

ESP32 EVSE

ATTENZIONE: visto che questo progetto prevede alcune parti connesse alla tensione di rete, non mi assumo nessuna responsabilità di eventuali danni, potete prendere quanto segue a scopo didattico e non come un invito alla realizzazione

Ho acquistato qualche mese fa un'auto elettrica, visto che a casa ho un impianto fotovoltaico connesso in rete con SSP , mi sono messo a studiare un qualcosa per ottimizzare l'autoconsumo dell'energia prodotta .

Per una serie di motivi ho scartato l'uso rele domotici e simili; potenze in gioco alte, impossibilità di modulare la potenza di ricarica, oltre che la mia l'auto identifica come anomalia l'accensione e lo spegnimento dell'EVSE, probabilmente pensa che sia saltata la corrente e quindi alla riaccensione dell'EVSE devo sfilare e reinfilare il connettore di carica.

Prima del progetto vero e proprio preferisco inserire una piccola serie di nozioni propedeutiche allo stesso.

VEICOLI ELETTRICI

Con il migliorare della tecnologia delle batterie e dei semiconduttori di potenza, i prezzi delle auto elettriche stanno diminuendo e le autonomie stanno diventando accettabili.

Vedremo se nel futuro questa tecnologia consentirà di abbattere l'inquinamento, per ora c'è chi dice che costruire le batterie inquina più di un'auto tradizionale in tutta la sua vita, ma a seconda di chi finanzia questi studi i risultati sono contrastanti.

Detto ciò, nel mio piccolo, mi piace l'idea di poter caricare la mia macchina sfruttando solo l'energia del sole.

IL CARICABATTERIE

Il vero caricatore delle batterie, che implementa tutte le funzioni di sicurezza, controlli temperatura, tensione interna delle celle, ecc... è interno all'auto, questo perchè ogni auto è diversa ed il costruttore per poter garantire la vita della batteria gestisce tempi, correnti e modi di ricarica in funzione della configurazione e chimica delle celle.

Il cavo di ricarica quindi altro non è che una spina abbinata ad un sistema che integra una serie di sicurezze per l'utente ed informa il caricatore interno all'auto della potenza massima utilizzabile.

IL CONNETTORE

Il connettore presente sulle auto europee vendute negli ultimi anni si chiama "MENNEKES TYPE 2", ed esiste anche nella versione combo o CCS, che ha dei pin separati per la carica rapida in corrente continua.

Per la carica casalinga, si usa un cavo dotato di presa TYPE 2, lo stesso connettore può supportare sia una rete trifase che monofase e la potenza di ricarica cambia in funzione del tipo di veicolo e della rete dove è connesso.

Il connettore TYPE 2 ha i seguenti pin:

L1 - line 1 - FASE1

L2 - line 2 - FASE2

L3 - line 3 - FASE3

N - neutral - NEUTRO

PE - protective earth - TERRA

PP - presence pilot

CP - control pilot

Lo standard prevede che questo connettore possa operare all'aperto ed in presenza di pioggia, quindi è dotato di una serie di guarnizioni che garantiscono la tenuta ermetica ed un sistema che toglie tensione alle fasi ed al neutro quando lo stesso non è innestato.

E' anche presente un blocco meccanico che impedisce di sfilare il connettore quando l'auto è in ricarica o vi è tensione sulle linee di potenza.

J1772

E' il protocollo con cui il caricabatterie dell'auto dialoga con l'EVSE (Electric Vehicle Service Equipment), la WallBox o l'IC-CPD (In Cable Control and Protection Device), che viene chiamato carichino, il quale permette di caricare l'auto con una comune presa 16A.

Quello che mi interessa del protocollo J1772 è la parte di ricarica AC, in quanto quella veloce in corrente continua non è fattibile a con la rete domestica per via delle potenze necessarie. Le linee richieste dallo standard sono:

J1772 - presence pilot (PP)

Il PIN serve per far capire all'auto la capacità massima in ampere del cavo connesso alla stessa, questo avviene tramite una resistenza posta tra il pin PP e la massa a terra, secondo questa tabella:

13A 1.5Kohm 0.5W (1k a 2.7K)

20A 680ohm 0.5W (330 as 1k)

32A 220ohm 1W (150-330)

63A 100ohm 1W (50-150)

La specifica, anche se prevede che la resistenza debba avere una

tolleranza massima del 3% prevede che l'auto identifichi un range molto ampio per ogni intervallo.

Quindi in base alla resistenza montata nel connettore l'auto non supererà l'assorbimento massimo dato dalla tabella sopra.

J1772 - control pilot (CP)

Questo è il pin dedicato al dialogo tra auto ed EVSE, in quanto il secondo deve sapere quando la macchina è connessa, dichiarare la potenza massima utilizzabile, sapere quando il veicolo è pronto per caricarsi in modo da alimentare le linee di potenza (L1, L2, L3 ed N).

Per fare questo l'EVSE ha all'interno un generatore di onda quadra ad 1kHz \pm 12V con una impedenza di uscita di 1kohm.

In base al duty cycle, l'auto rileva la potenza massima utilizzabile ed indica il suo stato con delle resistenze che caricano l'uscita CP dell'EVSE.

Nel caricatore interno all'auto vi è un primo diodo, in serie al CP, seguito da una resistenza da 2700ohm verso terra, in questo modo , sapendo che l'impedenza di uscita dall'EVSE del CP è di 1000ohm, la tensione da \pm 12V passa a \pm 9V, quindi misurando la tensione nella semionda positiva, l'EVSE capisce che l'auto è collegata.

Nel momento che l'auto richiede energia, viene posta in parallelo alla prima resistenza un'altra resistenza da 1300ohm, in questo modo l'EVSE vede una resistenza equivalente di circa 870ohm e la tensione misurata sul CP dall'EVSE è di \pm 6V, da questo momento l'EVSE ha alcuni secondi per mettere in tensione le linee di potenza e l'auto inizia a caricarsi.

Sono possibili altri stati, rilevabili sempre in base alla resistenza

equivalente vista dall'EVSE, ad esempio "VENTILATION REQUIRED", tensione rilevata +3/-12V ed "ERROR" +0/-12V.

Vi è poi un modo per mettere in pausa la ricarica, da quello che ho visto non è previsto nello standard J1772, anche se in realtà è una conseguenza di come hanno definito gli stati, ma è comunque descritto in un documento interno VW/AUDI che descrive la loro implementazione del protocollo.

In pratica, se il control pilot disattiva l'onda quadra nel semiperiodo positivo l'auto porta la corrente di ricarica a 0A. Questo perchè nella fase STANDBY del protocollo J1772 viene dato "PILOT HIGH VOLTAGE +12V, PILOT LOW VOLTAGE N/A", quindi questa situazione mette in pausa la ricarica.

SICUREZZA

I rischi da prendere in considerazione sono legati al sovraccarico/incendio dell'impianto elettrico o folgorazione.

Sovraccarico

La linea a cui si sollega il carichino o wallbox deve essere protetta da un magnetotermico che tenga conto della sezione e della lunghezza dei cavi, anche perchè a differenza di un utilizzatore elettrico standard, che di norma non assorbe mai la potenza massima di targa, l'auto assorbe quello che l'EVSE 'dichiara'.

C'è da dire che il caricatore interno ha un PFC attivo che regola il cosfi in modo ridurre le perdite sui cavi, ma morsetti e giunzioni fatte male, cavi di sezione piccola, canaline piene possono creare gravi problemi portando anche ad un incendio.

Folgorazione

L'IC-CPD o WALLBOX devono eseguire un test per verificare che linea di terra sia presente e funzionante, il progetto in seguito non fa questa verifica, anche se questa funzione è nella mia ToDo list.

Oltre a questo l'EVSE deve integrare un misuratore di corrente differenziale (salvavita), per misurare eventuali dispersioni verso terra, e deve poter eseguire un test ad ogni attivazione, anche questa funzione non la ho ancora implementata, ma ho montato un interruttore differenziale a monte.

Le linee L1, L2, L3 e N devono essere scollegate fino a quando il connettore non è inserito nel veicolo e l'auto non chiede di caricare, bloccando la presa meccanicamente, questo viene fatto dal mio circuito tramite un teleruttore da 63A che interrompe fase e neutro.

Una ulteriore sicurezza, che è richiesta, è la verifica che il teleruttore non presenti i contatti 'saldati', questo non è ancora implementato, ma lo sarà presto.

Danni al VEICOLO

Leggendo le specifiche del protocollo J1772 e le raccomandazioni, l'implementazione dovrebbe essere molto robusta e resistere anche a segnali con valori di tensione difforni dalle specifiche. Logicamente il livello di paracu..ggine dei produttori è molto elevato ed il fatto di usare un EVSE autocostruito può essere un appiglio per la garanzia, anche se difficile da dimostrare.

Detto questo, sconsiglio a chi non ha le competenze del caso di realizzare un circuito simile e prendere quanto segue a solo scopo didattico.

SCELTA DEL PROCESSORE/SOC

Prima di iniziare lo sviluppo ho studiato il progetto OpenEVSE.

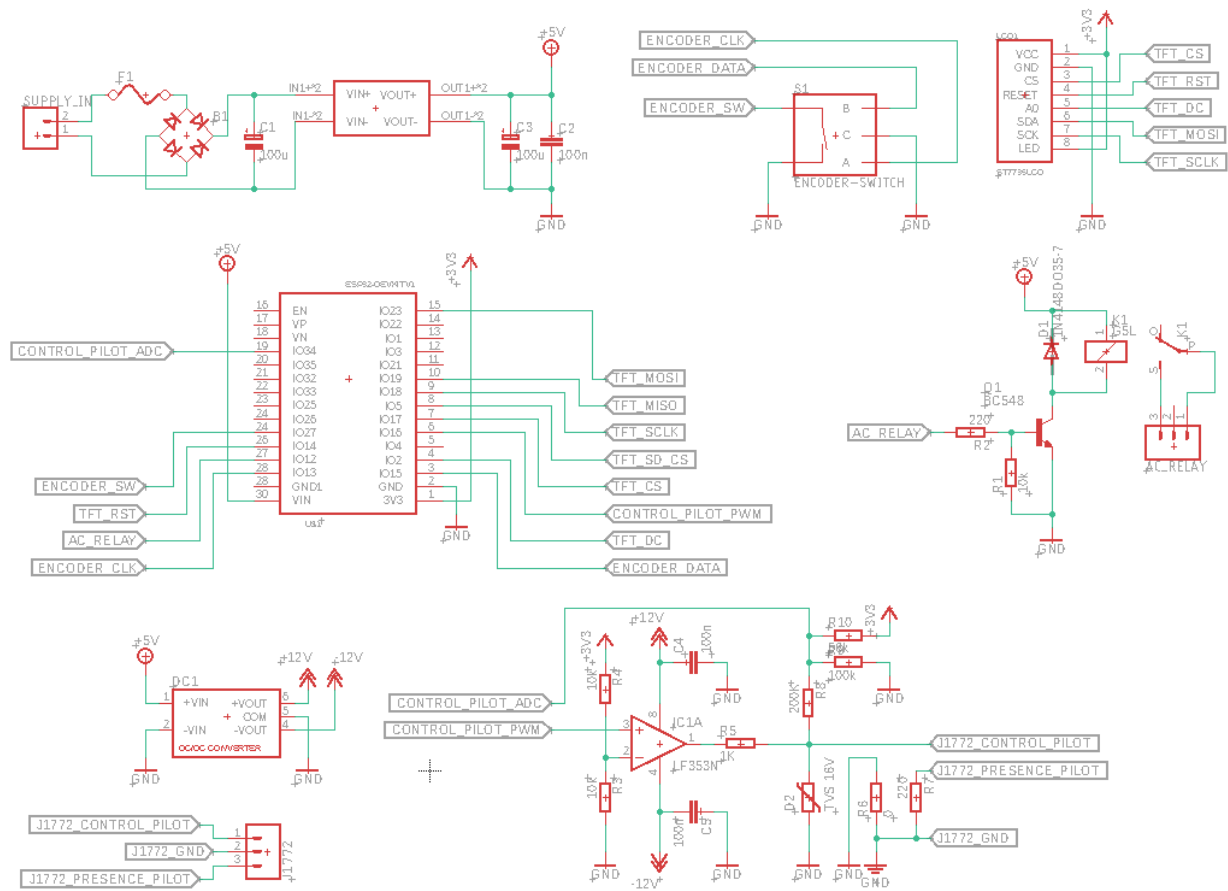
Questo usa arduino ed un modulo wifi aggiuntivo per la connettività, la mia idea è stata di testare lo sviluppo con un ESP32.

ESP32 è un System On Chip, che integra 320K di ram, due core a 160mhz e di norma tutti i moduli lo dotano anche di una flash SPI con una capacità di alcuni mega, utile per caricare le risorse per un pannello di gestione web.

Il micro, include il supporto per WiFi, Bluetooth, 34 gpio, adc, dac, uart, spi, i2c, pwm, ecc....

Il fatto di avere due core permette di avere un core dedicato allo stack di rete ed uno per altre operazioni.

SCHEMA ELETTRICO



schema in formato eagle

Lo schema elettrico è davvero minimale, partendo dall'ingresso d'alimentazione , dopo un ponte diodi (che permette di alimentare sia in AC che in DC), vi è un regolatore 5V buck (modulo MP1854).

ALIMENTAZIONI

La tensione +5V serve per alimentare il rele che servirà ad attivare il teleruttore che attiva l'energia verso il connettore MENNEKES ed il modulo ESP32, il quale al suo interno ha un regolatore low dropout da +3.3V che viene usato per TFT, encoder, ecc... .

Tramite un convertitore DC-DC isolato "DPUN02L-12" dalla +5V ottengo una tensione duale, rispetto alla massa, di +/-12V necessaria

per il control pilot, all'inizio avevo pensato di usare un 7912/7812 ed un trasformatore duale din, ma ho recuperato questo DC-DC che mi ha semplificato la vita.

LCD - ENCODER

Come display all'inizio ho usato un SSD1306 I2C, ma era troppo piccolo ed illeggibile, per cui ho optato per un display SPI a colori da 1.8" basato sul controller ST7735, sul display è presente anche uno slot SD, che ho cablato per dei test.

L'encoder è il classico encoder rotativo, con uno switch, ho evitato le resistenze di pull ed il filtro antirimbalo, in quando via software ho impostato i pin come ingressi con pullup e la libreria che uso gestisce l'antirimbalo via software.

CONTROL PILOT

Il segnale del control pilot in uscita dall'ESP32 viene traslato da 0-3.3V a -12/+12V tramite l'LF353, questo operazionale ha uno slew rate molto elevato e permette dei fronti di salita ripidi, come richiesto dalla specifiche J1772.

L'uscita dell'operazionale ha in serie una resistenza da 1000ohm, ed un TVS +/-16V come protezione (da specifiche J1772).

In parallelo all'uscita del control pilot vi è una rete di condizionamento per adattare i valori di tensione -/+ 12 a 0-3.3V per campionarli con l'ADC dell'ESP32 in modo da interpretare lo stato del caricatore interno all'autoveicolo.

AC RELAY

Per pilotare il teleruttore doppio da 64A/230V uso un rele da 5V, che pilota la bobina del teleruttore, il pilotaggio avviene con un classico BJT NPN ad emettitore comune ed un diodo in anti parallelo alla bobina.

CONNETTORE J1772

Alla mille fori sono connessi, terra, presence pilot, che va verso massa con una resistenza da 220 (vedi tabella sopra) e control pilot.

NOTE: L'alimentatore deve essere ISOLATO GALVANICAMENTE DALLA RETE, il negativo dell'EVSE, la TERRA del J1772, la TERRA dell'impianto elettrico vanno collegati tramite un morsetto equipotenziale!!!

Le sezioni dei cavi di potenza e tra connettore J1772 devono essere di almeno di 6mmq.

MISURATORE DI POTENZA

Per misurare la potenza istantanea del fotovoltaico, rete e consumo della casa, ho scelto di usare uno SHELLY EM, questo modulo è dotato di due trasformatori di corrente induttivi a doppia C, questo permette di misurare la corrente in un cavo AC senza doverlo scollegare, in questo modo si evita di modificare l'impianto elettrico di casa.

Dei due sensori, uno è posizionato sul cavo che dal contatore va verso il quadro di casa, ed il secondo sul cavo che arriva dall'inverter del fotovoltaico, in questo modo facendo la somma algebrica tra potenza immessa e consumata posso ricavare la potenza prelevata dalla rete.

Il sensore viene alimentato da una comune presa che ho di fianco al contatore (che arriva dal quadro, e quindi protetta da salvavita).

		
Schema shelly EM	Sensori montati (notare il cavo del 1960)	Pannello web interno dello shelly

EVSE PROFILI DI FUNZIONAMENTO

Il mio EVSE supporta 4 modi o profili di funzionamento, impostati di default nel seguente modo:

PAUSA (non carica)

ECO: potenza massima di carica 1500W, cercando di tenere la potenza usata dalla rete a 0W

SOLAR: come eco ma, modulando la potenza massima fino a 3500W

SLOW: potenza massima di carica 1500W, cercando di tenere la potenza usata dalla rete sotto i 3000W

FAST: come SLOW, ma modulando la potenza di carica fino a 3500W

Per ogni profilo i parametrai configurabili anche via WEB sono i seguenti:

EVSE Power Limit: potenza massima usabile dall'EVSE

GRID Power Limit: potenza massima prelevabile dalla rete

Hysteresys ON: nel caso l'EVSE vada in pausa, per superamento del "GRID Power Limit" o per evitare il supramento della potenza massima del contatore, attende questo valore in secondi prima di

riattivarsi.

Hysteresys OFF: nel caso l'EVSE rilevi una potenza usata superiore al "GRID Power Limit" ed anche riducendo la potenza, non riesca a diminuire l'assorbimento, aspetta questo valore in secondi prima di andare in pausa. Questo serve per evitare che picchi di consumi di alcuni elettrodomestici mandino in pausa l'EVSE.

My EVSE

HomeProfilesSettingsSystem

EVSE Profiles

Default BOOT

LAST

ECO evse power limit

1500

ECO grid power limit

-100

ECO hysteresis ON

180

ECO hysteresis OFF

600

SOLAR evse power limit

3500

SOLAR grid power limit

-100

SOLAR hysteresis ON

180

SOLAR hysteresis OFF

600

SLOW evse power limit

1500

SLOW grid power limit

2500

SLOW hysteresis ON

180

SLOW hysteresis OFF

3600

FAST evse power limit

3500

FAST grid power limit

2900

FAST hysteresis ON

180

FAST hysteresis OFF

3600

Submit

Modi di funzionamento

My EVSE

HomeProfilesSettingsSystem

EVSE Settings

WiFi SSID

ESP

WiFi PASS

ESP

GRID CAPACITY

3000

EVSE CAPACITY

3500

NTP POOL

pool.ntp.org

NTP GMT OFFSET

3600

NTP DAYLIGHT OFFSET

3600

SYSLOG

192.168.6.158

Power Meter URL

http://192.168.6.108/status

Power Meter TYPE

SHELLY2EM

Submit

Configurazioni dell'EVSE

Oltre ad i parametri dei modi di funzionamento, via interfaccia web è possibile settare:

GRID CAPACITY: La potenza massima nominale della propria fornitura elettrica, il sistema integra le tolleranze contrattuali e con queste gestisce le politiche di sovraccarico:

fino +10% potenza nominale non è sovraccarico.

tra +10% e +30% è sovraccarico ed attende alcuni secondi prima di mandare in pausa l'EVSE, ed il contatore staccherà la fornitura entro 180 minuti.

oltre 30% mette in pausa immediatamente l'EVSE.

Altri paramatri sono relativi alla configurazione di rete, syslog server (che raccoglie i log per il funzionamento ed il debug), ntp per l'ora, shelly per il monitoraggio della potenza.

IL FIRMWARE

IL FIRMWARE lo considero ALPHA, la macchina a stati va rivista, è funzionante, ma andrebbe ripulito da vari scenari di test e debug.

I sorgenti sono disponibili su github: <https://github.com/ig0rb/ESP32-EVSE>

Per lo sviluppo ho usato VSCode e PlatformIO, ed è scritto in C++.

La scelta dei GPIO da usare per le varie funzioni è legata al fatto che alcuni PIN 'flappano durante il boot' ed altri cambiano il modo di avvio, ho seguito questa utilissima guida:

"ESP32 Pinout Reference: Which GPIO pins should you use?"

Per la configurazione di default è possibile modificare il file:

`include/def_config.h`

oppure, crearne una copia chiamandola `my_config.h`, quest'ultimo file

è nel .gitignore e sarà possibile scaricare la nuova versione dal repository senza perdere le proprie impostazioni.

Partendo dal classico main.c, troviamo la funzione setup() che si occupa di inizializzerà i vari oggetti, e loop() che eseguirà l'handle.

Ho voluto evitare di usare variabili globali, per cui le classi di configurazione, stato, J1772 sono di tipo singleton.

Il singleton è un design pattern, che permette ad una classe di instanziare UN SOLO oggetto, in quanto il costruttore è un metodo privato, ed ogni volta che verrà richiamata con il metodo getInstance() tornerò il puntatore dell'oggetto istanziato la prima volta.

Questo è comodo in quanto molte funzioni asincrone hanno la necessità di modificare o leggere dei valori di stato dell'EVSE.

Spesso le funzioni asincrone accettano un solo parametro che è un puntatore void, per passare parametri alle stesse sarebbe necessario creare una struttura o classe in cui 'incapsulare i vari parametri e fare il casting del puntatore, oppure abusare l'uso delle variabili globali, riducendo la leggibilità e riusabilità del codice.

Per creare una classe singleton si deve definire un attributo puntatore static alla classe stessa, un metodo statico getInstance() che richiama il costruttore privato della classe stessa, se il puntatore instance è nullo, il metodo statico getInstance() richiama con new il costruttore, che crea l'oggetto e lo salva nell'attributo instance.

*** snippet di esempio del pattern singleton nella classe config ***

```
Config* Config::instance = 0;
Config* Config::getInstance() {
    if (instance == 0) {
        instance = new Config();
    }
}
```

```
    return instance;
}
```

```
*****
*****
```

Un'altra funzione interessante è quella di poter gestire dei task, invece che nel loop() tramite le API fornite da FreeRTOS, presenti nell'SDK Arduino-ESP.

Ad esempio per il polling dello SHELLEY, la classe PowerMonitor.cpp, nel metodo begin richiama:

```
xTaskCreatePinnedToCore(PowerMonitor::_begin, "PowerMonitor",
8192, NULL, 0, NULL, 0);
```

Questa API come primo parametro necessita del puntatore ad una funzione (in questo caso un metodo statico _begin della classe stessa, il nome del task, la dimensione dello stack da allocare, il puntatore ai parametri da passare, la priorità, l'eventuale puntatore al TaskHandler (per killarlo, ecc...), ed il core nel quale il task deve essere eseguito.

Nel mio caso il metodo statico _begin() è un loop infinito che viene eseguito ogni 5 secondi.

A differenza dei delay usati nel main loop od altri metodi per temporizzare l'esecuzione la API per mettere in pausa il TASK, e lasciare allo scheduler eseguire altre operazioni è la seguente:

```
void vTaskDelay( const TickType_t xTicksToDelay );
```

nel mio caso per aspettare 5 secondi:

```
vTaskDelay(5000 / portTICK_PERIOD_MS);
```

ossia valore in millisecondi diviso tick_ms dello scheduler, senza scendere troppo nel dettaglio tick è il 'quanto di tempo o la cadenza con cui lo scheduler gestisce il tempo.

FreeRTOS è un RealTime Operating System, non tutte le API sono disponibili con l'SDK Arduino-ESP in quanto la libreria FreeRTOS è

precompilata ed inclusa come static library (freertos.a), ma le principali funzioni sono utilizzabili.

La classe UI.cpp contiene la gestione del display a colori TFT SPI e l'encoder.

Per l'encoder ho usato una classe comodissima che si chiama "spacehuhn/SimpleButton" che ha una ottima gestione dell'encoder senza usare gli interrupt.

Relativamente al TFT ho invece usato "bodmer/TFT_eSPI", l'unica accortezza è che per far funzionare questa classe o si modificano gli headers della libreria, cosa molto scomoda se la li integra come dipendenza in PlatformIO (o Arduino IDE) oppure di passano i parametri del display nei build_flags tramite il file plaformio.ini.

A differenza dei display monocromatici un display a colori deve trasferire davvero molti dati per ogni singolo refresh, quindi per evitare il flickering si deve aggiornare SOLO la parte di display che contiene cambiamenti, andando ad impostare colore principale e di sfondo, per sovrascrivere l'area, evitando funzioni come clear() o fill().

In molti punti del codice, per eseguire una parte di istruzioni ogni X secondi o millisecondi uso due metodi:

- a) salvo in una variabile il timestamp del nextrun (con millis() + tempo) ed in ogni richiamo del metodo handle() verifico se millis() attuale è maggiore di nextrun, ed in caso negativo ritorno.
- b) verifico se millis() - last_run è maggiore del mio periodo di esecuzione.

millis() e micros() sono dei contatori, forniti dalle api arduino che ritornano il valore in uS o mS dall'avvio.

Le icone del menu le ho create tramite image2cpp e salvate in delle variabili in un header.

J1772.cpp è invece la classe per la gestione del 'protocollo' con il veicolo.

La parte interessante è la lettura dell'ampiezza della semionda positiva, siccome so che ad i 1khz il periodo è 1ms, campiono per 1.25ms il pin control_pilot e salvo il valore minimo e massimo dell'adc, il valore massimo è l'ampiezza del semiperiodo positivo. L'ADC impiega alcuni uS a campionare il valore e sapendo che il duty cycle è del 10% sono sicuro di avere abbastanza campioni per avere una stima corretta del valore.

NOTA: l'ESP32 ha due ADC che possono essere attestati su vari PIN, per una serie di motivi legati all'hardware interno e/o SDK del produttore, solo ADC1 può essere usato con il WiFi attivo.

Tra l'altro ho notato che a volte lo stack di rete interferisce con la lettura, ma potrebbe essere dovuto all'elevata impedenza della rete di condizionamento, che viene influenzata dall'antenna WiFi, avendo fatto tutto sulla mille fori, senza un buon piano di massa, la lunghezza dei cablaggi è paragonabile alla lambda del wifi e questo potrebbe introdurre rumore.

La generazione del PWM è abbastanza banale e la state machine è visibile nel codice.

PowerMonitor.cpp è invece la classe con cui viene fatto il polling dello SHELly, come detto sopra gira in un task sul core0 usato per le funzioni di rete.

SHELly restituisce un JSON e per il parsing ho usato "bblanchon/ArduinoJson", l'autore della libreria fornisce un ottimo assistant online per l'imbastitura del parser: "<https://arduinojson.org/v6/assistant/>".

Un'altra piccola accortezza relativamente al PowerMonitor è la creazione di una struttura per l'implementazione di una coda di N

campioni per poter calcolare delle medie sugli ultimi X campioni, utile per adattare la potenza allocabile all'EVSE.

Nella struttura viene anche salvato il valore di millis() dell'ultima lettura valida e nel caso lo SHELLY non risponda per più di X secondi l'EVSE va in pausa.

PowerManager.cpp si occupa invece di verificare la potenza disponibile ed attivare l'EVSE se questa è conforme al profilo selezionato, modulando la potenza o rimettendolo in pausa, oltre che prevenendo il distacco per superamento di potenza del contatore.

WebServer.cpp contiene l'implementazione del webserver asincrono per il pannello di configurazione.

Alcuni url tornano un JSON, altri eseguono comandi, come reset o reboot.

Per ragioni di sviluppo ho disattivato le protezione di cross-site-scripting, in modo da poter avere le pagine web sul mio pc e richiamando e postando JSON sull'ip dell'ESP.

Nel codice si possono identificare i vari url, tra cui:

/mode/[nome modo]

/reboot (fa il reboot)

/reset (resetta alle impostazioni di default)

/system (ritorna memoria liber ed altre info)

/ui/[up|down|enter] (per pilotare l'encoder da web)

/config [GET|POST] ritorna un json della configurazione per popolare la UI web o parsa il POST della UI salvando le modifiche alla configurazione.

/index.html (single page app della userinterface web)

MyLog.cpp è una classe per la gestione dei log, che integra la libreria arcao/Syslog che vengono inviati via protocollo syslog al mio

pc per debuggare e monitorare il sistema, buona parte è fatta con delle macro:

```
#define I_debug(fmt, ...) MyLog::write("[D][%s:%d] %s(): " fmt, __FILE__, __LINE__, __func__, ##__VA_ARGS__);
```

Queste richiamano il metodo statico `MyLog::write(const char*, ...)` che verifica sempre che la rete sia attiva prima di inviare il pacchetto UDP (prevendo il crash dell'ESP).

In questo modo il debug o log contiene il nome del file sorgente, la riga, la funzione o metodo in cui è stato richiamato.

L'implementazione syslog usata è ques

OTA.cpp permette l'UPDATE del firmware e della flash SPI via rete, visto che adesso l'EVSE è in BOX se voglio caricare un update posso farlo da remoto dalla mia scrivania :P

C'è poco da dire, è usata come da manuale ed in platformio.ini basta specificare:

```
upload_protocol = espota  
upload_port = IP_DELL_ESP
```

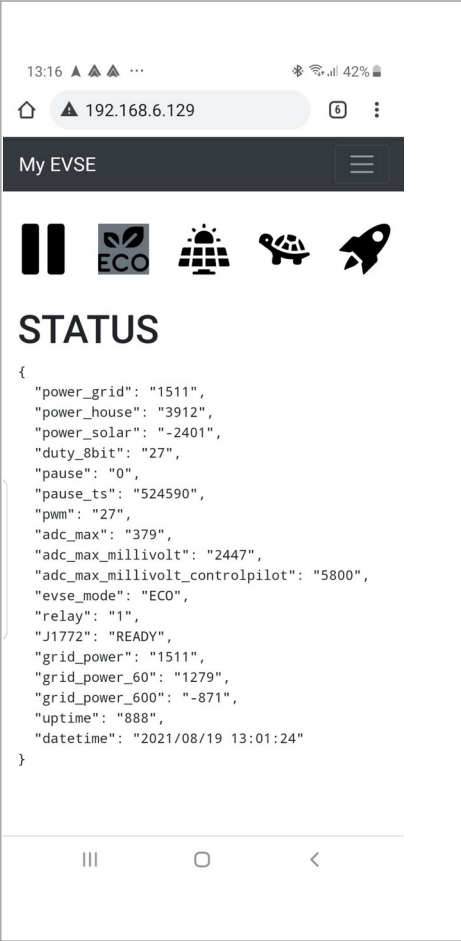
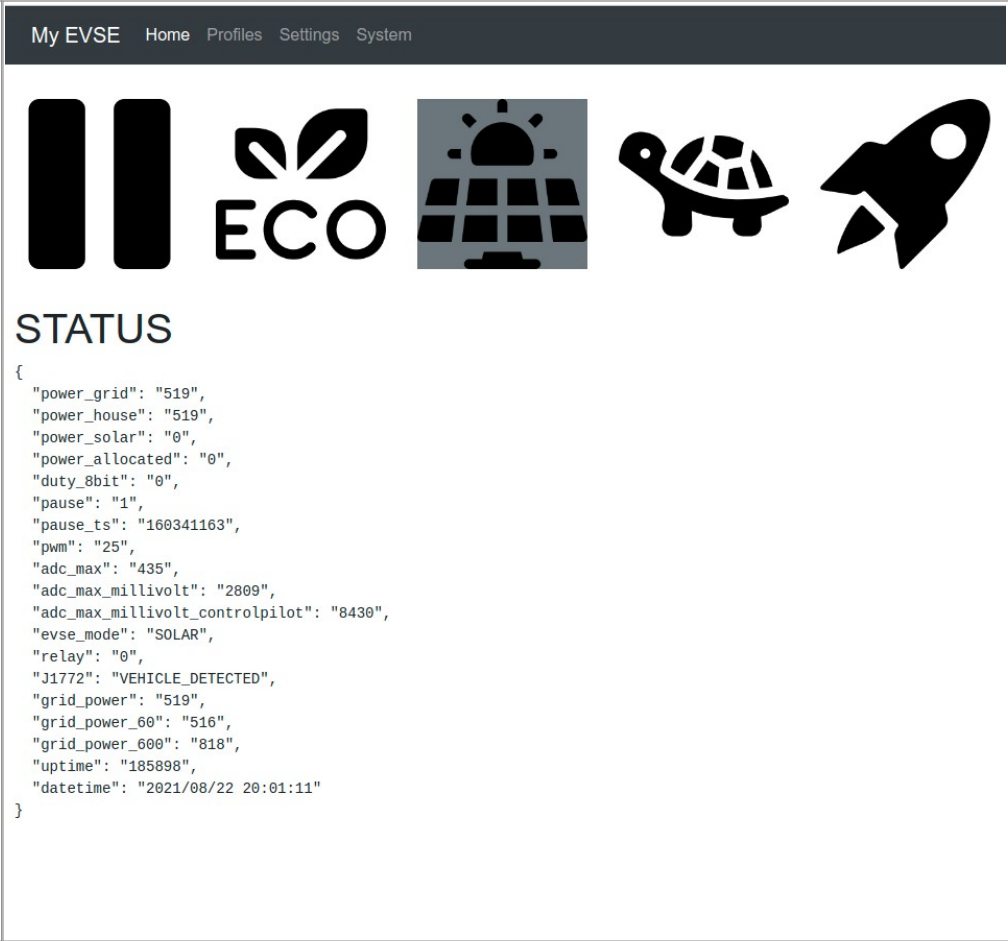
WEB SINGLE PAGE APP

Per evitare di caricare l'ESP32 con un postprocessor, l'interfaccia web è in html e javascript, senza usare templates.

Ho usato bootstrap 4 e jquery, utilizzando le CDN per evitare di riempire la flash dell'ESP, anche se le versioni minified ci starebbero senza problemi.

Al caricamento della pagina i dati dei form vengono popolati richiamando un url che ritorna un JSON, ed ogni tot secondi vengono refreshati gli altri dati dei tab visualizzati.

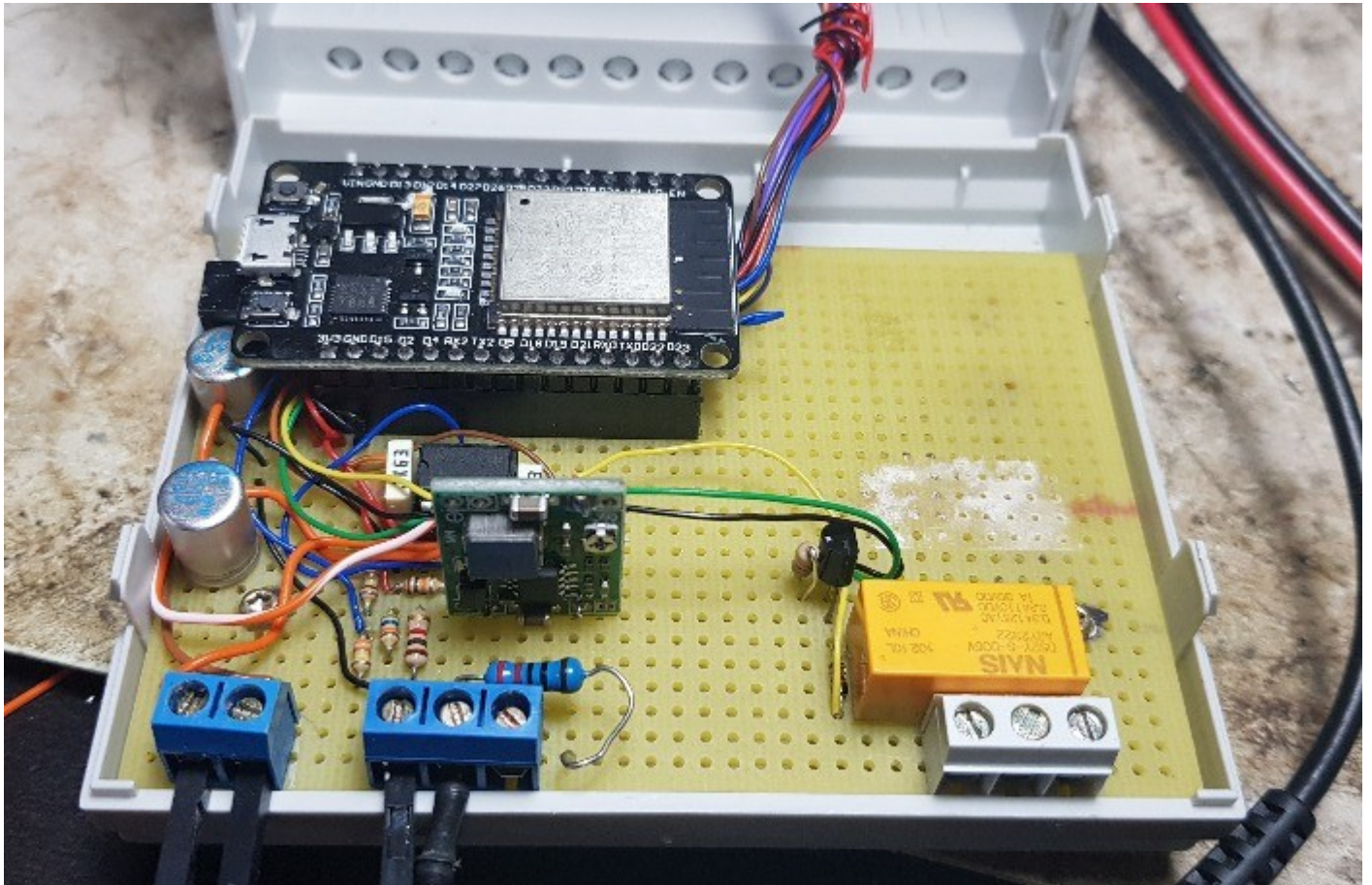
Al submit dei form uno script js si occupa di prevenire il post ed il refresh della pagina ed invia i dati come post all'ESP che aggiorna e salva la configurazione.

	
web UI su mobile	web UI su desktop

ASSEMBLAGGIO

Ho usato una scatola DIN 6 moduli della combiplast, che presi in qualche fiera anni fa, il frontalino con TFT ed LCD è stato disegnato con Fusion 360 e stampato al volo

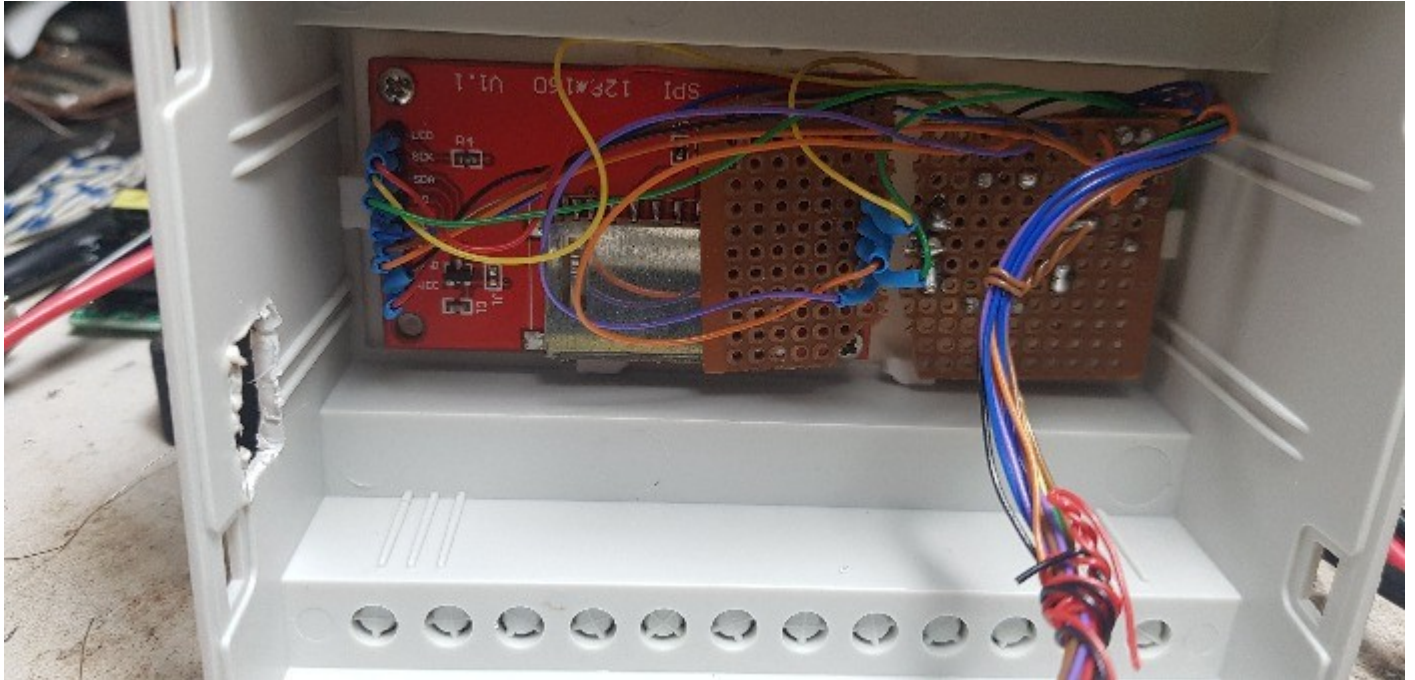
[Link al file fusion online.](#)



Nella foto si notano, il DC-DC sotto l'ESP32, i condensatori elettrolitici, il ponte diodi è sotto di tipo smd, il modulo buck tarato a 5V, il rele che piloterà il teleruttore, la resistenza da 1W del presence pilot, nella foto manca il TVS che ho poi montato prima di assemblare il tutto.

Siccome il rele da il neutro al teleruttore è in posizione distanziata dalla parte in bassa tensione, tra l'altro ho eliminato tutte la piazzole intorno allo stesso in modo

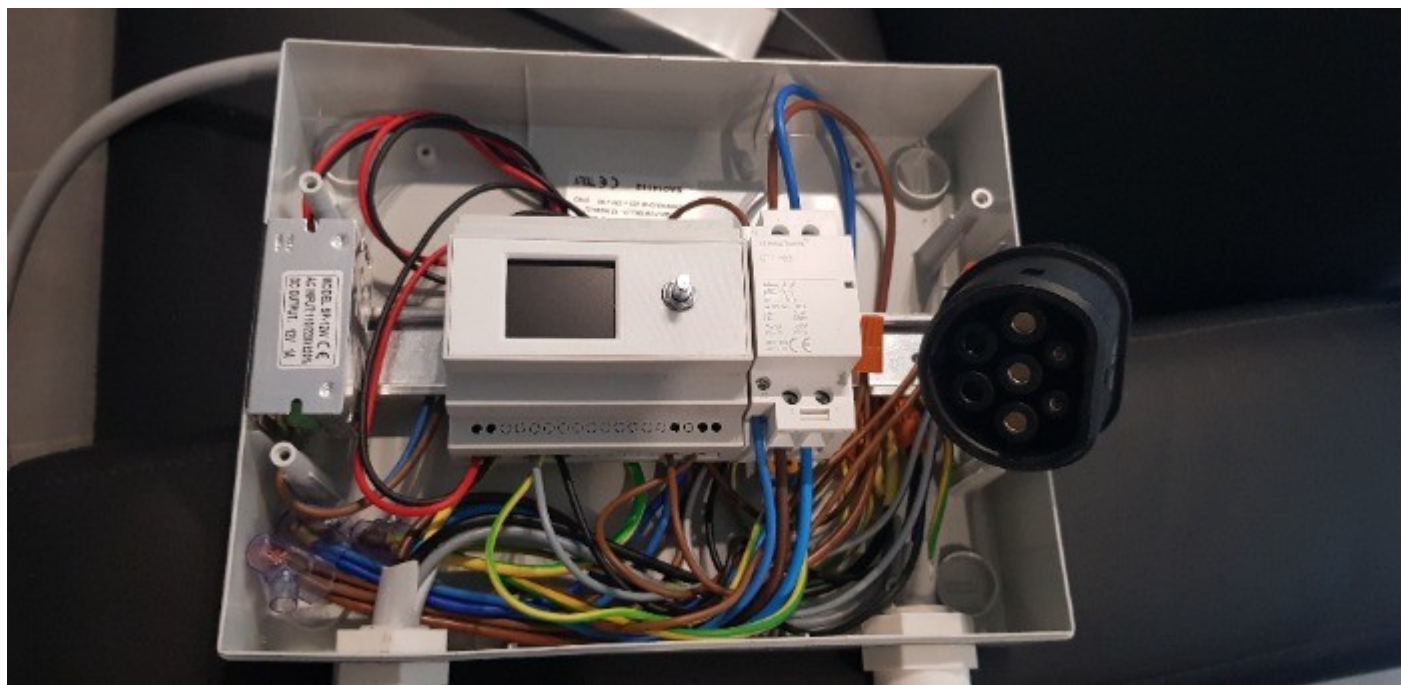
da garantire un'adequata distanza di isolamento e prevenire archi.



Un dettaglio del TFT, dell'encoder e del buco fatto con il saldatore per collegare l'USB all'ESP32.



L'EVSE attivo, mentre viene testato., si nota l'IP, il segnale WiFi, i modi di funzionamento, e la potenza allocata di 0 watt.



Il modulo AC-DC isolato, l'EVSE, il teleruttore da 64A, il connettore TYPE2, tutto montato in una quadro DIN da 12 moduli.

Anche i fermacavo sono stampati, visto che erano finiti in negozio, di questi non condivido i files in quanto non sono molto precisi i filetti e mi hanno fatto perdere parecchio tempo per adattarli.



Ecco l'EVSE in funzione, per ora è collegato con una spina 16A, in box arriva già un FG7 3x6mmq, che feci mettere quando sistemai la casa, a breve farò mettere una presa interbloccata da 32A, che sicuramente è più sicura di una comune schuko.



Connettore innestato, manca ancora il guscio da fissare, in quando una volta infilato sarà impossibile da togliere, per cui voglio essere sicuro che funzioni tutto bene, nella foto si notano una termo-restringete

verde ed una blu, servono ad adattare 2 fili da 6mmq del cavo da 1.5mmq per innestarli nel control-pilot e

ToDo:

Implementare tutte le sicurezze di un EVSE commerciale:

- *Controllo welded rele, tramite un foto-accoppiatore su fase neutro in uscita dal teleruttore.
- *Controllo presenza massa a terra, tramite un fotoaccoppiatore si testa il passaggio di 2ma tra terra e fase e terra e neutro.
- *Controllo e test dispersione verso terra AC/DC tramite un trasfomratore di corrente stile OpenEVSE.
- *Misura della potenza usata dall'EVSE.
- *Miglioramento PowerManager con PID
- *Implementazione MQTT
- *Miglioramento UI WEB e TFT

Link Utili:

<https://www.openevse.com/>

<https://www.ti.com/lit/ug/tiduer6/tiduer6.pdf?ts=1629675324100>