Risultato di valorizzazione applicativa:

Progettazione e realizzazione di un prototipo di sensore wireless per il monitoraggio di carichi elettrici in ambiente Smart Building

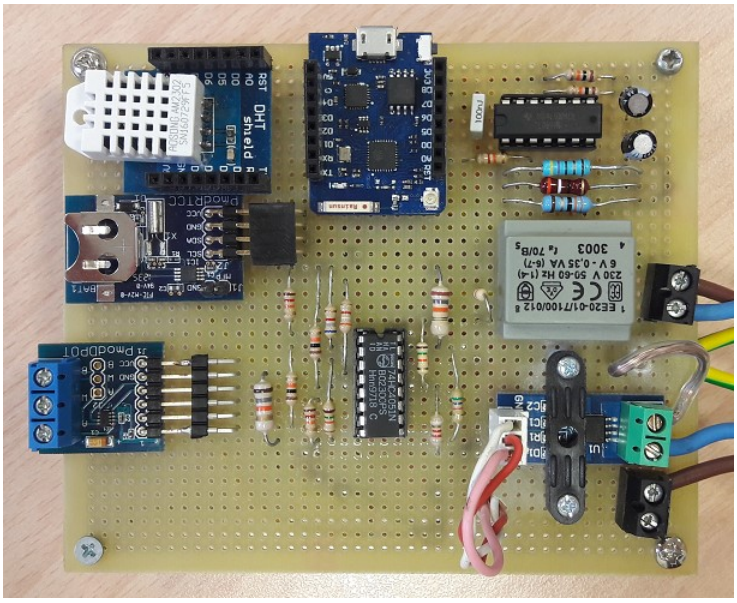
(Rapporto tecnico CNR-ISSIA n. 425 anno 2018)

Autori:
M. Luna[1], G. La Tona[1], S. G. Scordato[1], A. Sauro[1], M. C. Di Piazza[1], M. Pucci[1], C. Vetro[2], R. La Grassa[2]

[1] CNR-ISSIA sede secondaria di Palermo
Istituto di Studi sui Sistemi Intelligenti per l'Automazione
Via Ugo La Malfa, 153 - 90146 Palermo (Italy)
Tel. +39 091 6809111

[2] Università degli Studi di Palermo
Dipartimento di Matematica e Informatica
Via Archirafi, 34 - 90123 Palermo (Italy)
Tel. +39 091 23891111

Data di creazione: 3 aprile 2018
Prima distribuzione al pubblico e deposito legale: 3 aprile 2018



INDICE

INTRODUZIONE

La gestione intelligente dell'energia (Smart Energy) all'interno degli edifici è di vitale importanza per l'efficientamento degli stessi e lo sviluppo sostenibile della società. Un'indagine effettuata negli Stati Uniti ha mostrato che gli edifici sono responsabili di circa il 38% delle emissioni totali di CO_2, il 71% del consumo elettrico totale, il 39% dell'utilizzo complessivo di energia, il 12% del consumo di acqua e il 40% dei rifiuti non industriali [1]. In generale, gli edifici sono sistemi complessi e di natura eterogenea; il loro consumo energetico può, quindi, variare in dipendenza da diversi fattori interni ed esterni ed è estremamente difficile da ottimizzare senza ricorrere all'utilizzo di sistemi automatici. Inoltre, gli edifici convenzionali non sono stati progettati con particolare attenzione agli aspetti energetici; pertanto, oltre alla definizione di nuovi criteri di progetto e linee guida per la costruzione di nuovi edifici energeticamente efficienti, sono necessarie anche tecnologie e dispositivi che consentano di intervenire sugli edifici esistenti al fine di ottimizzarne le prestazioni energetiche.

Con particolare riferimento all'utilizzo dell'energia elettrica, lo sviluppo delle fonti rinnovabili e la crescente attenzione verso l'efficientamento energetico hanno consentito di sviluppare un nuovo sistema di distribuzione dell'energia elettrica agli utenti non più centralizzato, ma basato sulla generazione distribuita, permettendo all'occorrenza anche il funzionamento in autonomia di sottosistemi della rete comprendenti generazione locale, carichi e sistemi di accumulo, detti microgrid [2] o nanogrid [3] a seconda del livello di potenza. Le microgrid/nanogrid presentano innumerevoli vantaggi, tra cui:

- maggiore efficienza e riduzione dei costi di trasporto dell'energia grazie al fatto che il consumatore può coincidere con il produttore dell'energia (prosumer = producer/consumer);
- ottimizzazione dell'uso delle risorse energetiche esistenti e minore impatto ambientale, beneficiando dell'uso sempre più crescente delle fonti rinnovabili;
- miglioramento della Power Quality, ovvero della qualità dell'energia elettrica dal punto di vista del fattore di potenza, della distorsione armonica, del flicker, etc.

La rete elettrica nazionale, le sorgenti da fonte rinnovabile, i carichi e gli eventuali sistemi di accumulo devono concorrere allo scambio intelligente dell'energia con un funzionamento coordinato, al fine di perseguire obiettivi di efficienza, economia di gestione, etc. A tal fine risultano necessari un'idonea infrastruttura di comunicazione e una logica di controllo (centralizzata, distribuita o mista). Questa caratteristica rende le microgrid intelligenti e, pertanto, si parla di Smart Microgrid [4]. Lo stesso concetto può essere applicato a scala ridotta all'interno del singolo edificio o della singola abitazione, parlando in questo caso di Smart Nanogrid [5].

Spesso non è sufficiente effettuare soltanto effettuare il monitoraggio energetico in un edificio (in particolare di generazione e consumo di energia elettrica) attraverso sensori e reti di comunicazione e la successiva analisi di dati offline per individuare interventi migliorativi. In uno Smart Building [6] sono disponibili anche sistemi automatici capaci di elaborare tali dati automaticamente ed attuare opportune strategie di gestione ottimizzata dei flussi energetici, consentendo di ottenere prestazioni energetiche ottimali anche a fronte di incertezze sull'energia prodotta da fonti rinnovabili e sulle abitudini di consumo dell'utente finale.

Al fine di affrontare le suddette problematiche, è stato proposto in letteratura l'utilizzo di sistemi EMS [7]. Essi stabiliscono in tempo reale i flussi di energia tra le sorgenti, i carichi e i sistemi di accumulo al fine di perseguire uno o più obiettivi specificati dall'utente. Esempi di obiettivi possibili sono l'incremento dell'efficienza, la riduzione delle emissioni di CO_2, la minimizzazione del costo di esercizio, la massimizzazione dell'autonomia in caso di disconnessione

dalla rete elettrica nazionale, etc. [8]. Tali sistemi vanno oltre le funzionalità offerte dalla domotica e spesso, piuttosto che essere basati su regole statiche, risolvono problemi di ottimizzazione in tempo reale, riuscendo a correggere gli eventuali errori di previsione ed ottenendo risultati sensibilmente migliori.

Gli input per i sistemi EMS sono costituiti principalmente dai dati provenienti da sensori. A tal fine, nell'ottica di semplificare il cablaggio e consentire l'adeguamento più rapido degli edifici esistenti, risulta conveniente utilizzare sensori wireless, ovvero sensori che trasmettono i dati misurati via etere, ad esempio sfruttando la rete WiFi presente all'interno dell'edificio.

Oggetto del presente lavoro è stata la progettazione e realizzazione di un prototipo di sensore wireless a basso costo per il monitoraggio di carichi elettrici in ambiente Smart Building, capace di inviare dati ad un sistema remoto (ad esempio un EMS) mediante una comune connessione WiFi.

L'attività si inquadra nell'ambito di una collaborazione scientifica tra l'Istituto di Studi sui Sistemi Intelligenti per l'Automazione (ISSIA) del Consiglio Nazionale delle Ricerche (CNR) e il Dipartimento di Matematica e Informatica (DMI) dell'Università degli Studi di Palermo (UNIPA).

Il prototipo è stato realizzato interfacciando opportunamente alcuni dispositivi hardware commerciali, aggiungendo gli opportuni circuiti per il condizionamento dei segnali da acquisire e scrivendo il codice per l'implementazione del firmware del sensore wireless (per l'invio dei dati) e del client remoto (per la ricezione dei dati).

Il presente documento descrive le funzionalità del sistema realizzato ed è suddiviso nei seguenti capitoli. L'introduzione inquadra la tematica affrontata e descrive brevemente l'obiettivo del progetto. Il primo capitolo riporta le specifiche del dispositivo in termini di requisiti del sensore e del client remoto. L'architettura del sistema e l'infrastruttura di comunicazione sono descritte nel secondo capitolo. Il terzo capitolo presenta la descrizione dei dispositivi hardware commerciali utilizzati e dei circuiti appositamente realizzati. Il quarto capitolo è dedicato, invece, alla descrizione del software e, in particolare, del codice scritto per implementare il nodo sensore e il client remoto. Il capitolo successivo illustra il test del sistema. Infine, vengono presentate le conclusioni su quanto realizzato e la possibilità di sviluppi futuri.

1 – REQUISITI DI PROGETTO

Come anticipato nell'introduzione, l'obiettivo che ci si prefigge è la progettazione e realizzazione di un prototipo di sensore wireless basato su microcontrollore che consenta l'acquisizione di dati e il loro invio ad un sistema remoto. Specificamente, ciascun sensore dovrà poter essere collegato ad un qualsiasi carico elettrico per acquisirne i principali parametri elettrici (tensione, corrente, potenza, etc.) e il sistema remoto centralizzato dovrà poter ricevere e memorizzare i messaggi inviati dai sensori wireless, nonché consentire la riprogrammazione da remoto degli stessi qualora cambiasse qualche parametro di configurazione (ad es. l'SSID e la password della rete WiFi a cui connettersi).

Le specifiche del sistema da realizzare sono state formalizzate come segue.

Specifiche del sensore wireless:

- possibilità di misurare le principali grandezze elettriche mediante l'utilizzo di un convertitore analogico/digitale (ADC, Analog to Digital Converter);
- predisposizione alla misura di ulteriori grandezze fisiche (ad es. temperatura ambiente) mediante ADC o interrogazione di sensori basati su protocollo 1-Wire, I2C e/o SPI;
- mantenimento della data/ora corrente ed associazione del timestamp ai dati;
- invio dei dati su rete WiFi;
- alimentazione a batteria;
- salvataggio dei parametri di configurazione su EEPROM;
- possibilità di riconfigurazione del sensore e di scrittura della EEPROM da remoto;
- basso costo e ingombro complessivo.

Specifiche del sistema remoto:

- capacità di gestire almeno 254 sensori wireless;
- memorizzazione su database dei dati ricevuti;
- possibilità di riprogrammazione da remoto dei parametri di configurazione di uno o più sensori wireless;
- basso costo e ingombro complessivo.

2 – ARCHITETTURA DEL SISTEMA

2.1 Componenti del sistema

Il sistema realizzato prevede la cooperazione di diversi componenti distribuiti che comunicano attraverso la rete. Il sistema è stato realizzato in maniera tale da permettere la semplicità di configurazione e di modifica della configurazione, cioè l'aggiunta/rimozione di dispositivi con facilità. All'utente che decide di installare il sistema nella propria abitazione non è richiesto altro che di collegare i carichi elettrici ai nodi sensore. Questi ultimi comunicano senza fili attraverso la rete Wi-Fi già presente in casa. Il sistema realizzato è costituito dai seguenti componenti:

- gestore della configurazione;
- gestore della persistenza delle misure;
- nodi sensore.

La Fig. 1 schematizza i componenti del sistema e le relative connessioni. I carichi elettrici (che siano, ad esempio, semplici lampade o elettrodomestici) sono connessi ai nodi sensore, basati su schede embedded opportunamente scelte, che ne monitorano il consumo elettrico. I nodi di gestione (configurazione e persistenza delle misure) vengono eseguiti a bordo di una apposita piattaforma embedded insieme al framework di supporto alla comunicazione (descritto nella sezione successiva). Gli utenti possono conoscere lo stato del sistema e le informazioni di consumo attraverso i browser in esecuzione sui loro dispositivi, ad esempio PC fissi o tablet. Infine, il sistema espone un'interfaccia verso l'esterno che sistemi di più alto livello (ad esempio residenti nel cloud) possono sfruttare per ottenere i dati relativi alle misure e alle loro serie storiche. Ad esempio, sistemi di data mining possono analizzare le serie storiche per comprendere le abitudini dell'utente, oppure sistemi di forecast le possono utilizzare per prevederne gli andamenti futuri.

Con riferimento ai tre componenti del sistema, è possibile riassumerne le funzioni come segue:

- i nodi sensore periodicamente effettuano le misurazioni e pubblicano i relativi dati che il gestore della persistenza archivia su un database perché sia possibile consultarle, analizzarne l'andamento ed effettuare ricerche su queste;
- il nodo gestore della persistenza delle misure riceve le misure pubblicate dai nodi sensore e le salva su un database;
- il nodo gestore della configurazione imposta i parametri dei nodi sensore inviando a questi un messaggio di comando che specifica i valori da impostare; i parametri attualmente previsti sono la data, l'ora e le credenziali per l'accesso alla rete Wi-Fi; l'utente finale, interfacciandosi con il nodo gestore della configurazione, può richiedere che la configurazione dei sensori venga aggiornata.

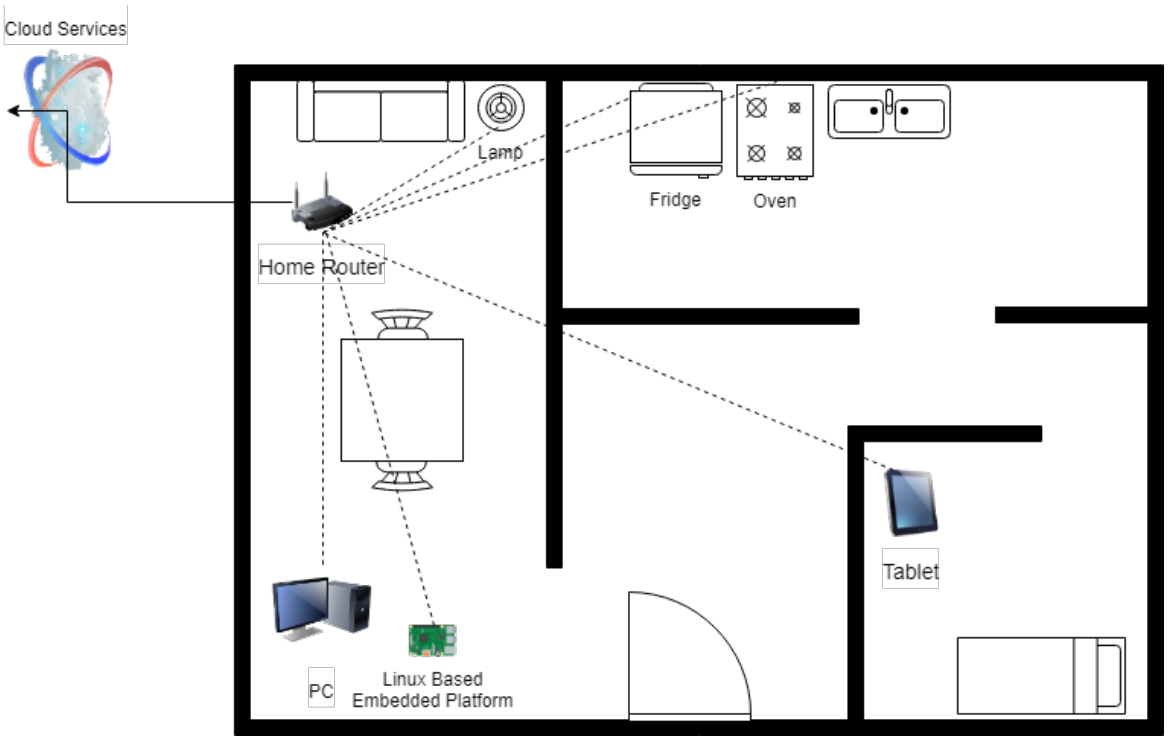


Fig. 1. Dispositivi del sistema e relative connessioni. A titolo d'esempio vengono riportati diversi elettrodomestici e carichi elettrici ai quali sono connessi i sensori sviluppati che comunicano attraverso la rete con gli altri componenti del sistema.

2.2 Infrastruttura di comunicazione

Considerata la natura asincrona e distribuita della comunicazione tra i nodi del sistema si è scelto di adottare un paradigma di comunicazione basato sullo scambio di messaggi. Il protocollo scelto per lo scambio di messaggi tra i componenti del sistema è Message Queue Telemetry Transportation (MQTT) [9]. Si tratta di un protocollo di messaggistica basato sul paradigma publish/subscribe (pubblicazione/abbonamento), cioè un protocollo che prevede che ci siano dei canali di comunicazione che definiscono l'argomento dei messaggi (topic) e che gli attori che intendono ricevere messaggi appartenenti a un topic debbano abbonarsi. Chi desidera inviare un messaggio deve sempre, contestualmente, dichiarare il topic su cui pubblicare. Questo paradigma richiede la presenza di un nodo intermediario (chiamato broker) che smisti i messaggi pubblicati a tutti i nodi abbonati ai relativi topic. Il protocollo MQTT sfrutta il protocollo TCP come protocollo di trasmissione.

MQTT nasce appositamente per applicazioni che necessitino di avere un footprint di codice limitato e comunichino anche quando la larghezza di banda è limitata. Pertanto, è comunemente utilizzato in prodotti Internet of Things (IoT) o più in generale per la comunicazione Machine To Machine (M2M) [10]. Per l'implementazione del sistema di messaggistica si è scelto di utilizzare il broker open source Mosquitto [11] che sarà eseguito su una piattaforma embedded basata su Linux scelta nel capitolo 3.

La Fig. 2 mostra le interazioni tra gli elementi del sistema attraverso lo scambio di messaggi. In particolare, vengono riportati alcuni esempi di richieste di pubblicazione di messaggi e di abbonamento ai topic che vengono inviate al broker. Il broker è responsabile dello smistamento dei messaggi pubblicati per dato topic ai relativi abbonati. Ad esempio, i messaggi contenenti le misure rilevate dai nodi sensori vengono inviati al nodo gestore della persistenza, che si abbona al relativo topic.

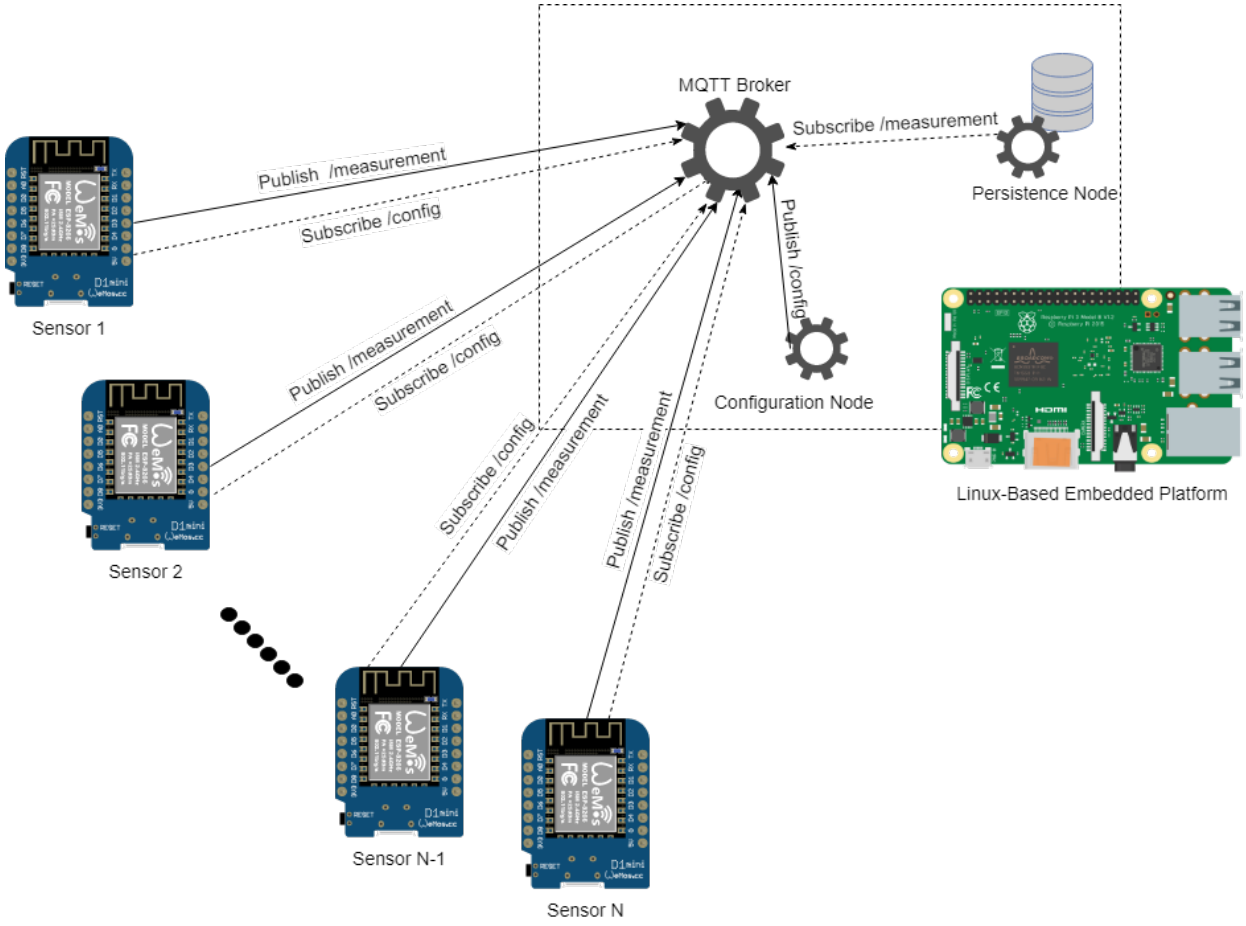


Fig. 2. Nodi del sistema e relativa infrastruttura di comunicazione basata su MQTT. Sono riportati dei topic esemplificativi per la pubblicazione delle misure e dei dati di configurazione. Il broker MQTT, il nodo di configurazione e il nodo di persistenza sono tutti processi che vengono eseguiti sulla piattaforma embedded basata su Linux.

Il protocollo MQTT prevede anche la definizione della Quality of Service (QoS) per i messaggi pubblicati e per le richieste di abbonamento a un topic. QoS è un accordo tra mittente e destinatario riguardo le garanzie di consegna di un messaggio. MQTT prevede tre livelli di QoS:

- Livello 0 – 'al massimo una volta': il messaggio viene inviato e non c'è alcun tipo di acknowledgement da parte del ricevente;
- Livello 1 – 'almeno una volta': chi invia un messaggio attende un acknowledgement e, nel caso non lo riceva, invia nuovamente il messaggio; viene garantito che il messaggio venga ricevuto almeno una volta, ma può anche essere ricevuto più volte;
- Livello 2 – 'esattamente una volta': mittente e ricevente si scambiano una serie di acknowledgement per assicurarsi che il messaggio venga ricevuto esattamente una volta e non ci siano quindi duplicati.

Al crescere del livello di QoS cresce la complessità e quindi il carico computazionale e la quantità di dati trasmessi.

Il livello di QoS con cui un messaggio viene consegnato a un client abbonato dipende dal livello con cui il messaggio è stato pubblicato e dal livello specificato dal client in fase di abbonamento. Il QoS specificato dal mittente viene utilizzato per la comunicazione tra questo e il broker; successivamente il broker inoltrerà il messaggio al client abbonato con il livello che questo

ha richiesto in precedenza. Il livello scelto per il sistema in esame è il livello 0 al fine di minimizzare il carico sulla rete e sui nodi sensore.

Il sistema prevede diversi topic per la raccolta dei dati sensoriali, per la configurazione e per la presentazione dei nodi sensore. L'elenco completo è riportato in Tabella 1.

Tab. 1. Topic utilizzati dal sistema.

Topic	Descrizione
/requestHello	Richieste a tutti i nodi sensore presenti nella rete di presentarsi
/config	Dati di configurazione dei nodi sensore
/datetime	Dati di configurazione dell'ora e della data
/answers	Feedback alle richieste di configurazione dei nodi sensore
/hello	Presentazioni dei nodi sensore
/data	Dati delle misure dei sensori

3 – PROGETTAZIONE DELL'HARDWARE

3.1 Scelta delle piattaforme embedded

Sulla base delle specifiche di progetto, sono state scelte le piattaforme embedded di seguito descritte.

Per quanto concerne il sensore wireless è stata scelta la board Wemos D1 mini pro [12]. Tale scheda, mostrata in Fig. 3, è basata sul System on Chip (SoC) ESP8266 della Espressif Systems, comprendente un chip Wi-Fi IEEE 802.11 b/g/n a basso costo con supporto completo a TCP/IP e un microcontrollore RISC L106 a 32 bit, basato su Tensilica Xtensa Diamond Standard 106Micro.

Tale piattaforma risulta adatta all'applicazione in esame, in quanto rappresenta un giusto compromesso tra costo (circa $ 5.00), prestazioni (circa 20 MIPS) ed ingombro (dimensioni della board: 34.2 mm x 25.6 mm; peso: 2.5 g). Inoltre, sfruttando apposite librerie, essa risulta facilmente programmabile in linguaggio C mediante l'IDE di Arduino [13].

Ulteriori specifiche tecniche della board Wemos D1 mini pro sono le seguenti:

- clock a 80/160 MHz
- circa 50 kB di RAM utilizzabili per i dati
- memoria Flash esterna da 16 MB
- 11 pin GPIO con supporto di interrupt, SPI, I2C e 1-Wire
- ADC integrato a 10 bit con 1 ingresso analogico (max 3.2 V)
- antenna ceramica integrata e connettore per antenna esterna.

Fig. 3. La board Wemos D1 mini pro

Per quanto riguarda l'esecuzione del broker MQTT (Mosquitto v. 1.4.14 per Linux), invece, è stato scelto di utilizzare una piattaforma embedded basata su Linux. In tal modo è stato possibile utilizzare Python per scrivere in modo semplice e veloce il codice necessario ad implementare i nodi di gestione (configurazione e persistenza delle misure) e ad interfacciarli con il broker utilizzando la libreria client per MQTT Eclipse Paho [14]. Specificamente, è stata utilizzata una scheda embedded Olimex A10 che è basata su un microprocessore Allwinner A10 Cortex-A8 con clock a 1 GHz [15]. Anche questa piattaforma, mostrata, in Fig. 4, rappresenta un giusto compromesso tra costo (circa € 30.00), prestazioni (circa 2000 MIPS) ed ingombro (dimensioni della board: 84 mm x 60 mm). Inoltre, rappresenta un ambiente di esecuzione molto potente e versatile, poiché basato sulla distribuzione Debian di Linux (oppure, a scelta, su Android). Ulteriori specifiche tecniche della board Olimex A10 sono le seguenti:

- GPU Mali 400
- 512 MB di RAM DDR3

- EEPROM da 2 kB
- Connettore per schede MicroSD
- Connettori SATA e HDMI
- 2 porte USB High-speed host
- 1 porta USB-OTG
- Connettore DEBUG-UART per il debug su console
- Ethernet a 100 MBit
- Connettore per batteria LiPo
- 160 porte GPIO
- 3 pulsanti con funzionalità per ANDROID
- Pulsante di reset

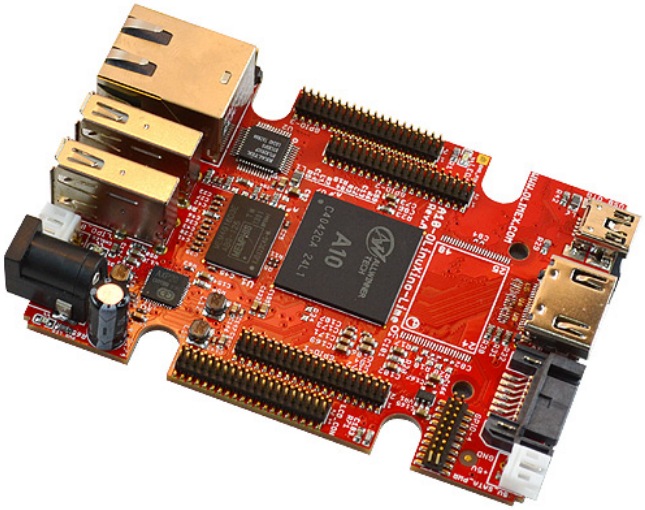


Fig. 4. La board Olimex A10

3.2 Scelta dei componenti elettronici

A partire dalle specifiche di progetto, sono state effettuate le seguenti considerazioni. L'ADC del microcontrollore ESP8266 presenta un solo canale analogico di ingresso. Di contro, nell'applicazione in esame occorre acquisire più di un segnale analogico (tensione, corrente ed eventualmente altre grandezze analogiche). Pertanto, risulta necessario l'utilizzo di un multiplexer analogico per aumentare il numero di canali di ingresso.

Inoltre, il microcontrollore ESP8266 non è dotato di un clock RTCC (Real-Time Clock Calendar), né di una EEPROM di piccola dimensione, indipendente dalla memoria flash che ospita il codice. Pertanto, risulta necessario l'utilizzo di un chip RTCC esterno per poter mantenere aggiornata la data e l'ora corrente da associare al dato misurato, e di una EEPROM esterna, dove salvare i parametri di configurazione.

Infine, per poter alimentare il sensore wireless a batteria, oltre che mediante connessione alla porta USB, è necessario utilizzare un regolatore di carica adatto alla gestione di una batteria ai polimeri di Litio da 3,7 V ed un convertitore DC/DC di tipo boost che innalzi tale tensione fino al livello di 5 V necessario ad alimentare la board Wemos D1 mini pro.

A seguito di ricerche sui cataloghi dei produttori, sono stati esaminati diversi datasheet e alla fine sono stati scelti i seguenti componenti:

- Multiplexer analogico (8:1) 74HC4051 (http://www.ti.com/lit/ds/symlink/cd74hc4051-ep.pdf)
- Shield Digilent PMOD con RTCC ed EEPROM (http://store.digilentinc.com/pmod-rtcc-real-time-clock-calendar/)
- Shield Wemos Battery con regolatore di carica e convertitore boost TP5410 (https://www.homotix.it/vendita/wemos/wemos-battery-shield)

Per quanto concerne i sensori, sono state fatte le seguenti scelte. Per la misura di corrente, piuttosto che il classico sensore LEM, che risulta ingombrante e costoso e necessita di un'alimentazione a ±15V, è possibile utilizzare il sensore ad effetto Hall Allegro ACS712.

In particolare, trattandosi di una realizzazione prototipale, per semplicità è stata utilizzata una breakout board basata su un circuito integrato ACS712ELCTR-05B-T, con portata di 5 A. I morsetti di ingresso di tale board saranno collegati in serie al circuito elettrico di cui misurare la corrente.

All'occorrenza, risulta eventualmente possibile utilizzare direttamente i chip ACS712ELCTR-20B-T e ACS712ELCTR-30B-T, con portata rispettivamente pari a 20 e 30 A.

Per quanto riguarda la misura di tensione, sempre al fine di ridurre il costo e l'ingombro e di evitare la necessità di fornire alimentazione a tensioni diverse da 5 V e 3.3 V, è stato deciso di utilizzare un semplice trasformatore per Printed Circuit Board (PCB) da 230V/6V di bassa potenza, con il secondario chiuso su un partitore resistivo. I morsetti di ingresso di tale trasformatore saranno collegati in parallelo al circuito elettrico di cui misurare la tensione.

In definitiva, i sensori di tensione e corrente utilizzati sono i seguenti:

- Trasformatore Block 230V/6V 0.35 VA (http://it.rs-online.com/web/p/trasformatori-per-pcb/7320525/)
- Sensore di corrente ACS712 (https://www.futurashop.it/index.php?_route_=sensore-corrente-5a-acs712)

Infine, per quanto concerne la misura di ulteriori grandezze fisiche mediante interrogazione di sensori basati su protocollo 1-Wire, I2C e/o SPI, a titolo di esempio è stata effettuata la rilevazione della temperatura ambiente mediante la lettura tramite protocollo 1-Wire di un sensore DHT22 a bordo della seguente breakout board:

- Shield Wemos DHT PRO con sensore di temperatura/umidità DHT22 (https://www.homotix.it/vendita/wemos/wemos-dht-pro-sheld)

La corretta comunicazione mediante protocollo I2C è stata verificata nel momento in cui è stato interfacciato il microcontrollore ESP8266 con il chip RTCC prima descritto, basato proprio su I2C. Risulta, pertanto, possibile il collegamento diretto di ulteriori periferiche al bus I2C e la loro interrogazione a partire dalla conoscenza dell'indirizzo e del registro.

Come esempio di periferica SPI, invece, è stato utilizzato un potenziometro digitale AD5160 presente a bordo della seguente breakout board:

- Shield Digilent PMOD DPOT con potenziometro digitale AD5160 (http://store.digilentinc.com/pmod-dpot-digital-potentiometer/)

Anche se non si tratta di un sensore, tale componente è stato scelto perché disponibile in laboratorio ed ha consentito di testare la corretta comunicazione tra il microcontrollore ESP8266 ed una generica periferica sul bus SPI. Va osservato che, a differenza del bus I2C, nel bus SPI l'indirizzamento della periferica avviene in hardware mediante il segnale \CS. Non avendo a disposizione un numero sufficiente di porte GPIO sul microcontrollore, risulta, pertanto, possibile il collegamento di una sola periferica SPI, da interrogare a partire dalla conoscenza del registro che contiene il dato desiderato.

3.3 Circuiti di condizionamento dei segnali

I sensori di tensione e corrente scelti forniscono in uscita segnali non compatibili con il range di ingresso dell'ADC. Pertanto, risulta necessario l'utilizzo di opportuni circuiti di condizionamento dei segnali. Il relativo schema elettrico è riportato in Fig. 5.

In particolare, il sensore di corrente fornisce un'uscita compresa tra zero e 5 V. Conseguentemente, è necessario attenuarla di un fattore 1.5625 per riportarla al range 0-3.2 V, così da renderla compatibile con l'ingresso dell'ADC. A tal fine è possibile utilizzare un partitore di tensione, seguito da un amplificatore operazionale in configurazione da inseguitore di tensione, così da disaccoppiare lo stadio di ingresso dell'ADC dal partitore. Le resistenze del partitore possono essere calcolate come segue. Oltre ad imporre il rapporto di partizione, si sceglie di far circolare sul partitore una corrente non superiore a circa un decimo della corrente massima di uscita del sensore ACS712 (3 mA); conseguentemente si ha:

$$\begin{cases} \dfrac{R_2}{R_1+R_2} = \dfrac{1}{1.5625} \\ \dfrac{5}{R_1+R_2} \le \dfrac{3\cdot 10^{-3}}{10} \end{cases}$$

Pertanto, è possibile scegliere ad esempio $R_2 = 16k\Omega$ e $R_1 = 9k\Omega$. Le potenze dissipate sulle resistenze saranno rispettivamente 0.64 mW e 0.20 mW.

L'amplificatore differenziale dovrà essere alimentato a 5 V ed avere elevata impedenza di ingresso; è stato scelto il circuito integrato TL084 comprendente quattro amplificatori operazionali con ingresso a JFET (http://www.ti.com/lit/ds/symlink/tl084.pdf).

Per quanto riguarda il trasformatore utilizzato come sensore di tensione, esso viene fatto lavorare quasi alla potenza nominale, così da avere bassa distorsione sulla tensione al secondario. Considerato che a carico il valore efficace della tensione sinusoidale al secondario è pari a circa 6.3 V, il valore di picco sarà pari a 8.9 V e risulterà necessaria un'attenuazione; inoltre, considerato che il segnale misurato e scalato presenta alternativamente segno positivo e negativo, risulta necessario aggiungere un offset per far sì che la tensione in ingresso all'ADC si mantenga nel range 0-3.2 V. A tal fine, è possibile effettuare un accoppiamento capacitivo con un partitore di tensione, come mostrato in Fig. 5. Le resistenze R_2, R_4, R_5 e R_6 possono essere calcolate come segue.

In assenza di segnale, la tensione di ingresso dell'ADC deve essere pari a 1.6 V. Pertanto, deve essere:

$$\frac{R_6}{R_5+R_6} = \frac{1.6}{5}$$

La corrente assorbita dal circuito di ingresso dell'ADC (che presenta impedenza di ingresso pari a circa 320 kΩ) deve essere piccola rispetto a quella che circola nel partitore. A tal fine, è sufficiente che sia:

$$R_6 \le \frac{320k\Omega}{50}$$

Si sceglie pertanto $R_6 = 5.6k\Omega$ e, conseguentemente, $R_5 = 11.9k\Omega$. Le potenze dissipate sulle resistenze saranno rispettivamente 0.457 mW e 0.971 mW.

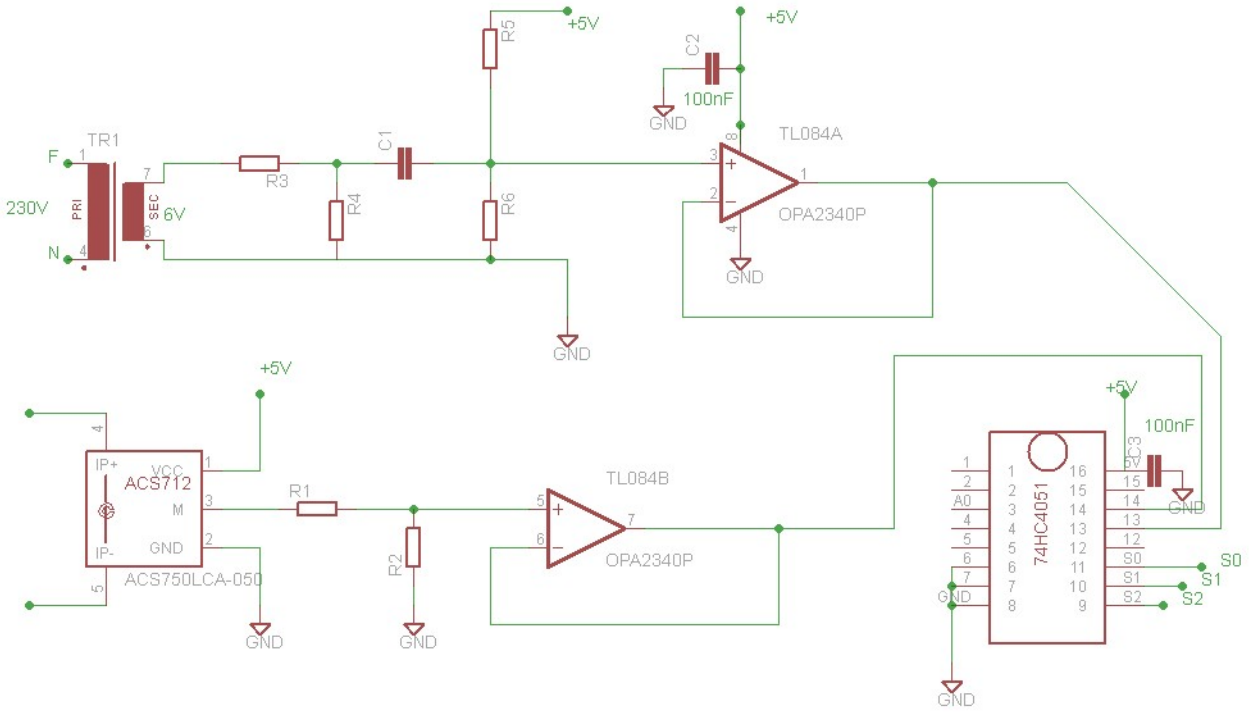


Fig. 5. Schema elettrico dei circuiti di condizionamento dei segnali

L'ampiezza del segnale sinusoidale deve essere portata ad un livello massimo pari a 1.4 V cosicché, una volta sommata all'offset a 1.6 V, generi un segnale compreso tra 0.2 V e 3.0 V (va considerato un margine per tenere conto di eventuali disturbi sulla tensione di rete). A tal fine deve essere:

$$\frac{R_4}{R_3 + R_4} = \frac{1.4}{8.9}$$

Oltre che sul partitore, la corrente al secondario si richiude, attraverso il condensatore, sul parallelo di R_5 e R_6, ovvero su $R_p = \frac{R_5 R_6}{R_5 + R_6} = 3.808 k\Omega$. Per non caricare il partitore costituito da R_3 e R_4, la resistenza R_p deve essere almeno 50 volte maggiore rispetto alla R_4, ovvero $R_4 \leq 76\Omega$.

Inoltre, imponendo che il trasformatore non lavori a potenza superiore a quella nominale, si ottiene:

$$\frac{6.3^2}{R_3 + R_4} \leq 0.35$$

Quest'ultima equazione, combinata con l'equazione che esprime il rapporto di partizione, impone che sia:

$$R_4 > \frac{1.4}{8.9}\frac{6.3^2}{0.35}$$

ovvero $R_4 > 17.8\Omega$.

Se si sceglie $R_4 = 28\Omega$, si ottiene conseguentemente $R_3 = 150\Omega$. Le potenze dissipate sulle resistenze saranno rispettivamente 0.035 W e 0.187 W. Tutte le resistenze dello schema di Fig. 5 dovranno avere tolleranza dell'1% per garantire il risultato desiderato.

Infine, per quanto concerne il condensatore C di accoppiamento, esso realizza insieme alla resistenza R_p un filtro passa alto. Imponendo che la frequenza di taglio del filtro sia una decade

prima della frequenza di rete, ovvero 5 Hz, si ottiene:

$$C = \frac{1}{2\pi f_T R_p} = 8.36\mu F \rightarrow 10\mu F$$

Va osservato, inoltre, che i due amplificatori operazionali non utilizzati del circuito integrato TL084 potranno eventualmente essere sfruttati per il condizionamento di altri due segnali analogici da acquisire. In caso contrario, l'integrato potrà essere sostituito da un TL082, che contiene soltanto due amplificatori operazionali e presenta un ingombro dimezzato rispetto al TL084.

3.4 Realizzazione del prototipo della board del sensore wireless

Trattandosi di un prototipo, il sensore wireless è stato realizzato su una comune basetta millefori di dimensioni 12 mm x 10 mm, collegando insieme la board principale (Wemos D1 mini pro) e i componenti scelti e descritti precedentemente. La realizzazione finale è mostrata in Fig. 6.

A valle della verifica sperimentale, sarà possibile disegnare lo schema elettrico della board mediante appositi CAD elettronici (ad es. Eagle) e fornirlo ad un'azienda del settore per far realizzare una PCB in modo professionale.

Va osservato che, nell'ipotesi di utilizzare direttamente i circuiti integrati anziché le breakout board e considerando l'utilizzo di componenti a montaggio superficiale (SMD, Surface Mounting Device) anziché i classici componenti PTH (Pin Through Hole), le dimensioni del sensore wireless possono essere ridotte sensibilmente. Inoltre, qualora si desiderasse effettuare soltanto il monitoraggio di tensione e corrente, i componenti al di fuori dell'area delimitata in rosso in Fig. 6 potranno essere eliminati.

Se si desidera alimentare il sensore wireless mediante una batteria al Litio, è possibile collegare quest'ultima alla scheda Wemos D1 mini pro mediante lo shield Wemos Battery, mostrato in Fig. 7. Tale shield andrà a sua volta connesso alla Wemos D1 mini pro mediante le due file parallele di contatti.

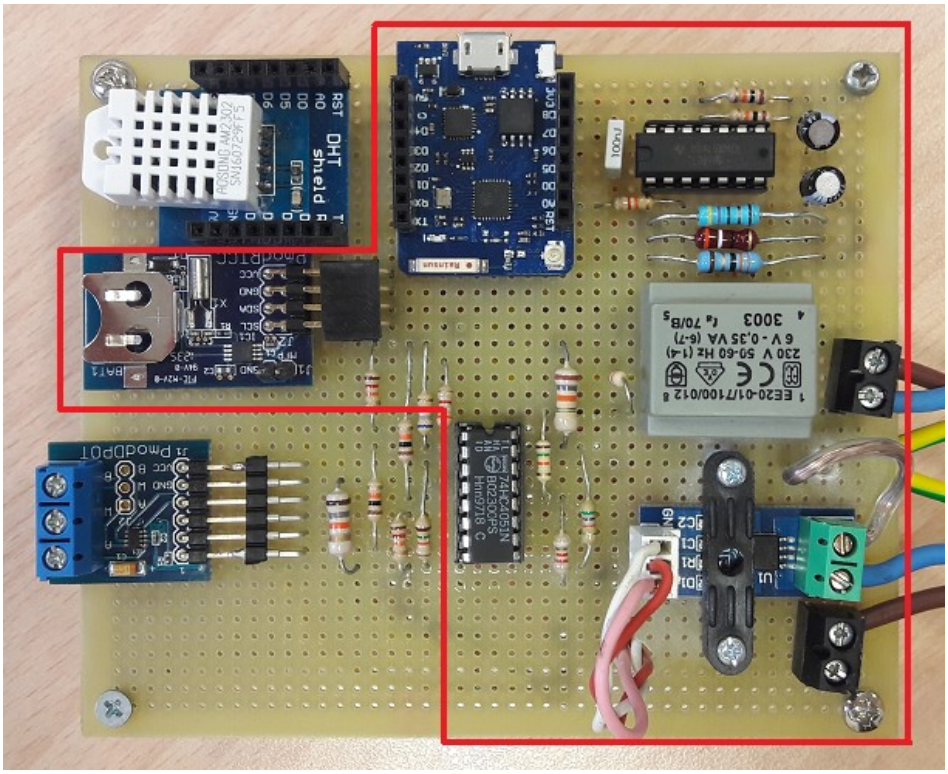


Fig. 6. Prototipo della board del sensore wireless

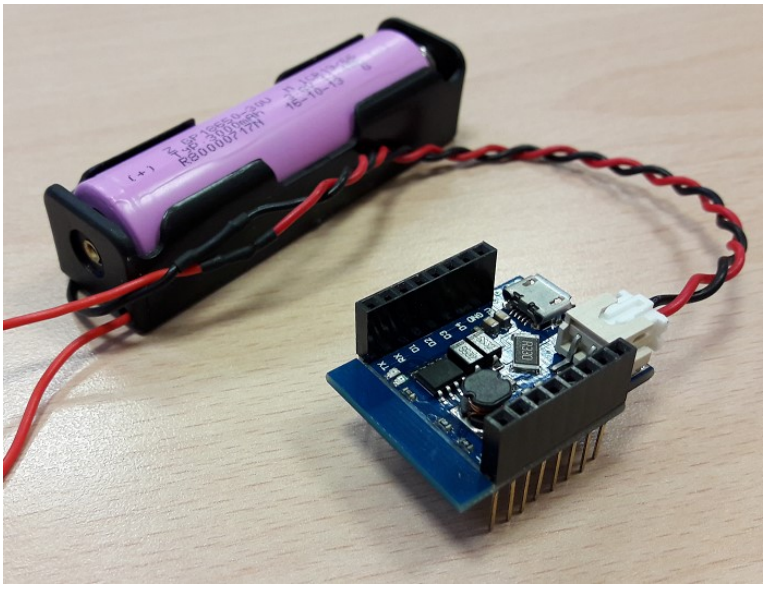


Fig. 7. Batteria al Litio e shield Wemos Battery

4 – SOFTWARE DEL NODO SENSORE E DEI NODI DI GESTIONE

4.1 Struttura del codice del nodo sensore

La programmazione del microcontrollore ESP8266 su cui è implementato il sensore wireless è stata effettuata in linguaggio C/C++ all'interno dell'IDE di Arduino. Di fatto, Arduino prevede un'implementazione di tipo bare-metal, quindi senza sistema operativo, e non consente l'accesso diretto allo scheduler. E' necessario soltanto definire la funzione `setup()`, che viene chiamata una sola volta all'avvio del sistema, e la funzione `loop()` che è richiamata periodicamente, dopo l'esecuzione delle funzioni di sistema (ad esempio, la gestione della radio WiFi).

La versione di Arduino utilizzata è la 1.8.4; sono state utilizzate, inoltre, le seguenti librerie:

- ESP8266 v. 1.0.0
- ESP8266WiFi v. 1.0.0
- SPI v. 1.0.0
- Wire v. 1.0.0
- WiFi v. 1.2.7
- PubSubClient v. 2.6.0

Lo pseudocodice della funzione `setup()` e della funzione `loop()` definite nel file `wemos_wireless_sensor.ino` è riportato in Appendice A. L'Appendice B riporta, invece, il file di *include* comprendente le definizioni delle costanti e delle strutture dati del sistema e i prototipi delle funzioni di supporto.

La funzione `setup()` inizializza le risorse utilizzate (porta seriale, pin GPIO e i bus 1-Wire, I2C e SPI); successivamente, legge i parametri di configurazione dalla EEPROM, effettua la connessione alla rete WiFi e al broker MQTT, effettua la sottoscrizione ai topic `/requestHello`, `/config`, `/datetime`, `/answers` su cui il client remoto invierà le proprie richieste ed, infine, pubblica un messaggio di presentazione sul topic `/hello`.

La funzione `loop()` esegue in sequenza i quattro step in cui è stata suddivisa la logica di funzionamento del sensore wireless:

1) controllo ed elaborazione dei messaggi ricevuti;
2) acquisizione dei dati;
3) formattazione del messaggio da inviare, incluso il timestamp;
4) invio del messaggio al broker sul topic `/data` (publish).

Se richiesto, vengono misurati i tempi di esecuzione dei quattro step, così da verificare quale sia più oneroso e da poter valutare la periodicità di esecuzione del loop principale. Infine, nel caso in cui il loop sia eseguito più velocemente rispetto alla periodicità impostata, l'esecuzione del loop principale viene messa in attesa per un opportuno lasso di tempo, consentendo allo scheduler di mandare in esecuzione le funzioni di sistema mediante l'utilizzo degli *interrupt*.

Va evidenziato che il messaggio inviato sul topic `/data` è formattato secondo la seguente struttura:

```
struct txdata_t {
  byte msg_id;
  byte board_id;
  byte board_type;
  byte yy;
  byte mth;
```

```
  byte dd;
  byte hh;
  byte mm;
  byte ss;
  float ch_a0;
  float ch_a1;
  float ch_a2;
  float ch_a3;
  float ch_a4;
  float ch_a5;
  float ch_a6;
  float ch_a7;
  float ch_1wire;
  float ch_i2c;
  float ch_spi;
};
```

I primi 3 campi della struttura contengono l'ID del messaggio, l'ID del sensore wireless ed un codice (`board_type`) indicante il tipo di misure effettuate dal sensore. I successivi 6 campi contengono il timestamp associato ai dati misurati. Gli ultimi 11 campi della struttura contengono le misure effettuate. In particolare, per quanto riguarda le 8 misure analogiche, sono possibili due modalità di funzionamento del sensore wireless:

- se il campo `board_type` =0xF0, il sensore sarà concepito per misurare direttamente gli 8 segnali analogici applicati agli ingressi del multiplexer, senza ulteriori elaborazioni;
- se, invece, il campo `board_type` =0xFE, il sensore sarà concepito per la misura di tensione e corrente mediante i canali 0 e 1 del multiplexer analogico e il successivo calcolo di tutti gli altri parametri elettrici; in tale ipotesi, gli 8 campi da `ch_a0` a `ch_a7` conterranno, rispettivamente, i seguenti valori: V_{dc}, V_{rms}, I_{dc}, I_{rms}, P_{dc}, P, A, T, con ovvio significato dei simboli utilizzati. L'algoritmo con cui vengono calcolate le suddette grandezze elettriche è spiegato nel paragrafo seguente.

4.2 Algoritmo per il calcolo delle grandezze elettriche

Sfruttando il principio di sovrapposizione degli effetti, i segnali elettrici (tensione o corrente) possono essere scomposti in una componente costante pari al proprio valore medio (componente continua o DC) ed una alternata (AC) sovrapposta alla prima. In particolare, la componente AC ha di solito forma d'onda sinusoidale, come mostrato in Fig. 8, ed è caratterizzata dai seguenti parametri:

1 – valore di picco V_{peak}
2 – valore picco-picco V_{pp}
3 – valore efficace V_{rms}
4 – periodo T

In particolare, è sufficiente conoscere i parametri 3 e 4, poiché per un segnale sinusoidale valgono le seguenti relazioni: **Errore. L'origine riferimento non è stata trovata.** e **Errore. L'origine riferimento non è stata trovata.**.

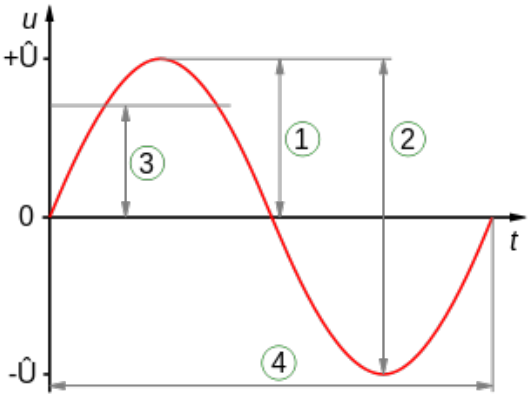


Fig. 8. Parametri di un'onda sinusoidale

Per effettuare la misura dei parametri elettrici delle due componenti (DC e AC) di un segnale elettrico mediante ADC bisogna procedere come segue:

- Acquisire un numero N sufficiente di campioni di tensione e corrente ($sample_i$) utilizzando una frequenza di campionamento F_s che rispetti il teorema di Shannon-Nyquist; in pratica, per ottenere buoni risultati, la frequenza di campionamento deve essere almeno 50 volte maggiore della frequenza del segnale da acquisire;
- Scalare i campioni acquisiti tenendo conto della risoluzione M dell'ADC e del suo range di ingresso V_{max}:
  - **Errore. L'origine riferimento non è stata trovata.** e analogamente per i_i
- Calcolare il valore DC come media del segnale istantaneo:
  - **Errore. L'origine riferimento non è stata trovata.** e **Errore. L'origine riferimento non è stata trovata.**
- Calcolare la potenza DC come prodotto di tensione e corrente DC:
  - **Errore. L'origine riferimento non è stata trovata.**
- Determinare le componenti AC di tensione e corrente sottraendo le componenti DC dai campioni acquisiti:
  - **Errore. L'origine riferimento non è stata trovata.** e **Errore. L'origine riferimento non è stata trovata.**
- Calcolare i valori efficaci di tensione e corrente AC come segue:
  - **Errore. L'origine riferimento non è stata trovata.** e **Errore. L'origine riferimento non è stata trovata.**
- Calcolare la potenza apparente A e la potenza attiva P come segue:
  - **Errore. L'origine riferimento non è stata trovata.** e **Errore. L'origine riferimento non è stata trovata.**
- In generale, il calcolo del periodo può essere effettuato in diversi modi; nel caso in esame, è stato effettuato calcolando il tempo che la forma d'onda acquisita impiega per ripetere la transizione da valori negativi a valori positivi. In particolare, è stato realizzato utilizzando un vettore di appoggio contenente 0 qualora il valore del campione dovesse essere negativo, 1 se positivo. In questo modo, il cambio di stato tra 0 ed 1 indica il passaggio per lo zero della forma d'onda e l'inizio del periodo. Pertanto:
  - **Errore. L'origine riferimento non è stata trovata.** dove Δx è la differenza tra gli indici che identificano il passaggio fra 0 e 1.

4.3 Struttura del codice del gestore della configurazione

Il linguaggio di programmazione utilizzato per implementare il gestore della configurazione è Python 2.7; sono state utilizzate, inoltre, le seguenti librerie:

- re (supporto per le espressioni regolari)
- datetime (manipolazione di data e ora)
- time (accesso al clock di sistema e conversioni varie)
- sys (funzioni di sistema)
- paho (interfacciamento con il broker MQTT)
- struct (formattazione di sequenze di byte).

Il codice del modulo Python `config_manager.py` che implementa il gestore della configurazione è riportato in Appendice C. Il suo funzionamento è molto semplice. Viene chiesto all'utente se desidera effettuare la configurazione e, in caso affermativo, se intende soltanto aggiornare la data e l'ora oppure effettuare una riconfigurazione completa. Il modulo si connette al broker utilizzando il suo indirizzo IP e il numero di porta; quindi, sottoscrive il topic `/answers` per poter ricevere un feedback da parte dei nodi sensore sull'esito dell'operazione di configurazione. Il modulo formatta il messaggio con la data e l'ora corrente o con l'intero set di parametri di configurazione in un formato compatibile con le corrispondenti strutture definite in linguaggio C nel file `ws_support_fcns.h` riportato in Appendice B; quindi, procede a pubblicarlo sul topic `/config` e rimane in attesa fin quando ottiene risposta da parte di tutti i nodi sensore a cui è stato inviato il messaggio, aspettando fino ad un massimo di 10 secondi. In caso di esito positivo, si riprende il normale funzionamento, ovvero l'esecuzione viene trasferita al modulo `store_it.py` che implementa il gestore della persistenza che si occupa di ricevere e memorizzare i dati. In caso contrario, l'esecuzione termina con un errore.

È opportuno notare che l'attività oggetto del presente report riguarda principalmente l'implementazione del nodo sensore. Pertanto, sono state implementate soltanto le funzionalità minime del modulo Python `config_manager.py` che hanno consentito di testare il corretto funzionamento del sensore. Conseguentemente, i vari parametri di configurazione (nomi utente, password, indirizzo IP, etc.) sono attualmente hard-coded all'interno del suddetto modulo Python e non viene effettuata una completa gestione degli errori.

4.3 Struttura del codice del gestore della persistenza

Anche il gestore della persistenza è stato implementato come modulo Python; specificamente, sono state utilizzate le seguenti librerie:

- datetime (manipolazione di data e ora)
- time (accesso al clock di sistema e conversioni varie)
- sys (funzioni di sistema)
- paho (interfacciamento con il broker MQTT)
- struct (formattazione di sequenze di byte)
- sqlite3 (interfacciamento con database SQLite).

Il codice del modulo Python store_it.py che implementa il gestore della persistenza è riportato in Appendice D. Il suo funzionamento è di seguito descritto. Il modulo si connette al broker utilizzando il suo indirizzo IP e il numero di porta; quindi, si sottoscrive al topic `/hello` per poter ricevere le risposte da parte dei nodi sensore ed effettua la publish in modalità broadcast della

richiesta di presentazione. In tal modo viene implementata la fase di discovery dei nodi sensore, che dura 10 secondi. In seguito alle risposte fornite dai nodi sensori in tale lasso di tempo, viene costruita una lista delle board presenti all'interno del sistema. Al termine della fase di discovery il modulo si disconnette dal broker. Nel caso in cui la lista delle board del sistema sia vuota, l'esecuzione termina con un errore. In caso contrario, viene inizializzata la connessione al database su cui memorizzare i dati ricevuti, viene effettuata una nuova connessione al broker e il modulo si sottoscrive a due topic (`/hello` e `/data`), associando le rispettive funzioni callback. Ogni volta che arriva un messaggio sul topic `/data`, esso viene processato estraendo i dati misurati, stampandoli sulla console a scopo di debug e memorizzandoli sul database sfruttando le funzioni della libreria sqlite3. Qualora arrivassero nuovi messaggi sul topic `/hello`, sarebbero aggiunte ulteriori board alla lista dei nodi sensore.

Come detto in precedenza, l'attività oggetto del presente report riguarda principalmente l'implementazione del nodo sensore. Pertanto, sono state implementate soltanto le funzionalità minime del modulo Python `store_it.py` che hanno consentito di testare il corretto funzionamento del sensore. Conseguentemente, i vari parametri di configurazione (nomi utente, password, indirizzo IP, etc.) sono attualmente hard-coded all'interno del suddetto modulo Python e non viene effettuata una completa gestione degli errori.

5. VERIFICA SPERIMENTALE

5.1 Test delle funzionalità

Sono stati eseguiti diversi test che hanno confermato il corretto funzionamento dei componenti hardware e software del sensore e dei nodi di gestione (configurazione e persistenza dei dati). Le Fig. 9 e 10 mostrano due screenshot delle console di debug, di immediata interpretazione.

L'esecuzione del loop del nodo sensore richiede non più di 500 ms. La trasmissione dei dati al broker richiede normalmente qualche centinaio di millisecondi, anche se il tempo può aumentare fino a 5 secondi in caso di congestione della rete. Ad ogni modo, in caso di disconnessione dalla rete WiFi o dal broker, il sensore wireless provvede automaticamente ad effettuare la riconnessione.

```
Connecting to issia1.................................
WiFi connected - IP address: 192.168.0.100
Attempting MQTT connection... connected
topics subscribed
Hello message sent: wemos_33_254
setup completed

Electrical meas.:
Vdc=0.001000
Vrms=228.403000
Idc=0.000200
Irms=3.982000
Pdc=0.000000
P=873.120728
A=909.500732
T=0.020005
Temp. meas.:
25.90
I2C meas.:
1.00
SPI meas.:
123.456
txdata buffer has been filled
data sent
loop iteration took 816629 us

Electrical meas.:
```

Fig. 9. Screenshot della console seriale del nodo sensore implementato su scheda Wemos

```
[giuseppe@localhost tirocinio_la_grassa]$ python mqtt_client/config_m
anager.py
Do you want programming [y/n]? y
Datetime or general config [d/c]? d
Wemos: wemos0 Answer: wemos_33_254_1
boards programmed


Starting module store_it
Store_it module active
discovering boards...
Hello client, this is wemos0 Answer: wemos_33_254
1 boards answered
running...
wemos0 id: 33 type: 0xfe -- 2018-03-07 13:43:14
0.00    228.40  0.00    3.98    0.00    873.12  909.50  0.0200  25.90
3.00    123.46
wemos0 id: 33 type: 0xfe -- 2018-03-07 13:43:16
0.00    228.40  0.00    3.98    0.00    873.12  909.50  0.0200  25.90
3.00    123.46
wemos0 id: 33 type: 0xfe -- 2018-03-07 13:43:18
0.00    228.40  0.00    3.98    0.00    873.12  909.50  0.0200  25.90
3.00    123.46
wemos0 id: 33 type: 0xfe -- 2018-03-07 13:43:20
0.00    228.40  0.00    3.98    0.00    873.12  909.50  0.0200  25.90
3.00    123.46
wemos0 id: 33 type: 0xfe -- 2018-03-07 13:43:22
0.00    228.40  0.00    3.98    0.00    873.12  909.50  0.0200  25.90
3.00    123.46
```

Fig. 10. Screenshot della console di Python durante l'esecuzione dei moduli config_manager.py e store_it.py

5.2 Test di portata e affidabilità della connessione wireless dei sensori

Un ulteriore test è stato effettuato per valutare la perdita di pacchetti e la portata del segnale WiFi quando il sensore wireless utilizza l'antenna ceramica integrata. Su 1000 pacchetti inviati dal sensore al broker, tutti i pacchetti sono stati ricevuti correttamente per distanze fino a 35 m anche in presenza di ostacoli e con QoS 0 (senza conferma di avvenuta ricezione dell'informazione). Ovviamente, il tempo richiesto per la trasmissione dati aumenta con la distanza. Ciò è dovuto al protocollo IEEE802.11, che prevede la possibilità di adattare dinamicamente la velocità di trasmissione a seconda delle condizioni del canale. Oltre i 35 m si cominciano a perdere pacchetti. Tale range è adeguato all'applicazione in esame ma, se necessario, può essere esteso utilizzando un'antenna esterna.

Per poter evidenziare i vantaggi legati all'uso del protocollo MQTT, sono stati confrontati i tempi di trasmissione, una volta mediante protocollo HTTP e successivamente mediante MQTT. In entrambi i casi la lunghezza del payload è pari a 4 byte (escluso header, che nel caso di HTTP è molto più grande: ~100-800 bytes contro i 2-5 bytes di MQTT).

Sono stati confrontati i risultati ottenuti effettuando una Request/Response HTTP e una publish effettuata da un client MQTT con QoS 0. I tempi misurati nel caso di HTTP sono 9-50 millisecondi. Utilizzando il broker MQTT Mosquitto, i tempi misurati sono di circa 2-4 millisecondi (con QoS 0 e senza abbattere la connessione).

CONCLUSIONI E SVILUPPI FUTURI

È stato progettato e realizzato un prototipo di sensore wireless per il monitoraggio di carichi elettrici in ambiente Smart Building, in accordo con le specifiche individuate. Il sensore acquisisce i segnali di tensione e corrente sul carico elettrico, calcola ulteriori parametri elettrici (potenza AC e DC, periodo, etc.) e li invia attraverso la rete WiFi ad un broker di tipo Message Queue Telemetry Transportation (MQTT). Quest'ultimo inoltra i messaggi ad un client remoto che si sottoscrive ai rispettivi topic. Il client remoto riceve i dati, li memorizza in un database e li elabora per varie finalità, ad esempio per l'implementazione di sistemi EMS (Energy Management System) per un uso ottimale dell'energia elettrica all'interno dello Smart Building.

Il prototipo è stato realizzato collegando insieme una scheda embedded con radio WiFi, diversi dispositivi hardware commerciali ed alcuni circuiti di condizionamento dei segnali realizzati appositamente per l'applicazione in esame.

La programmazione del microcontrollore ESP8266 su cui è implementato il sensore wireless è stata effettuata in linguaggio C/C++ all'interno dell'IDE di Arduino. Il broker MQTT Mosquitto è stato installato su una scheda embedded basata su Linux. L'applicazione client per PC che riceve e memorizza i dati è stata sviluppata in Python.

Il corretto funzionamento dell'hardware e del software è stato verificato sperimentalmente.

Si ritiene che il sensore wireless possa essere ulteriormente migliorato riducendone le dimensioni mediante la realizzazione di una PCB e l'utilizzo di componenti SMD. Rispetto alla versione prototipale, la versione definitiva potrebbe anche utilizzare sensori di corrente con portata maggiore e sfruttare un opportuno circuito per consentire anche l'alimentazione da rete.

BIBLIOGRAFIA

[1] J. Pan, R. Jain, S. Paul, T. Vu, A. Saifullah, and M. Sha, "An Internet of Things Framework for Smart Energy in Buildings: Designs, Prototype, and Experiments," IEEE Internet Things J., vol. 2, no. 6, pp. 527–537, 2015.

[2] X. Fang, S. Misra, G. Xue, and D. Yang, "Smart Grid — The New and Improved Power Grid: A Survey," IEEE Commun. Surv. Tutorials, vol. 14, no. 4, pp. 944–980, 2012.

[3] S. Teleke, L. Oehlerking, and M. Hong, "Nanogrids with energy storage for future electricity grids," 2014 IEEE PES T&D Conf. Expo., pp. 1–5, 2014.

[4] J. Mitra and S. Suryanarayanan, "System analytics for smart microgrids," IEEE PES Gen. Meet. PES 2010, pp. 1–4, 2010.

[5] M. Biabani, M. A. Golkar, A. Johar, and M. Johar, "Propose a home demand-side-management algorithm for smart nano-grid," PEDSTC 2013 - 4th Annu. Int. Power Electron. Drive Syst. Technol. Conf., pp. 487–494, 2013.

[6] D. Zhang, N. Shah, and L. G. Papageorgiou, "Efficient energy consumption and operation management in a smart building with microgrid," Energy Convers. Manag., vol. 74, pp. 209–222, Oct. 2013.

[7] M. C. Di Piazza, G. La Tona, M. Luna, and A. Di Piazza, "A two-stage Energy Management System for smart buildings reducing the impact of demand uncertainty," Energy Build., vol. 139, pp. 1–9, Mar. 2017.

[8] B. Asare-Bediako, W. L. Kling, and P. F. Ribeiro, "Home energy management systems: Evolution, trends and frameworks," in 2012 47th International Universities Power Engineering Conference (UPEC), 2012, pp. 1–5.

[9] "ISO/IEC 20922:2016: Information technology -- Message Queuing Telemetry Transport (MQTT) v3.1.1," International Organization for Standardization, Geneva, Switzerland, 2016.

[10] Y. Cao, T. Jiang, and Z. Han, "A Survey of Emerging M2M Systems: Context, Task, and Objective," IEEE Internet Things J., vol. 3, no. 6, pp. 1246–1258, Dec. 2016.

[11] R. A Light, "Mosquitto: server and client implementation of the MQTT protocol," J. Open Source Softw., vol. 2, no. 13, p. 265, May 2017.

[12] "D1 mini Pro [WEMOS Electronics]" [Online]. Available: https://wiki.wemos.cc/products:d1:d1_mini_pro [Accessed: 23-Jan-2018]

[13] "Arduino – Software" [Online]. Available: https://www.arduino.cc/en/Main/Software [Accessed: 23-Jan-2018].

[14] "Eclipse Paho" [Online]. Available: http://wiki.eclipse.org/Paho [Accessed: 23-Jan-2018].

[15] "A10-OLinuXino-LIME – Olimex" [Online]. Available: https://www.olimex.com/wiki/A10-OLinuXino-LIME [Accessed: 23-Jan-2018]

APPENDICE A: wemos_wireless_sensor.ino

```
#include "ws_support_fcns.h"
#include <ESP8266WiFi.h>
#include "PubSubClient.h"
#include <SPI.h>
#include <Wire.h>
#include "DHT.h"
DHT dht(PIN_1WIRE, DHT22);

WiFiClient espClient;
PubSubClient client(espClient);

//global variables
//default wemos parameters
static const byte default_board_id = 0x21;
static const byte default_board_type = 0xF0;
//board_type = 0x00 means unprogrammed board
//board_type = 0xFF is the broadcast address

//default wifi credentials
static const char* const default_wifi_ssid = "xxxxx";
static const char* const default_wifi_pwd = "xxxxx";

//default broker credentials
static const char default_mqtt_addr[16] = "xxx.xxx.xxx.xxx";
static const unsigned int default_mqtt_port = 8883;
static const char* const default_mqtt_user = "xxxxx";
static const char* const default_mqtt_pwd = "xxxxx";

//other global variables
static char mqtt_addr[16];
static unsigned int mqtt_port;
static struct config_data_t config_data;
static byte *aconfig_data = (byte *) &config_data; //alias of config_data as an
array
static char mybuffer[200]; //a buffer to build formatted strings
static byte rxdata[RXDATA_BUFSIZE];
static unsigned int rxdatalen;


//*************************************************************************
//callback for retrieving new messages
void mqtt_callback(char* topic, byte* payload, unsigned int length) {
  if (length <= RXDATA_BUFSIZE) {
    memcpy(rxdata, payload, length);
    rxdatalen = length;
  }
  else {
    rxdatalen = 0;
  }
}

//*************************************************************************
void setup()
```

```
{
  initialize_peripherals();

  //read config data from eeprom
  read_config_data_from_eeprom(aconfig_data);

  //fill mqtt_addr and mqtt_port (global variables)
  buildIPaddress(mqtt_addr, config_data.MQTT_IPaddr_3, config_data.MQTT_IPaddr_2,
                 config_data.MQTT_IPaddr_1, config_data.MQTT_IPaddr_0);
  mqtt_port = buildIPport(config_data.MQTT_port_1, config_data.MQTT_port_0);

  //print all credentials
  Serial.println();
  snprintf(mybuffer, sizeof(mybuffer), "SSID: %s, pwd: %s, broker addr: %s, port:
           %u, user: %s, pwd: %s",
           config_data.WiFi_SSID, config_data.WiFi_pwd,
           mqtt_addr, mqtt_port, config_data.MQTT_user, config_data.MQTT_pwd);
  Serial.println(mybuffer);

  //setup WiFi connection
  WiFi.mode(WIFI_STA);
  setup_wifi(config_data.WiFi_SSID, config_data.WiFi_pwd);

  //setup MQTT server
  client.setServer(mqtt_addr, mqtt_port);
  client.setCallback(mqtt_callback);

  //connect to broker and subscribe to config manager's topic
  mqtt_reconnect("WemosClient", config_data.MQTT_user, config_data.MQTT_pwd);

  //send hello msg
  snprintf(mybuffer, sizeof(mybuffer), "wemos_%u_%u", config_data.board_id,
       config_data.board_type);
  client.publish("/hello", mybuffer); //topic, payload
  Serial.print("Hello message sent: ");
  Serial.println(mybuffer);
}

//*********************************************************************

void loop() {
  static bool toggle_flag = false;
  static unsigned long initial_time_us, start_time_us, stop_time_us;
  static struct channels_t channels;
  static float *achannels = (float *) &channels;
  static struct txdata_t txdata;
  static byte *atxdata = (byte *) &txdata;

  initial_time_us = micros();

  //STEP 1: process new messages
  start_time_us = micros();
  client.loop();

  if (/* msg arrived and sent to this wemos or a broadcast msg */) {
    if (msg_ID == MSG_ID_WHO_ARE_YOU) {
        snprintf(mybuffer, sizeof(mybuffer), "wemos_%u_%u", config_data.board_id,
```

```
            config_data.board_type);
      if (!client.connected()) {
        // reconnect to server mqtt
        mqtt_reconnect("WemosClient", config_data.MQTT_user, config_data.MQTT_pwd);
      }
      client.publish("/hello", mybuffer); //topic, payload
      Serial.print("Hello message sent: ");
      Serial.println(mybuffer);
    }
    else if (msg_ID == MSG_ID_CONFIG) {
      //store config_data in EEPROM
      aconfig_data = &rxdata[2];
      bool b = program_eeprom(aconfig_data);

      if (!client.connected()) {
        // reconnect to server mqtt
        mqtt_reconnect("WemosClient", config_data.MQTT_user, config_data.MQTT_pwd);
      }
      snprintf(mybuffer, sizeof(mybuffer), "wemos_%u_%u_%u", aconfig_data[0],
            aconfig_data[1], b);
      client.publish("wemos0/answers", mybuffer);
      Serial.println("Sent programming result to broker");

      //alert user on console
      Serial.println("Trying to reset the board...");
      ESP.restart();
    }
    else if (msg_ID == MSG_ID_SETDATETIME) {
      program_RTCC(RTCC_ADDR, rxdata);
      Serial.println("RTCC programmed");

      if (!client.connected()) {
        // reconnect to server mqtt
        mqtt_reconnect("WemosClient", config_data.MQTT_user, config_data.MQTT_pwd);
      }
      snprintf(mybuffer, sizeof(mybuffer), "wemos_%u_%u_%u", aconfig_data[0],
            aconfig_data[1], 1);
      client.publish("wemos0/answers", mybuffer);
      Serial.println("Sent confirmation of setdatetime to broker");
    }
  }
  rxdatalen = 0; // mark the msg as read
  stop_time_us = micros();
  if (PARTIAL_EXEC_TIME)
    print_elapsed_time("msg processing finished in ", start_time_us, stop_time_us);

  //STEP 2: acquire data from hw sensors
  //acquire analog channels
  if (config_data.board_type == 0xFE) {
    //acquire V and I and compute electrical quantities
    start_time_us = micros();
    acquire_and_process_v_and_i(&channels);
    stop_time_us = micros();
    if (PARTIAL_EXEC_TIME)
      print_elapsed_time("electrical acquisition finished in ", start_time_us,
stop_time_us);
    if (VERBOSE) {
```

```
      Serial.println("Electrical meas.:");
      Serial.print("Vdc=");
      Serial.println(channels.ch_a0, 6);
      Serial.print("Vrms=");
      Serial.println(channels.ch_a1, 6);
      Serial.print("Idc=");
      Serial.println(channels.ch_a2, 6);
      Serial.print("Irms=");
      Serial.println(channels.ch_a3, 6);
      Serial.print("Pdc=");
      Serial.println(channels.ch_a4, 6);
      Serial.print("P=");
      Serial.println(channels.ch_a5, 6);
      Serial.print("A=");
      Serial.println(channels.ch_a6, 6);
      Serial.print("T=");
      Serial.println(channels.ch_a7, 6);
    }
  }
  else {
    //acquire raw analog signals
    start_time_us = micros();
    acquire_raw_analog_channels(&channels);
    stop_time_us = micros();
    if (PARTIAL_EXEC_TIME)
      print_elapsed_time("raw analog acquisition finished in ", start_time_us,
stop_time_us);
    if (VERBOSE) {
      Serial.println("Raw analog meas.:");
      Serial.println(channels.ch_a0, 6);
      Serial.println(channels.ch_a1, 6);
      Serial.println(channels.ch_a2, 6);
      Serial.println(channels.ch_a3, 6);
      Serial.println(channels.ch_a4, 6);
      Serial.println(channels.ch_a5, 6);
      Serial.println(channels.ch_a6, 6);
      Serial.println(channels.ch_a7, 6);
    }
  }

  //acquire 1-Wire data
  start_time_us = micros();
  channels.ch_1wire = dht.readTemperature();
  stop_time_us = micros();
  if (PARTIAL_EXEC_TIME)
    print_elapsed_time("1-Wire acquisition finished in ", start_time_us,
stop_time_us);
  if (VERBOSE) {
    Serial.println("Temp. meas.:");
    Serial.println(channels.ch_1wire, 2);
  }

  //acquire I2C data
  start_time_us = micros();
  channels.ch_i2c = ReadI2CByte(I2C_SENSOR_ADDR, I2C_SENSOR_REG);
  stop_time_us = micros();
  if (PARTIAL_EXEC_TIME)
```

```
    print_elapsed_time("I2C acquisition finished in ", start_time_us, stop_time_us);
  if (VERBOSE) {
    Serial.println("I2C meas.:");
    Serial.println(channels.ch_i2c, 2);
  }

  //acquire SPI data
  start_time_us = micros();
  channels.ch_spi = ReadSpiByte(SPI_SENSOR_REG);
  stop_time_us = micros();
  if (PARTIAL_EXEC_TIME)
    print_elapsed_time("SPI acquisition finished in ", start_time_us, stop_time_us);
  if (VERBOSE) {
    Serial.println("SPI meas.:");
    Serial.println(channels.ch_spi, 3);
  }

  //STEP 3: retrieve timestamp and prepare data
  byte data;

  start_time_us = micros();
  txdata.msg_id = MSG_ID_NEWDATA;
  txdata.board_id = config_data.board_id;
  txdata.board_type = config_data.board_type;

  getDateTimeFromRtcc(RTCC_ADDR, &txdata);

  fill_data_structure(atxdata, achannels);

  stop_time_us = micros();
  if (PARTIAL_EXEC_TIME)
    print_elapsed_time("data preparing finished in ", start_time_us, stop_time_us);

  //STEP 4: send atxdata to broker
  start_time_us = micros();
  if (!client.connected()) {
    // reconnect to server mqtt
    mqtt_reconnect("WemosClient", config_data.MQTT_user, config_data.MQTT_pwd);
  }
  client.publish("wemos0/data", atxdata, TXDATA_LEN);
  Serial.println("Newdata sent");
  stop_time_us = micros();
  if (PARTIAL_EXEC_TIME)
    print_elapsed_time("sending msg finished in ", start_time_us, stop_time_us);

  //print total time
  stop_time_us = micros();
  if (VERBOSE)
    print_elapsed_time("loop iteration took ", initial_time_us, stop_time_us);
  Serial.println();

  //wait delta time
  long deltaT = TLOOP_US - (micros() - initial_time_us);
  if (deltaT > 0) {
    delayMicroseconds(deltaT);
  }
}
```

APPENDICE B: ws_support_fcns.h

```
//ws_support_fcns.h

#ifndef WS_SUPPORT_FCNS_H_
#define WS_SUPPORT_FCNS_H_

#include <Arduino.h>
#include "PubSubClient.h"

//operating mode
#define VERBOSE 0
#define PARTIAL_EXEC_TIME 0
#define TLOOP_US 2000000

//pin definitions
#define PIN_LED D4
#define PIN_MUX_S0 D0
#define PIN_MUX_S1 D3
#define PIN_MUX_S2 D8
#define PIN_1WIRE D4
#define PIN_I2C_SCL D1
#define PIN_I2C_SDA D2
#define PIN_SPI_SCLK D5
#define PIN_SPI_MISO D6
#define PIN_SPI_MOSI D7
#define PIN_SPI_SSn D8

//HW peripherals
#define EEPROM_SIZE 128
#define EEPROM_ADDR 0x57
#define RTCC_ADDR 0x6F
#define I2C_SENSOR_ADDR 0x6F
#define I2C_SENSOR_REG 0x05

//ADC parameters
#define NSAMPLES 1000
#define TSAMPLE_US 200
//sampling freq. is 5 kHz (max 8 kHz)
#define ADC_CALIB_GAIN 1.0625

//topics on which msgs will be published:
//hello
//data

//topics to be subscribed from wemos board:
//requestHello
//config
//datetime
//answers

//type definitions:
//signal values, i.e., either 8 raw measured data
//or Vdc, Vrms, Idc, Irms, Pdc, P, A, T
#define CHANNELS_LEN 11
```

```
struct channels_t {
  float ch_a0;
  float ch_a1;
  float ch_a2;
  float ch_a3;
  float ch_a4;
  float ch_a5;
  float ch_a6;
  float ch_a7;
  float ch_1wire;
  float ch_i2c;
  float ch_spi;
};

//msg id for messages sent by remote client
#define MSG_ID_WHO_ARE_YOU 0x68
/* msg format
struct {
  byte msg_id = MSG_ID_WHO_ARE_YOU
  byte current_board_id or 0xFF for broadcast;
  byte verbose;
}
*/
#define MSG_ID_CONFIG 0x63
/* msg format
struct {
  byte msg_id = MSG_ID_CONFIG
  byte current_board_id or 0xFF for broadcast;
  ---
  byte new board_id
  byte board_type;
  byte MQTT_IPaddr_3;
  ...see below...
}
*/
#define MSG_ID_SETDATETIME 0x05
/* msg format
struct {
  byte msg_id = MSG_ID_SETDATETIME
  byte current_board_id or 0xFF for broadcast;
  byte yy;
  byte mth;
  byte dd;
  byte hh;
  byte mm;
  byte ss;
};
*/

//config data sent by remote client and stored in EEPROM
#define CONFIG_DATA_STR_START_IDX 8
#define CONFIG_DATA_STR_SIZE 30
#define CONFIG_DATA_STR_NUM 4
#define CONFIG_DATA_LEN (CONFIG_DATA_STR_START_IDX + CONFIG_DATA_STR_NUM *
CONFIG_DATA_STR_SIZE)
#define RXDATA_BUFSIZE (CONFIG_DATA_LEN + 2)
struct config_data_t {
```

```
  byte board_id;
  byte board_type;
  byte MQTT_IPaddr_3;
  byte MQTT_IPaddr_2;
  byte MQTT_IPaddr_1;
  byte MQTT_IPaddr_0;
  byte MQTT_port_1;
  byte MQTT_port_0;
  char MQTT_user[CONFIG_DATA_STR_SIZE];
  char MQTT_pwd[CONFIG_DATA_STR_SIZE];
  char WiFi_SSID[CONFIG_DATA_STR_SIZE];
  char WiFi_pwd[CONFIG_DATA_STR_SIZE];
};

//msg id for messages sent by wireless sensor
#define MSG_ID_NEWDATA 0x22
//measured data sent by wireless sensor to remote client
#define TXDATA_LEN 53
#define TXDATA_CH_START_IDX 9
struct txdata_t {
  byte msg_id;
  byte board_id;
  byte board_type;
  byte yy;
  byte mth;
  byte dd;
  byte hh;
  byte mm;
  byte ss;
  float ch_a0;
  float ch_a1;
  float ch_a2;
  float ch_a3;
  float ch_a4;
  float ch_a5;
  float ch_a6;
  float ch_a7;
  float ch_1wire;
  float ch_i2c;
  float ch_spi;
};

//function prototypes
bool program_eeprom(byte *aconfig_data);
void read_config_data_from_eeprom(byte *aconfig_data);
byte ReadI2CByte(byte addr, byte reg);
void WriteI2CByte(byte addr, byte reg, byte data);
void WriteSPIByte(byte value);
void toggle_error_led(int pin);
void toggle_confirmation_led(int pin);
void set_mux_ch(unsigned int ch);
void acquire_raw_analog_channels(struct channels_t *channels);
void acquire_and_process_v_and_i(struct channels_t *channels);
float compute_period(float *v, float *signs);

String rtcDate(void);
String rtcTime(void);
```

```
void timestamp(void);
void print_elapsed_time(String msg, unsigned long start_time_us,
                        unsigned long stop_time_us);
void splitIPaddress(const char *ipstr, byte *addr3, byte *addr2,
                    byte *addr1, byte *addr0);
void buildIPaddress(char *ipstr, byte addr3, byte addr2, byte addr1, byte addr0);
void splitIPport(unsigned int port, byte *hi, byte *lo);
unsigned int buildIPport(byte hi, byte lo);
void dump_hex_bytes(byte *mybuffer, int mybufferlen);
void subscribe_topics(PubSubClient client);
void unsubscribe_topics(PubSubClient client);

#endif /* WS_SUPPORT_FCNS_H_ */
```

APPENDICE C: config_manager.py

```
import re
import datetime
import time

import paho.mqtt.client as paho
import struct

def decimalToBCD(date):
    u=(date.year % 100) % 10
    d=(date.year % 100) // 10
    c_year=((d << 4) + u)

    u=date.month % 10
    d=date.month // 10
    c_month=((d << 4) + u)

    u=date.day % 10
    d=date.day // 10
    c_day=((d << 4) + u)

    u=date.hour % 10
    d=date.hour // 10
    c_hour=((d << 4) + u)

    u=date.minute % 10
    d=date.minute // 10
    c_minute=((d << 4) + u)

    u=date.second % 10
    d=date.second // 10
    c_second=((d << 4) + u)

    return c_year,c_month,c_day,c_hour,c_minute,c_second

global all_wemos_response


def answer(client, userdata, msg):
    global package
    package = 0
    name_board=(msg.topic).split('/', 1)[0]
    board_list=[]
    board_list.append(msg.payload)
    print("Wemos: {} Answer: {}".format(name_board, msg.payload))
    package=package+1
    if (package == 1):
        all_wemos_response=True
        package=0

        reply=raw_input('Do you want programming [y/n]? ')
flag=False
all_wemos_response=False


if (re.search('[yes|y|Y|Yes|YES]',reply)):
```

```
    reply1=raw_input('Datetime or general config [d/c]? ')
    client = paho.Client()
    client.username_pw_set("*****", "*****")

    client.connect("***.***.***.***", 8883)
    client.subscribe("+/answers",qos=0)
    client.message_callback_add("+/answers", answer)

    if (re.search('[d|D|Datetime|datetime|DATETIME]',reply1)):
        date=(datetime.datetime.now())
        year,month,day,hour,minute,second=decimalToBCD(date)


        x=struct.pack("cccccccc",chr(0x05),chr(0xFF),chr(year),chr(month),chr
        (day),
                  chr(hour),chr(minute),chr(second)) #data reprogramming
        client.publish("/datetime" ,x, 0)

    if (re.search('[c|C|Config|CONFIG]',reply1)):
        user="*****"
        passw="*****"
        ssid="*****"
        wifi_passw="*****"
        port = 8883

        x=struct.pack("!ccccccccH30s30s30s30s",chr(0x63),chr(0xFF),chr(0x21),
        chr(0xFE),chr(150),

        chr(145),chr(127),chr(45),port,user,passw,ssid,wifi_passw) #data
        reprogramming
        client.publish("/config" ,x,0)

    client.loop_start()

    start=time.time()
    while not(all_wemos_response) and (time.time() - start < 10.0):
       # I'm waiting till all wemos answer ok
        pass
    client.loop_stop()
    client.disconnect()
    all_wemos_response=False
    flag=True


if (re.search('[no|N|n|No|NO]',reply) or flag):
        print("Starting module store_it")
        import store_it #call module as subprocess of this process
```

APPENDICE D: store_it.py

```
import datetime
import time
from sys import exit
import paho.mqtt.client as paho
import struct
import sqlite3


global board_list
board_list=[]

def init_db():
    #omissis
    pass


def add_to_db(buffer_b):
    #omissis
    pass


def on_subscribe(client, userdata, mid, granted_qos):
    print("Subscribed: " + str(mid) + " " + str(granted_qos))


def hello(client, userdata, msg):
    board_name=(msg.topic).split('/', 1)[0]
    board_list.append(msg.topic)
    print("Hello client, this is {} Answer: {}".format(board_name,
msg.payload))


def store_data(client, userdata, msg):
    buffer_b=bytearray()
    buffer_b.extend(msg.payload)
    #store new data
    add_to_db(buffer_b)
    #print new formatted data
    board_name=(msg.topic).split('/', 1)[0]
    board_id=ord(msg.payload[1])
    board_type=ord(msg.payload[2])
    date = '{:02d} {:02d} {:02d} {:02d} {:02d} {:02d}'.format(*[bcdToInt(i)
for i in msg.payload[3:9]])

    date = datetime.datetime.strptime(date,"%y %m %d %H %M %S")

    (Vdc, Vrms, Idc, Irms, Pdc, P, A, T, wire, i2c, spi) =
struct.unpack_from("!fffffffffff", buffer_b, 9)

    fmtString = "{:.2f}\t" * 7 + "{:.4f}\t" + "{:.2f}\t" * 3
    print("{} id: {} type: 0x{:0x} -- {}".format(board_name, board_id,
board_type, date))
    print(fmtString.format(Vdc, Vrms, Idc, Irms, Pdc, P, A, T, wire, i2c,
spi))
```

```
def bcdToInt(bcdStr):
    bcd, = struct.unpack('b', bcdStr)
    bcdInt = (bcd >> 4) * 10 + (bcd & 0x0f)
    return bcdInt

#main code

print("Store_it module active")
print("discovering boards...")

client = paho.Client()
client.username_pw_set("*****", "*****")
client.connect("***.***.***.***", 8883)

client.subscribe("wemos0/hello",qos=0)
client.message_callback_add("wemos0/hello", hello)

#send a broadcast request on topic /requestHello
x=struct.pack("ccc",chr(0x68),chr(0xFF),chr(0x00))
client.publish("/requestHello", x, 0)

#check for answers
client.loop_start()
start=time.time()

#allow 10 seconds for all wemos to respond

while (time.time() - start < 10.0):
   # print("waiting...")
    pass

client.loop_stop()
client.disconnect()

board_number = len(board_list)

if (board_number == 0):
    print("No board answered")
    exit()
else:
    print(str(board_number) + " boards answered")

#initialize database
init_db();

print("running...")

#initialized and starting new client for multiple subscriptions
client = paho.Client()
client.username_pw_set("*****", "*****")
client.connect("***.***.***.***", 8883)
client.subscribe("+/data", qos=0)
client.subscribe("+/hello", qos=0)

client.message_callback_add("+/data", store_data)
client.message_callback_add("+/hello", welcome)
client.loop_forever()
```

```
try:
    client.loop_forever()
except KeyboardInterrupt:
    print("Disconnecting...")
    client.disconnect()
    time.sleep(2.0)
    print("Exiting.")
```