Report di tirocinio

Gabriele Monaco

26 maggio 2018

Indice

1 GESTIONE NEGOZIO CON STM32F429ZI

1.0.1 Introduzione

Con questo firmware si vuole implementare una parte del progetto *Alterline*, che mira a fornire un controllo remoto delle colorazioni delle vetrine di un negozio, con possibilità di essere esteso con nuove funzionalità e di essere il più possibile automatizzato nella diagnosi degli errori. All'interno del negozio ogni vetrina avrà un led RGB, controllato con PWM da una schedina STM32, ognuna di queste schede sarà in contatto bluetooth con una scheda centrale (per cui questo firmware è pensato), unica per ogni negozio, che sarà a sua volta connessa via ethernet ad un server web, contenente il database utilizzato nel progetto.

1.0.2 Ambiente di sviluppo

Il progetto è portato avanti sull'IDE `Atollic TrueStudio for STM32`, partendo da un progetto generato dal software `STM32CubeMX`, per includere tutte le estensioni Driver e Middelware fornite da ST e poterle configurare in un'unica schermata. Per il controllo delle versioni si ricorre a `git`, direttamente accessibile da TrueStudio attraverso l'apposita estensione di Eclipse (su cui si basa il tool), è così possibile avere una traccia puntuale dei progressi (attraverso lo storico dei commit) ed è possibile creare più varianti di test del progetto (attraverso i branch).
Il debug può essere portato avanto con gdb sempre su Atollic, nella Perspective di debug appare anche una finestra con la console SWV dove, se abilitato tramite la flag `DEBUG` in `includes/debug.h`, appaiono i log lanciati con PRINTF. In alternativa può essere utilizzato il software `STMStudio` per tenere traccia del valore delle variabili durante l'esecuzione (non è possibile usare entrambi i metodi contemporaneamente).

1.0.3 Multithreading con FreeRTOS

Il progetto implementa le due connettività (ethernet e bluetooth) in thread separate, all'interno di queste viene eseguita un'azione tra INIT, READ, WRITE o END a seconda del valore della variabile `*_state` (presente sia per eth che per ble), che permette di eseguire un operazione per ogni ciclo e passare all'attività successiva al termine di ciascuna. Una terza thread può innescare ciascuna trasmissione settando lo stato come INIT (ora viene gestito alla pressione del pulsante).

1.0.4 Implementazione del database locale

files *Src/db_struct.c*, *Inc/db_struct.h*
Il database interno, per quanto riguarda la comunicazione con il server web, contiene due `struct`, rispettivamente per i dati del negozio e quelli delle vetrine (array), entrambi hanno 2 istanze statiche e 2 puntatori, `shadow_` e `active_` (che puntano rispettivamente a una delle due versioni) durante il fetch i dati vengono salvati all'interno delle strutture puntate da `shadow_` e se i dati sono coerenti i puntatori `shadow_` e `active_` vengono scambiati. Alla ricezione della linea riguardante il negozio, vengono controllati l'id e la versione, se il primo coincide e la seconda è maggiore di quella salvata localmente, il fetch delle vetrine prosegue. Ogni riga (negozio e vetrina) contiene un checksum calcolato con l'algoritmo BSD su 2 byte, se la sua verifica fallisce il fetch è interrotto.
I dati ricevuti dal server web vengono trasferiti dalla struttura `active_` alle rispettive vetrine, lì saranno gestiti in una tabella pronti per essere attivati in base alla data. Ora è possibile la lettura dei dati effettivamente attivi su ciascuna vetrina, che verranno poi salvati in una tabella specifica, contenente anche dati relativi al negozio (`running_shop` e `running_entries`). Quando tutti i dati sono pronti è possibile spedirli al server. Anche lì verrà controllato il checksum dei dati ricevuti e nel responso sarà utilizzato il codice 206 (al posto di 200) per chiedere una ritrasmissione, discorso

analogo quando si verifica un errore nel database, qui il codice sarà 205. Nel caso di `shop_id` errato viene restituito 404 (NOT FOUND) e la scrittura si blocca.
Le tabelle (array di tipo `struct vetrina`), contengono i dati nell'ordine in cui vengono ricevuti dal server (non necessariamente ordinati) senza lasciare spazi vuoti, viene quindi utilizzata la funzione `findEntry(id)` che restituisce l'indice corrispondente alla vetrina con id passato come parametro, se presente. Per come è costruita, se viene raggiunto un id nullo la tabella è giunta al termine. La scrittura dei dati ricevuti dai dispositivi ble segue l'ordinamento imposto dal server (usando quindi findEntries che tiene come riferimento `active_entries`), così facendo se una vetrina presente sul server non può essere raggiunta, la sua riga nella tabella `running_entries` viene segnata con id=0.
Le funzioni di trasferimento hanno nomi del tipo `[get/set][Shop/Entry][Eth/Ble]`, dove vengono indicati rispettivamente se devono leggere o scrivere dati, per quale struttura e da che interfaccia, in generale servono per decodificare i dati ricevuti o preparare i dati da inviare.

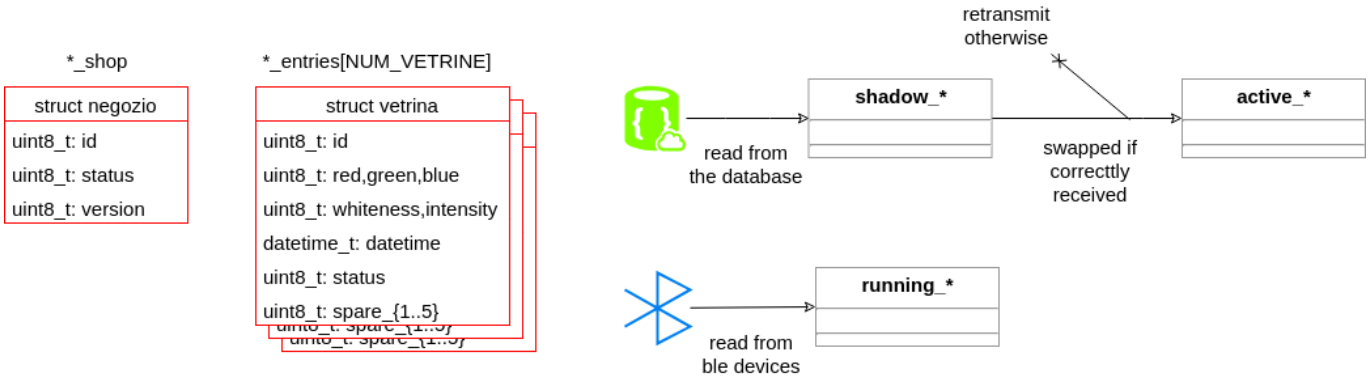

Figura 1: Struttura del database locale

1.0.5 Database remoto e interfaccia web

Il database remoto viene gestito su server apache (o qualunque altra alternativa che supporti php e mysql), nella cartella `server_files` sono contenute le pagine che devono stare nella cartella pubblica sul server e gli script di creazione del database. `index.php` non è nient'altro che il form iniziale per accedere (lato utente) alla visualizzazione grafica delle tabelle, che avviene su `control.php`. All'interno di `fetchData.php` vengono gestite tutte le richieste di lettura dalla board mentre i file `write*.php` gestiscono rispettivamente la scrittura su server delle vetrine e dei dati del negozio.
Lo script `create_table.sql` contiene le istruzioni per definire le due tabelle (vetrine e negozi), collegate da una foreign key che è l'id del negozio. Sono presenti anche tabelle `feedback_*` con la stessa struttura di vetrine e shops, che conterranno i dati attivi letti dalle board.
Le tabelle vengono popolate con alcuni dati iniziali e viene creato un utente per gestire l'unico negozio inserito, al momento per il negozio 1, l'utente è `shop1` con `password1` come chiave (da cambiare), è importante però che dal nome utente sia deducibile l'id del negozio (per come è ora deve essere l'unico numero all'interno del nome). Per evitare di permettere a ciascun utente di poter accedere all'intero database, si possono creare delle VIEW sulle tabelle, in modo da limitare la visione e gestione dell'utente i alle sole righe con i come `shop_id` (l'amministratore non avrà queste limitazioni).
Per poter aggiornare i dati delle tabelle o del negozio da questa scheda su server, è necessario che le righe in tabella siano già presenti, per questa ragione, ogni volta che verrà inserita un vetrina o un negozio si dovrà inserire anche nella tabella `feedback_*` (o quantomeno inserire una riga contenente gli id corretti). Questo capita perché l'inserimento è consentito solamente all'utente root (l'amministratore), mentre gli utenti specifici di ogni negozio possono soltanto eseguire UPDATE su dati già esistenti. Per ogni nuovo negozio vanno inoltre creati l'utente e tutte le rispettive VIEWS (4 in totale). (**TODO** includere queste istruzioni in script specifici)

All'interno degli script php per la gestione del database sono utilizzati dei prepared statement al posto di istruzioni tradizionali, con la spesa di qualche riga di codice in più, vengono prodotte istruzioni generali, alla quale vengono sostituiti i dati come variabili al momento dell'esecuzione. Questo in genere migliora notevolmente le prestazioni al livello del database, in quanto non è necessario un nuovo parsing ad ogni istruzione che differisce solo per valori numerici dalle precedenti (da valutare l'impatto reale su questa applicazione). Inoltre questo metodo protegge ulteriormente da sql injection in quanto i dati inseriti sono sempre numerici (non possono essere inseriti comandi maligni).

1.0.6 Stati di esecuzione

files *Src/main.c*, *Inc/main.h*

Dopo le inizializzazioni di hardware e dati (ancora da definire) il sistema entra in funzione e periodicamente aggiorna i dati (ora questo avviene con la pressione del pulsante). Prima, se necessario (**TODO** da definire) la data viene sincronizzata (*eth*), successivamente parte il fetch dei dati dal server (*eth*), prima i dati del negozio e poi delle vetrine. Questo processo viene saltato qualora la versione locale sia aggiornata, quando questo arriva al termine parte la scrittura dei nuovi dati su ciascuna vetrina (*ble*), successivamente si può procedere con la lettura dei dati attivi (*ble*) e la risposta al server con questi dati (*eth*), (si può mettere in parallelo la comunicazione in entrata e in uscita dal server purché non si utilizzi contemporaneamente la stessa interfaccia, *ble* o *eth*, quando finisce dunque la lettura da entrambi i lati si può passare alla scrittura). Le definizioni di questi processi avvengono nelle funzioni `[BLE,ETH]_Process()`, quando lo stato diventa INIT comincia la connessione, ad ogni esecuzione viene eseguita l'operazione indicata (READ/WRITE/SYNC) e se questa va a buon fine, si passa alla successiva, altrimenti in base all problema riscontrato si ripete il procedimento o si termina il servizio (stato END). (**TODO** implementare una trasmissione delle informazioni sugli errori al server utilizzando lo stato in `running_shop`).

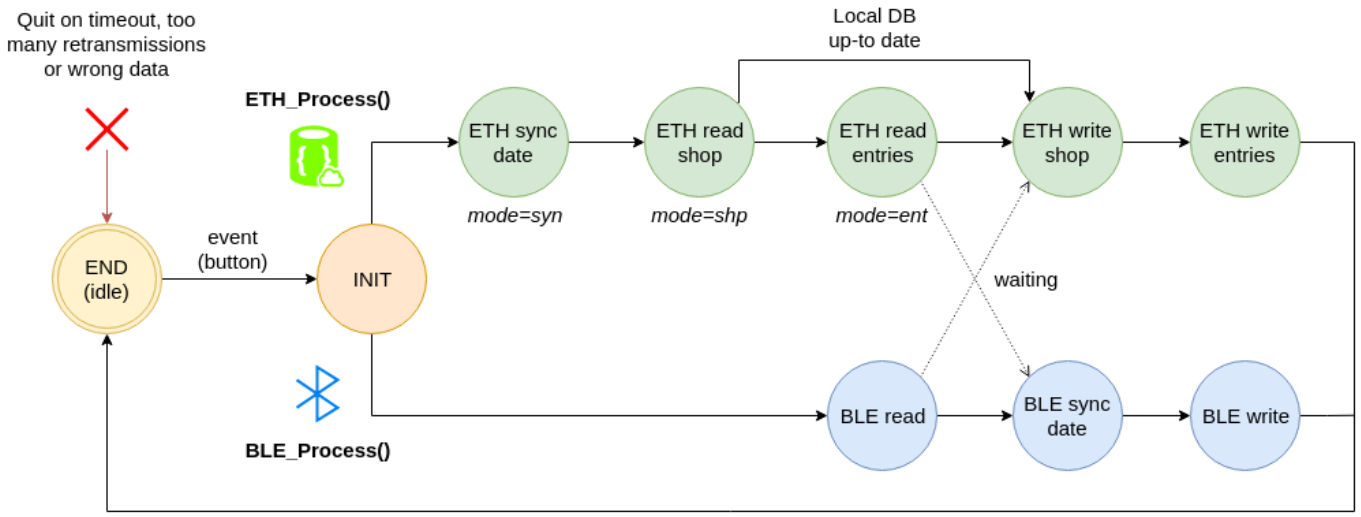

Figura 2: Schema della macchina a stati

1.1 COLLEGAMENTO BLUETOOTH STM32F429ZI

1.1.1 Comunicazione come client BLE

funzione *BLE_Process* e files *Src/sensor_service.c*, *Inc/sensor_service.h*

La connessione bluetooth tra master (negozio) e slaves (vetrine) viene gestita utilizzando il primo come client BLE, che richiede letture e scritture sui servizi, detti caratteristiche, forniti dai server (colori, intensità, spare...). L'interazione con ciascuna caratteristica viene fatta in maniera separata

ad ogni esecuzione della thread, per fare sì che ogni pacchetto sia correttamente ricevuto e gestito. Si interagisce con una vetrina alla volta, vengono dunque eseguite le operazioni su una vetrina, lo stato viene impostato a NEXT e viene chiusa la vecchia connessione per aprirne una nuova con la vetrina successiva. Al termine, per tutte le vetrine, di un'operazione (r/w o sync) si ricomincia dalla prima vetrina con l'operazione seguente. La vetrina che dev'essere utilizzata viene decisa dalle funzioni `nextEntry()` e `currEntry()` in `db_struct`, che attraverso un indice globale, restituiscono progressivamente gli id delle vetrine presenti in `active_table` oppure 0 quando si è giunti al termine (e l'indice viene ri-inizializzato).
Le caratteristiche numeriche vengono convertite in formato testuale, 3 cifre come caratteri ascii per tutte eccetto la data che ne occupa 10 in formato yymmddhhmm. (**TODO** possibile ottimizzazione accorpando alcune caratteristiche e gestendo più vetrine in parallelo, se consentito dagli standard)
Durante la lettura/scrittura delle vetrine, vengono usati due array (`toRead` e `toWrite`) che contengono gli id dei valori da leggere/scrivere (anche utilizzati come indici per nell'array degli handle), le variabili `reading` e `writitng` vengono incrementate ad ogni passo e rappresentano il valore attualmente richiesto al dispositivo ble. Quando queste arrivano al termine (fine dell'array) la procedura è terminata e si può passare alla prossima operazione (per quanto riguarda la lettura esiste una seconda variabile di indice incrementata nel callback all'interno di `HCI_Event_CB`, dove avviene effettivamente la lettura).
Il led blu (LD2) viene attivato all'inizio della comunicazione con la prima vetrina e spento alla fine dell'ultima, per ogni operazione.

1.2 COLLEGAMENTO ETHERNET STM32F429ZI

1.2.1 Crittografia con protocolli TLS/HTTPS

Per migliorare la sicurezza della comunicazione, il sistema implementa la libreria di cifratura `mbedtls.h` (selezionabile da STMcube), abbastanza leggera per poter svolgere il suo lavoro con poche risorse e abbastanza avanzata nei metodi e protocolli. La comunicazione HTTP viene dunque 'filtrata' attraverso questa libreria, che la rende dunque accessibile solo a server e client. Durante la procedura di handshake, entrambi gli attori concordano un metodo di cifratura (vengono supportate ciphersuite con `ECHDE` e certificati `RSA` o `ECDSA`, che dovranno essere implementati sul server) e il certificato del server viene controllato (non ne sono stati utilizzati lato client). La scheda è in grado di autorizzare certificati self signed solo se il certificato della CA è salvato all'interno del firmware (come stringa in `srv_cert.c`, **TODO** implementare un filesystem e caricare lì il certificato).

1.2.2 Comunicazione come client web

funzione *ETH_Process* e files *Src/mbedtls.c*, *Inc/mbedtls.h*
Le interazioni (r/w) con il server trattano prima i dati del negozio (id, versione e stato) e successivamente i dati di ciascuna vetrina. Quando la connessione è stata stabilita si utilizza la funzione `PollEthData()` per mandare la richiesta e leggerne il responso, restituendo il corpo del messaggio e eventualmente un codice di errore.
Le richieste HTTP utilizzano il metodo POST (la query string si trova dentro il corpo del pacchetto) e viaggiano all'interno della stessa connessione TCP creata all'inizio del processo (pressione del bottone) e chiusa quando la comunicazione è terminata. La prima richiesta che viene fatta (QUERY `mode=syn`) richiede dal server data e ora aggiornate (se necessario), successivamente con `mode=shp` si richiedono i dati specifici del negozio e la versione del database presente sul server, se questi dati sono corretti vengono fatte successive richieste per gli effettivi dati delle vetrine. Questi vengono richiesti con `mode=ent` e nella risposta il numero di vetrine precede gli altri dati. Nella scrittura su server i dati sono direttamente inclusi nel corpo del pacchetto e vengono inviati a pagine specifiche (per evitare di utilizzare una query string insieme ai dati). Tutti i pacchetti contenenti dati di negozio o

vetrine sono in formato binario, dove ogni bit contiene un numero relativo ad un campo, per i campi a 16 bit è utilizzata la notazione big-endian (0xabcd diventa 0xab,0xcd).
Ipotizzando che ogni vetrina contiene 17 bit con 2 bit aggiuntivi di checksum, un massimo di 50 vetrine possono essere scritte su pacchetti con corpo 1000 bit, che con buona probabilità riempiono un singolo frame ethernet. I dati di accesso al database su server sono inclusi nell'header del pacchetto (sempre criptati).
Quando una funzione di connessione ritorna un errore, il led rosso (LD3) si attiva, mentre il led blu (LD2) rimane attivo tra una trasmissione e una ricezione.

1.3 NOTE

- La comunicazione bluetooth viaggia in formato testuale, una possibile ottimizzazione può essere passare al binario, riducendo così la grandezza dei pacchetti e le istruzioni necessarie alla gestione.

- La scrittura su server web è in po' più delicata perché nello spedire uno stream binario nel pacchetto HTTP, è molto facile utilizzare caratteri particolari che invaliderebbero il parsing della query string. A tal proposito si può gestire il dato ricevuto come puro binario su pagine apposite (invece di utilizzare la stessa pagina e cambiare in base al valore di `mode`).

- Bisogna prestare particolare attenzione alla configurazione del server HTTPS per garantire l'autenticazione col client, la board ha un supporto limitato di ciphersuite, una normale configurazione di apache con openssl dovrebbe funzionare. Il certificato può essere self-signed, in quel caso bisogna installare tutta la chain o anche solo la CA sul dispositivo, nel test è stato generato con chiave RSA ma dovrebbe essere supportata anche ECDSA.

- Utilizzando questi metodi di cifratura non è possibile leggere il contenuto del pacchetto da programmi come wireshark.

- Nonostante la maggior parte dei campi all'interno del database possano stare in 8 bit, per ognuno sono utilizzate variabili di tipo `uint16_t`, per garantire il corretto funzionamento di scanf nella lettura da stringa. Per come sono compilate le librerie, non si può eseguire una scansione numerica su una variabile short short (possibile solo da C99), ovviamente il problema non si porrebbe se si usassero stream binari nel trasfermimento.

- Le azioni r/w con ble, vengono gestite per singolo attributo e ognuno corrisponde ad un campo nella struct definita per le vetrine, per identificarla dall'esterno si usano le macro `FIELD_*`, utilizzate anche come indici nell'array degli handle (`CH_HANDLES[FIELD_FOO]==FOO_HANDLE`), per questa ragione è bene che vengano tenute in ordine, consecutive e che partano da 0.

- La sincronizzazione della data viene ripetuta ad ogni connessione, questo può essere cambiato in base alla modalità (`active_shop.status`) esattamente come l'ordine e la frequenza delle operazioni che vengono eseguite. Per fare ciò si devono modificare le funzioni `*_Process` in `main.c`.

- Con l'utilizzo di stream binari nella comunicazione, non è più possibile utilizzare le funzioni legate alle stringhe, poiché il valore `'\0'` che termina le stringhe è un valore perfettamente lecito all'interno di un pacchetto binario.

- Le query string utilizzate per le richieste al server sono nella forma `mode=abc` per poter occupare una dimesione fissa di 8 bit, necessaria alla funzione che compone il pacchetto http e non sempre ricavabile con `strlen()` (vedi sopra).

- I nomi utente per i singoli negozi (necessari per accedere al database) vanno scelti con cautela, nel php il nome viene filtrato per estrapolarne un contenuto numerico e questo viene usato come id del negozio, deve dunque essere l'unico numero presente nel nome (in questa fase vengono usati nomi del tipo shopX)

2 CHANGELOG DEI PROGETTI

v1.0.0 *eth_test*

- Comunicazione http configurata con api `netconn.h`
- Aggiunto ritardo per acquisizione ip con dhcp

v1.0.1 *eth_test*

- Aggiunta di funzioni per inviare richieste e gestire la risposta
- Implementazione del metodo POST nelle richieste
- Introduzione delle strutture per il database locale
- Formato testuale nel pacchetto in entrata (problemi con scanf e uint8)

v1.0.2 *eth_test*

- Aggiunta delle strutture `shadow_` per gestire i dati durante il fetch
- Chiusura della connessione dopo ogni trasmissione (limitazione di netconn)
- Separazione in thread diverse (coordinate da semafori) per distribuire la memoria

v1.0.3 *eth_test*

- Migrazione alla api `socket.h` per poter trasmettere più pacchetti nella stessa connessione
- Aggiunto calcolo del checksum
- Ritrasmissione dei pacchetti su errore (checksum)

v1.1.0 *eth_test*

- Aggiunta del supporto a mbedTLS, importato da STM32Cube
- Inizia il passaggio a HTTPS (http criptato)
- Errori nella procedura di handshake con il server

v1.0.4 *eth_test*

- Aggiunta del supporto alle ciphersuite con RSA per una migliore compatibilità
- Ridimensionata il numero di MPI per gestire le chiavi RSA
- Risolti problemi sull'handshake, da installare il certificato sul dispositivo

v1.0.5 *eth_test*

- Installato il certificato in una sorgente a parte (`srv_cert.c`)
- La connessione si apre e si chiude correttamente

v1.0.0 *bl_merge*

- Porting da sensor-demo per stm32f401
- Adattati i nomi dei pin

v1.0.1 *bl_merge*

- Risolti i problemi di spi configurando il pin SPI_IRQ del BNRG come EXTI (EXTeral Interrupt)
- Impostazione come client ble (originariamente server)

v1.1.0 *bl_merge*

- Comunicazione con server ble funzionante
- Definizione della data come struct

v1.1.0 *bl_merge*

- Aggiunto supporto per connettersi a più dispositivi (array contenente tutti gli indirizzi)
- Aggiunta del timeout per la connessione

unione tra i progetti *eth_test* e *bl_merge* nel nuovo progetto *shop429*

v1.2.0 *shop429*

- Merge dei progetti su eth e ble
- Particolare attenzione alle librerie come `Inc/stm32f4xx_it` per il corretto funzionamento del ble

v1.2.1 *shop429*

- Implementazione della macchina a stati anche per la connettività web (`ETH_Process()`)
- Funzione richiamata ad ogni passaggio, necessario l'uso di variabili statiche
- Riunione in singola thread del processo di eth rimuovendo i semafori
- La versione del database locale viene aggiornata soltanto dopo il fetch completo della tabella

v1.2.2 *shop429*

- Risolti problemi della macchina a stati ble aggiornando lo stato in base a quello vecchio quando ricominciano le vetrine
- Implementazione della scrittura su server (eth) in formato testuale

v1.2.3 *shop429*

- Uso parziale della variabile locale di stato del negozio
- PRINTF configurato per il debug (SWV console)
- Ora viene gestita la disconnessione bluetooth anche dall'altro dispositivo

v1.2.4 *shop429*

- Uso del formato binario nelle connessioni al server web
- Inclusa l'autenticazione al server nell'header http
- Connessione con database reale sul server (sorgenti php incluse nel progetto)