compare input event with rules_list
if input event is validated
write
MQTT publish on OUTPUT/IDoutput
MQTT subsrcibe NEW
read
Update available devices list
Push
Global available devices_list
read
Legend
MQTT topic published or subscribede
javascript function
condition
shared variable
flow
GUI editor rule input envent/ output action
input ID
input event
output State
read
Write a new rule
Push
Global rules_list

Figure 9 - Schematic representation of the controller flow-based program

V. Deviations from interim solution

1. DriverLib

At the beginning of the implementation, we noticed that the *DriverLib* library is not compatible with our MSP430 version. Several peer-review documents pointed out this mistake. Here is a list of supported devices for the library, found on the TI resource center:

- MSP430F5XX_6XX
- MSP430FR2XX_4XX
- MSP430FR57XX
- MSP430FR5XX_6XX
- MSP430I2XX

To bypass this non-compatibility, we have implemented our own low-level set of functions to modify the microcontroller registers with several set/setup functions for every possible input/output type.

2. Peer-review comments

Several peer-review documents have indicated the lack of flexibility of our solution regarding the poor input/output tool possibilities. In fact, we chose to keep a small list of input/output types and events. It may reduce the possibilities but keeps the system easy to use, and robust. Also, the code genericity and flexibility (for port and pin management and tools use) increases the general I/O flexibility. Furthermore, the user can quickly extend the possibilities by adding a new device entity (using a define statement) with a free pin, it only requires a couple of “copy and paste”.

VI. Usage instructions

1. Server instructions

a. Connection to the user interface.

The user can access the web-based configuration interface using any web browser. The connection is made using the server's DNS address with the port corresponding to the Node-RED core (1880). For the final configuration of the server, it corresponds to the following address: http://3.8.210.204:1880/ui.

b. Rule Management.

On the home page, the users can visualize a list of all the satellite devices already connected to the system. The management of association between an input events and output actions is handled using rules. A rule associates a specific source of input event with a specific action. By clicking on the "*Add Rule*" button on the home page, the user is redirected to the rule editing page.

The rule editing page is divided into 3 sections. The first section, on the left-hand side, is used to tune the parameters of the input event that will trigger the output reaction. Those parameters are the id of the device that must generate the event, the type of event observed (switch, potentiometer, etc.), the intensity threshold for the potentiometer (the event will trigger a reaction only if the intensity is higher than this value), the duration threshold for the switch event and finally a state (push, release, or potentiometer update).

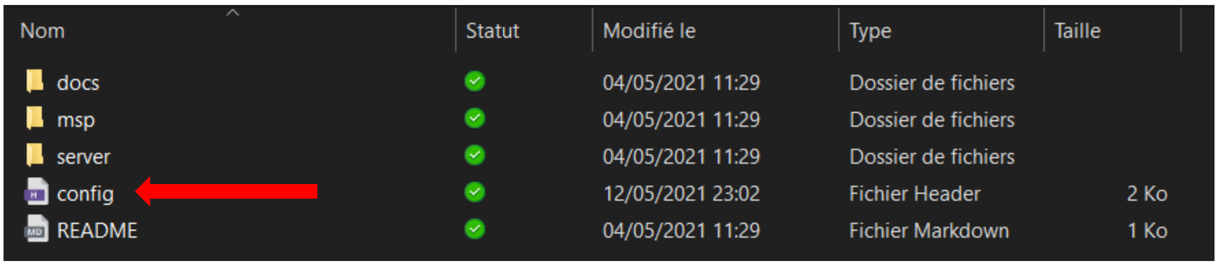
After defining the input event, the user can use the middle part of the page to fill in the information corresponding to the output reaction. That information is, the id of the reacting device, the type of reaction (LED, color LED, buzzer), the intensity of the reaction, and the duration of the reaction.

The last part of the rule editing page is used to give a name and a description to this rule and displays a "*validate rule*" button that applies this new rule to the system.

A list of already created rules is displayed on the home page and those rules can be cleared using the "*clear*" button.

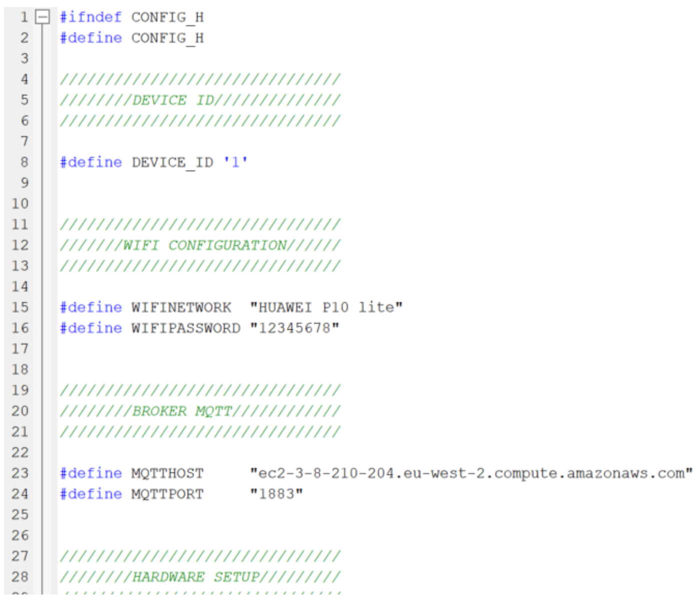
2. Microcontroller instructions

On the microcontroller side, the configuration can be handled in the *config.h* file. This file can be found at the root of the project. In this file, the user can modify the configuration of the system comprising several parameters.



Nom	Statut	Modifié le	Type	Taille
docs	✓	04/05/2021 11:29	Dossier de fichiers	
msp	✓	04/05/2021 11:29	Dossier de fichiers	
server	✓	04/05/2021 11:29	Dossier de fichiers	
config	✓	12/05/2021 23:02	Fichier Header	2 Ko
README	✓	04/05/2021 11:29	Fichier Markdown	1 Ko

Figure 10 - Location of the config.h file



```
#ifndef CONFIG_H
#define CONFIG_H

//////////////////////////////
////////DEVICE ID/////////////
//////////////////////////////

#define DEVICE_ID '1'


//////////////////////////////
///////WIFI CONFIGURATION//////
//////////////////////////////

#define WIFINETWORK  "HUAWEI P10 lite"
#define WIFIPASSWORD "12345678"


//////////////////////////////
////////BROKER MQTT///////////
//////////////////////////////

#define MQTTHOST     "ec2-3-8-210-204.eu-west-2.compute.amazonaws.com"
#define MQTTPORT     "1883"


//////////////////////////////
////////HARDWARE SETUP////////
```

Figure 11 - Overview of the config.h file

a. Device ID

The ID of the device for which the firmware built will be flashed in is specified with the #define DEVICE_ID statement. The Device ID must be a single character.

b. WiFi Configuration

The WiFi connexion which will be used for the communicating with the MQTT broker can be modified by specifying the name of the network as well as its password with the corresponding #define statements.

c. MQTT Broker

Regarding the MQTT Broker, its address and the port used for the communication can be modified in case another server should be used for the system.

d. Board setup

The user must indicate which port and pin external devices are connected to. To do so, *#define* statements are available for all devices port and pin when necessary. Possible ports are 1 and 2, and there are 8 pins per port (0 to 7). An example is provided below, with the External Switch 1 parameters:

```
// uncomment if an external switch is used, and fill the port and pin
#define EXT_SWITCH_1
#ifdef EXT_SWITCH_1
  #define EXT_PORT_1 1          // port (set to 1 or 2)
  #define EXT_PIN_1 0x10        // pin (mask of the pin position between pin 0 and 7)
#endif
```

The line where the *EXT_SWITCH_1* is defined can be commented or uncommented to use an external switch or not. If the statement is commented, all the code lines associated with the external switch are discarded during pre-processing steps thanks to *#ifdef* statements encapsulating the external *EXT_SWITCH_1* code.

If the external switch is used, *EXT_PORT_1* and *EXT_PIN_1* macros are defined and refer to the external switch port and pin. The user has to set the appropriate values corresponding to the hardware setup. Then, the rest of the code is using the defined macros when interacting with the external switch.

The user must be careful to avoid connecting devices on embedded LED pins.

e. Firmware compilation and download

Once the desired configuration has been set for the device, the project must be built in IAR Embedded Workbench IDE and downloaded to the board.

Table of Figures