PROJECT DOCUMENTATION

Objective:

Aim of the project is to build a wireless switch to control the brightness of office or home lights connected over a wireless network. The control is not just limited to lights. It can be used to control almost all kind of Electrical and Electronic devices. And the switch can be an electrical switch, smart phone application or software running on a PC.

Need or Benefits of Wireless Switching:

Wireless switches can lower equipment costs in a variety of ways. For one, the cost of manufacturing and installation is reduced. Not only the expense of wiring is eliminated, but no clips or connectors are required. There are no wire routing problems to solve, no need for pulling wire during installation and fewer restrictions on location and placement of the switch.

Wireless limit switches can also reduce maintenance costs. Equipment wiring is less complex with the elimination of wired switches from the mix, simplifying troubleshooting and reducing maintenance time. Further, going wireless increases system reliability by eliminating the potential for having issues with switch wiring or connectors. Switches also become simpler to replace, with no need to disconnect and re-attach wiring and no risk of incorrect wire attachment.

Introduction:

This project uses ESP8266, a low-cost Wi-Fi chip with full TCP/IP stack and microcontroller capability, integrated on a small embedded development board Node MCU, which is an IoT platform. It is interactively programmable in Lua scripts. It can also be programmed in C using Arduino IDE.

Features of ESP8266:

- 32-bit RISC CPU running at 80 MHz.
- 64 KB of instruction RAM, 96 KB of data RAM.
- External flash memory- 512 KB to 4 MB (up to 16MB is supported).
- IEEE 802.11 b/g/n Wi-Fi.
    - WEP, WPA/WPA2 authentication and Open Network feature.

- 16 GPIO pins.
- SPI, I²C.
- I²S interfaces with DMA (sharing pins with GPIO).
- UART on dedicated pins, plus a transmit-only UART can be enabled on GPIO2.
- One 10-bit ADC.

Networking Part:

MQTT protocol is used for communication between the devices connected on the wireless network. MQTT was invented by Andy Stanford-Clark (IBM) and Arlen Nipper (Arcom, now Cirrus Link) back in 1999. This protocol uses publish and subscribe mechanism. It uses binary format (with shorter headers), requiring less bandwidth and battery. It is a Client Server publish/subscribe messaging transport protocol. In this protocol, there are clients connected to an MQTT server or broker, such that all clients are directly connected to the broker but no two clients are connected together and they have no idea of each other. These clients can either subscribe or publish or both, on a topic or more. The message published by a client on a topic is sent to all the other clients who have subscribed for that topic and are connected to the broker. And this controlled distribution of messages based on their topic interests, is the sole responsibility of the MQTT broker and no client has to do anything else than publish and subscribe. Hence, this protocol is very much useful to machines (clients) having restricted battery usage issue. In this protocol, a good point is that the protocol does not ensure the best service every time, the quality of service (QoS) is defined by user, and hence it doesn't have to waste its power in trying to achieve best service possible, which makes it simple and reliable.

- **Features of MQTT:**
    - MQTT is a Client Server Publish/Subscribe messaging transport protocol.
    - Simple to implement.
    - Provides a Quality of Service Data Delivery.
    - Lightweight and Bandwidth Efficient.
    - Data Agnostic.
    - Continuous Session Awareness.

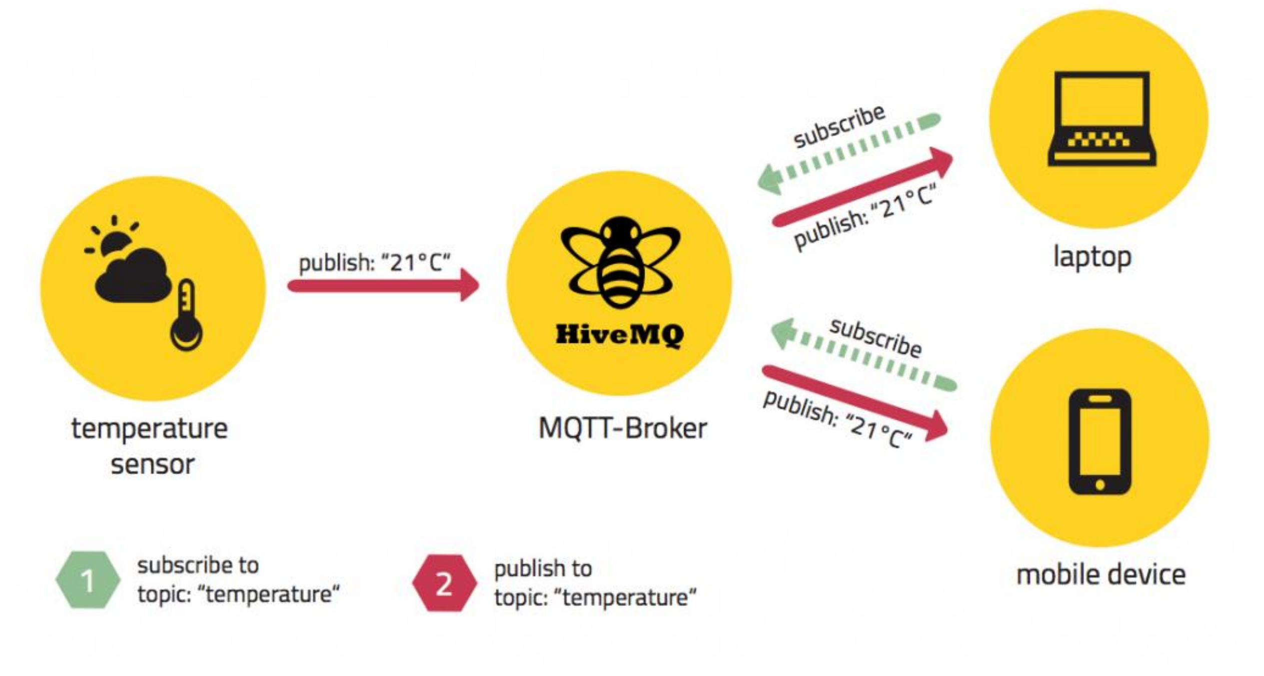
Often MQTT is incorrectly named as message queue protocol, but this is not true. There are no queues as in traditional messaging. However, it is possible to queue message in certain cases.

The protocol was named after a product of MQ Series, developed by IBM which supports MQTT, when it was invented 1999.

So its acronym is just MQ Telemetry Transport.

The Publish/Subscribe Pattern:

The publish/subscribe pattern (pub/sub) is an alternative to the traditional client-server model, where a client communicates directly with an endpoint. However, Pub/Sub decouples a client, who is sending a particular message (called publisher) from another client (or more clients), who is receiving the message (called subscriber). This means that the publisher and subscriber don't know about the existence of one another. There is a third component, called broker, which is known by both the publisher and subscriber, which filters all incoming messages and distributes them accordingly.



The clients are nothing but the devices working as Input/Output or both. Like temperature sensor (Input device to read temperature), LCD display (Output device to display the temperature) or a Smartphone (used to send messages as well as receive readings from temperature sensor).

Main aspect in pub/sub is the decoupling of publisher and receiver, which can be differentiated in more dimensions:

1. Space decoupling: Publisher and subscriber do not need to know each other (by ip address and port for example).

2. Time decoupling: Publisher and subscriber do not need to run at the same time.

3. Synchronization decoupling: Operations on both components are not halted during publish or receiving

In summary publish/subscribe decouples publisher and receiver of a message, through filtering of the messages it is possible that only certain clients receive certain messages. The decoupling has three dimensions: Space, Time and Synchronization.

Message Filtering:

So what's interesting is, how does the broker filter all messages, so each subscriber only gets the messages it is interested in?

Option 1: Subject-based filtering

The filtering is based on a subject or topic, which is part of each message. The receiving client subscribes on the topics it is interested in with the broker and from there on it gets all message based on the subscribed topics.

Topics are in general strings with an hierarchical structure, that allow filtering based on a limited number of expression.

Option 2: Content-based filtering

Content-based filtering is as the name already implies, when the broker filters the message based on a specific content filter-language. Therefore clients subscribe to filter queries of messages they are interested in. A big downside to this is, that the content of the message must be known beforehand and can not be encrypted or changed easily.

Option 3: Type-based filtering

When using object-oriented languages it is a common practice to filter based on the type/class of the message (event).

In this case a subscriber could listen to all messages, which are from type Exception or any subtype of it.

Challenges:

Of course publish/subscribe is not a silver bullet and there are some things to consider, before using it.

The decoupling of publisher and subscriber, which is the key in pub/sub, brings a few challenges with it.

You have to be aware of the structuring of the published data beforehand. In case of subject based filtering, both publisher and subscriber need to know about the right topics to use. Another aspect is the delivery of message and that a publisher can’t assume that somebody is listening to the messages he sends. Therefore it could be the case that a message is not read by any subscriber.

Methodology:

After learning about hardware of ESP8266 and basics of MQTT it was to learn about working on Node MCU board with programming in Arduino IDE. Different aspects of study and implementation are as follows:

Pin Mapping:

When working with Arduino IDE, it is important to consider proper mapping of the pin numbers during coding programs. For example if pin D5 connected to GPIO14, then we have to put pin number 14 and not 5 in the program code. But an easy way to avoid such mapping confusions, is to use #define ledPin D5, which also works fine.

WiFi:

Next step into it was to learn about WiFi library for WiFi access of Esp8266. WiFi.begin(ssid, password), WiFi.status(), WiFi.disconnect(), WiFi.localIP(), and etc. were used in this aspect.

Polling vs Interrupt:

As the project was mostly centered around obtaining different functionality of pushbuttons on the board, it was important to check button status and state if it was a short press, long press, still pressed or released. Initially this reading of button state was done on pooling basis, that is, every time in the main loop function, the program checks the state of the buttons. Since this pooling is never considered an efficient way to do this, the next challenge was to find interrupt method for this. It was found that the interrupt function used for arduino boards worked well with this board too. Functions like attachInterrupt(pin, function, mode), interrupts(), detachInterrupt(), noInterrupts() which work for an arduino boards, were found to work well with this board also. So making interrupt calls became easy.

Detection of Short Press and Long Press of Buttons:

Now the next thing to be done was to identify and differentiate between a short press or a long press. In this step, the challenge was to implement a timer interrupt which could trigger up just after a second of pressing the button and call an ISR (Interrupt Service Routine) to check if any falling edge of the button was encountered. If there was a low state of button before the timer interrupt came, it was considered as a short press or else it was a long press at hold situation until the interrupt for the falling edge of the button was found.

Timer Interrupt:

But it was not easy to find a timer function which could work for nodeMCU in Arduino IDE. Timer0 was found to work fine with it. It could work well as a timer interrupt and hence the problem was resolved.

Testing the Program:

After the programming part was done, it was time to test the program by flashing the Node MCU board by connecting it to laptop via serial COM port. After uploading the program the working of the project was analyzed by connecting client application (running on laptop) to the broker to whom Node MCU board (another client) was to be connected. Program is such that when started, Node MCU board will connect to WiFi and then connect to the broker. As soon as the button is pressed on the board an interrupt signal is generated which in turn starts a timer. The timer generates an interrupt after a fixed interval of time that can be set as required time duration between two successive publishes. After the button is pressed and the first timer interrupt comes it is again checked. In case it is released a message e.g. “Button is short pressed” is published to a predefined topic e.g. “ButtonStatus” otherwise if it is still pressed another message e.g. “Button is on hold” is published and the timer is reloaded with the same interval. If the button is released after generation of one or more than one timer interrupt signal a message e.g. “Button is released” is published and the timer interrupt is detached. From the PC client the same topic (i.e. “ButtonStatus) was subscribed to test whether the messages were being published correctly.After few improvements in the program the desired result was obtained.

Setting up our own Broker:

After testing the program by connecting the board to CDAC broker, it was time to test if it runs on our own broker or not. So an MQTT broker (by Hivemq) was setup in the laptops. And by changing the connection credentials in the program we were able to connect the ESP8266 with the local broker set up on laptops. After analyzing the messages published it was observed that the code works fine.

Modularization of the program:

Though the code worked fine but it was a bit slow and the code became complex and lengthy that it was a difficult task to debug the code. So we decided to make the code smaller and simpler by creating multiple source code files in C and by defining our own project header file containing the functions required in main program.