Project report: Connectivity infrastructure

Guilherme R. C. - 169127

April 7th, 2019

1 Purpose

The purpose of this module was to build an IOT connectivity infrastructure using the MQTT protocol, in which the development of the two following modules can rely on. The program has to provide (i) an automated communication establishment process and (ii) the functionality of exchanging messages using topics ([1]).

2 Materials

- EA076 kit, including a platform board with multiple peripheral connections (we used the female DB9 in this project module), (ii) wire wrapping wire and (iii) tools for wire wrapping
- FRDM-KL25Z: Freedom Development Platform for Kinetis, featuring MKL25Z128VLK4 MCU – 48 MHz, 128 KB flash, 16 KB SRAM, USB OTG (FS), 80LQFP
- ESP8266 ESP-01 WiFi module (programmed with [2])
- SERIAL-TO-TTL converter (for debugging purposes)

3 Hardware design

The hardware project, designed in EAGLE, was sent in "Roteiro 1".

4 Software design

First of all, it's important to state that the communication with the MQTT broker is abstracted by a firmware programmed in the ESP01 module. This firmware implements a protocol layer ([2]) over the MQTT protocol ([1]. So, the microcontroller actually talks with ESP01 module using the former protocol, which then talks with the MQTT broker using the latter protocol (and vice versa). From now on I'll be using ESP01 module and MQTT broker interchangeably.

The communication with the MQTT broker can be divided into two different global states: 'establishment' and 'communication indeed'. The former follows a well-defined sequence of message exchanging, while the latter acts more like an event-oriented state. Both of them were implemented using a finite state machine, which easily allows connection retries and message handling that rely on context (that is, the same message can have a different "meaning" upon a different context).

NOTE: Since I'm using an external serial communication program - called *moserial* ([3]) - that can send the message-terminating chars specified in [2], I didn't need to handle this in the code.

4.1 Processor Expert (PE) components

In this project the following PE components were used: UART0 and UART2. I decided to not use the Utility component since it just wraps the libc string functions, which would only add more delay to the communication.

4.2 Communication algorithm

The main goal of the software design was to be as non-blocking as possible in each peace of the communication process. This enables a more responsive behavior from the microcontroller side.

Therefore, both UART0 and UART2 are using interruptions to deal with the characters being received/sent and I tried to put as less code as I could in the interruption handlers. The only peace of code that inevitably uses polling is the main loop, which keeps checking a global variable that says when a valid message was received, and then treats it.

The algorithm can be divided into four main steps:

Step 1: Receiving a message

When a character is received, a interruption is generated and the handler adds it to a linear buffer. This occurs until a sequence of message-terminating chars is received, in which case a global flag is set indicating that a valid message was completely received.

Step 2: Parsing the received message

The main loop keeps checking if a valid message was received (using the global flag cited above). If it is, then (i) a parse function is called to interpret the message and change the communication state (if needed); (...)

Step 3: Assembling the response message

(...) and thereafter (ii) a response function is called, which assembles the response message and triggers the sending process.

Its important to state that (i) and (ii) are tightly based on the current state of the communication.

Step 4: Sending the response message

Since the most useful transmission interrupt that PE exports is the one generates a interruption after a character is sent to the TX channel (that is, the UART TX register is empty), the idea here was: The handling step above triggers a sequence of chained interruptions calling the sending character function once. When this first character is sent to the TX channel, a interruption is generated, updating the character to be sent and also calling the sending character function again, which will be responsible for generating a interrupt and so on so forth. This chained calls occurs until the whole message is sent.

Limitations

Since a linear buffer is being used, the program cannot handle the case in which a second message is received while the first one is being treated. Although the program is pretty robust to network connection failures and unexpected messages, it was designed to a modest use case: Receive a message > Parse the received message > Assemble the response message > Send the assembled message. This queueing of the multiple messages feature could be implemented using a ring buffer.

Also, both UARTs can write in the same linear buffer at any time. There's no protection and neither a single linear buffer for each one (although the program has a different one for the to-be-send messages). This methodology is simpler, but it decreases the robustness of the program in terms of bad usage. As long as the user respects the protocol ([2]), there will be no problems, but the program shouldn't expect that. In addition, there's no validation of the data received (e.g. Is it a valid IP number? Is it a valid MAC address?).

5 Validation

Since I developed most of the program away from University, I couldn't connect with the local provided network. In order to bypass this, I used a SERIAL-TO-TTL converter acting like the ESP01 module, opening two serial connections in my computer (the first one over OpenSDA provided by the FRDM-KL25Z board).

Then, when I felt the program was ready to be deployed, I did it connecting to the local provided network. I also tested the communication using the *MyMQTT app* on my smartphone, both publishing a message into the "EA076/grupoD3/celular" topic and receiving a message from the "EA076/grupoD3/ESP" topic.

References

[1] ftp://ftp.dca.fee.unicamp.br/pub/docs/ea076/manuais/MQTT_V3.1_Protocol_Specific.pdf.

[2] ftp://ftp.dca.fee.unicamp.br/pub/docs/ea076/complementos/ESP8266_CLIENTE_MQTT.pdf.

[3] https://wiki.gnome.org/action/show/Apps/Moserial?action=show&redirect=moserial.