# Relazione di tirocinio

Gabriele Monaco 20 giugno 2018



# Indice

	0.1	Note s	ulla relazione	3
1	$\mathbf{GE}$	STION	IE NEGOZIO CON STM32F429ZI	3
	1.1	Introd	uzione	3
		1.1.1	Hardware impiegato	4
		1.1.2	Ambiente di sviluppo	4
	1.2	Strutti	ura del software	
		1.2.1	Multithreading con FreeRTOS	5
		1.2.2	Stati di esecuzione	6
	1.3	Strutture dati		
		1.3.1	Implementazione del database locale	7
		1.3.2	Database remoto e interfaccia web	9
	1.4	Gestio	ne firmware e memoria	10
		1.4.1	Mappatura delle memorie	10
		1.4.2	Emulazione di una EEPROM su memoria flash	10
		1.4.3	Aggiornamenti FOTA	10
		1.4.4	Struttura base del bootloader	11
	1.5	College	amento Bluetooth STM32F429ZI	
		1.5.1	Funzionamento delle librerie e dei protocolli	
		1.5.2	Comunicazione come client BLE	12
	1.6	College	amento Ethernet STM32F429ZI	
		1.6.1	Crittografia con protocolli TLS/HTTPS	
		1.6.2	Funzionamento delle librerie	
		1.6.3	Comunicazione come client web	14
		1.6.4	Scaricamento dal server del firmware	15
	1.7	Note		17
<b>2</b>	СН	ANGE	LOG DEI PROGETTI	20

## 0.1 Note sulla relazione

All'interno di questa relazione userò sempre la terza persona, nonostante ciò questa parte di progetto è stata tenuta da me, dunque le varie implementazioni (qualora non siano librerie o specifiche richieste del cliente) sono da considerarsi mie contribuzioni

Spesso per trattare di strutture dati, variabili o funzioni con parti del nome in comune, verranno usate alcune wildcards, ad esempio quando si parla di vetrine e negozio, le variabili foo\_\* saranno foo\_vetrine e foo\_shop, in maniera analoga dove questo può non essere facilmente deducibile vengono usati metodi del tipo [foo/bar] [Eth/Ble] che stanno ad indicare tutte le combinazioni possibili cambiando le parole tra le [ ] (quindi in questo caso ci si riferisce a fooEth,fooBle,barEth,barBle). In ogni caso queste notazioni vengono utilizzate per esprimere identificatori presenti nel codice, è dunque immediata la comprensione quando si ha accesso alla sorgente e comunque non viene preclusa la comprensione dei ragionamenti espressi in caso contrario.

Vengono utilizzate all'interno del documento (e del codice) abbreviazioni come ETH, per riferirsi alla connessione che passa attraverso l'interfaccia ethernet e BLE, che indica la connessione bluetooth (low energy). Inoltre le operazioni di lettura/scrittura vengono spesso abbreviate con le iniziali dall'inglese  $(\mathbf{r}/\mathbf{w})$ .

Questa relazione è stata pensata sia per documentare il lavoro svolto nell'ambito del tirocinio, che per spiegare al meglio a chi prenderà in mano il codice come questo è stato pensato. Per ragioni aziendali il codice stesso non può essere distribuito e dunque non sarà incluso all'interno di questa relazione. Si cerca comunque di rendere più chiaro possibile il lavoro fatto anche senza l'accesso alle sorgenti.

Nella sezione Note sono riportate alcune notazioni importanti utili per comprendere alcune scelte all'interno del codice o per capire quali scelte non possono essere prese sviluppando il progetto. Sono principalmente intese per chi lavora sul progetto anche se possono dare strumenti in più per comprendere alcuni problemi riscontrati durante la fase di sviluppo.

# 1 GESTIONE NEGOZIO CON STM32F429ZI

### 1.1 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 e nell'assistenza remota. All'interno del negozio ogni vetrina avrà un led RGB, controllato con PWM da una schedina STM32 (da definire se F401 o F072), 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 e i firmware per poter eseguire un aggiornamento OTA (Over The Air).

Verranno utilizzati principalmente software e hardware prodotti dalla ST Microelectronics, azienda italo-francese specializzata nella produzione di dispositivi elettronici, principalmente utilizzati dello sviluppo di tecnologie embedded. STM32 è una famiglia di microcontrollori basati su processori ARM a 32 bit, la ST produce piattaforme hardware e software per poter sviluppare su questi dispositivi. Questo specifico progetto è nato inizialmente per la singola gestione della comunicazione via ethernet dei dati con il server (eth\_test), successivamente è stata implementata in maniera separata la funzionalità bluetooth (bl\_merge) e infine le due sezioni sono state unite per comporre il progetto come lo si trova al momento della stesura di questo documento (shop429). Uno storico delle varie modifiche può essere consultato nella sezione CHANGELOG del documento, questa ripercorre quasi

completamente i commit effettuati sulle repository git dei diversi progetti, divisi opportunamente tra versioni in base all'entità delle modifiche.

### 1.1.1 Hardware impiegato

## figura 1

Questo progetto mira a scrivere un firmware per la board STM32F429ZI Nucleo, una scheda di sviluppo fornita dalla STM e basata su processore ARM Cortex-M4, l'architettura presente è interamente a 32 bit. La board presenta 144 pin utilizzabili per collegare altre periferiche (tra cui quelle che soddisfano le specifiche ST morpho e Arduino R3) ed è costruita per poter essere assemblata con altre board simili in una pila avente i pin comunicanti. Il SoC supporta diverse periferiche tra cui diversi bus SPI e una presa ethernet (il connettore è presente sulla board).

Per poter utilizzare la connettività bluetooth è necessario includere una estensione. In questo caso, è stata utilizzata la scheda X-NUCLEO-IDB05A1, un modulo che implementa la connettività BLE (Bluetooth Low Energy) e installabile sulle board di sviluppo STM32 utilizzando i connettori Arduino R3. A causa di sovrapposizione tra alcuni pin del processore, utilizzabili sia per ethernet che SPI, è stato necessario fare delle modifiche su entrambe le board per poterli spostare. Questo è stato facilmente ottenibile muovendo i relativi soulder bridge sulle schede.

Le board di sviluppo STM32 (tra cui quella utilizzata) contengono l'interfaccia ST-Link per agevolare la connessione e l'interazione della scheda con il PC, questa sezione contiene l'hardware necessario per permettere programmazione e debug del microcontrollore e può essere collegata con un cavetto micro-USB (in alternativa è possibile configurare l'UART). Tale estensione può essere rimossa, se necessario, o collegata ad altre schede per poterle programmare.

### 1.1.2 Ambiente di sviluppo

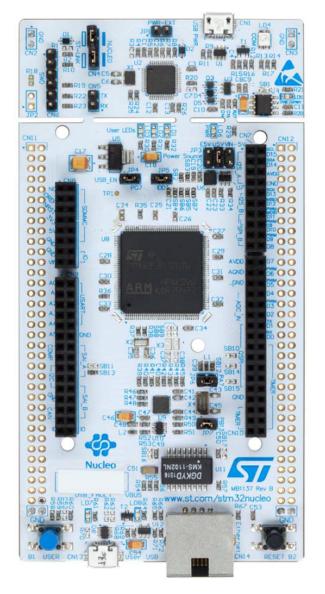
Il progetto è portato avanti sull'IDE Atollic TrueStudio for STM32, toolchain basata su Eclipse che usa i tool di compilazione e debug GNU (gcc e gdb) recentemente acquisita dalla ST stessa (è stata utilizzata la versione 9.0.0).

Si è partiti da un progetto generato dal software STM32CubeMX, in grado di generare progetti per le varie toolchain contenenti tutte le estensioni Driver e Middleware fornite da ST, impostando i valori specifici per ogni board o SoC. Vengono generate anche le varie inizializzazioni di periferiche e software aggiuntivi e quasi tutti i parametri possono essere direttamente configurati dalle apposite schermate del tool (è stata utilizzata la versione 4.25.1).

Per il controllo delle versioni si ricorre a git, direttamente accessibile da TrueStudio attraverso l'apposita estensione di Eclipse, 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 avanti 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 (definita tra le sorgenti BlueNRG), 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).

Nonostante la programmazione del firmware in flash (in formato ELF) avvenga direttamente in fase di debug utilizzando Atollic, esiste anche il tool STM32Programmer, utile per gestire direttamente la flash con operazioni come lettura e scrittura. Con questo software si può dunque programmare la scheda a partire dall'area di memoria specifica e utilizzando diversi formati (utile se si utilizza un bootloader all'inizio della flash e per avere un eseguibile con dimensioni più contenute) oppure





(b) X-NUCLEO-IDB05A1

(a) STM32F429ZI-NUCLEO

Figura 1: Board di sviluppo ST

leggere specifiche aree di memoria che possono essere state programmate dalla board stessa (eeprom simulata o download del firmware).

## 1.2 Struttura del software

### 1.2.1 Multithreading con FreeRTOS

Nel firmware è abilitata la funzionalità di FreeRTOS (Real Time Operating System), una sorta di rudimentale sistema operativo in grado di gestire funzionalità base di programmazione concorrenziale su dispositivi embedded. Supporta thread multiple e varie strutture per sincronizzare e gestire queste ultime, come semafori e timer. Per limitazioni di hardware (il processore è unico), le thread non possono essere realmente eseguite in parallelo, ma l'RTOS è in grado di alternare le esecuzioni delle varie thread quando queste vengono temporizzate. In pratica ogni thread è strutturata all'interno di una funzione contenente un loop infinito, all'interno di questo risiede l'effettivo codice da eseguire e

al termine viene chiamata la funzione osDelay(). Questa informa il sistema operativo che le risorse possono essere impiegate per un'altra thread in attesa e la precedente viene messa in coda per un numero di millisecondi passato come argomento della chiamata. Al termine di tale intervallo la thread potrà essere eseguita non appena il sistema sarà in grado di allocarle le risorse.

All'inizio dell'esecuzione (main()), vengono eseguite tutte le inizializzazioni necessarie di hardware, strutture dati e OS, a questo punto è possibile caricare quest'ultimo. Al momento in cui questo accade, il controllo passa allo scheduler (osKernelStart()), che eseguirà le thread allocate precedentemente e tutto il codice, all'interno del main, che viene dopo questa chiamata non sarà più eseguito, in quanto il controllo passerà unicamente dalle thread.

La memoria utilizzata dall'RTOS è allocata dinamicamente (nella heap), viene però settato un limite all'interno della configurazione in STM32Cube che rappresenterà la memoria effettivamente designata al sistema operativo, utilizzata poi per allocare le varie thread, semafori e altre strutture. Ogni thread avrà a sua volta (all'interno della definizione) un limite di memoria che potrà utilizzare, questa memoria farà chiaramente parte di quella destinata all'OS e verrà utilizzata come stack della funzione (e delle varie funzioni richiamate da questa, nonostante sia fisicamente in heap). E importante notare che questo limite di memoria per thread dev'essere aggiustato a dovere, da un lato per evitare di utilizzare più risorse del necessario (questo può dipendere dal metodo di allocazione utilizzato, sempre impostabile dal STM32Cube), dall'altro però deve esserci spazio per tutte le variabili locali dichiarate all'interno della thread (di fatto dinamiche). Una mancata considerazione di tale limite può portare a comportamenti inaspettati, in quanto il sistema non è in grado di capire quando la memoria a disposizione è giunta al termine e quindi smette semplicemente di funzionare, spesso senza mostrare errori in particolare. Tutte le variabili globali sono allocate staticamente (in quanto accessibili da ogni thread), non vanno quindi ad impattare sulla memoria dedicata all'OS, in maniera analoga si possono impostare le variabili locali come static per ottenere un risultato più simile a quello delle variabili globali (con tutto ciò che ne consegue, la keyword static in questo caso permette un'allocazione statica, ma non è questa la sua funzione).

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). Le funzioni che gestiscono le connettività [ETH/BLE]\_Process(), sono richiamate all'interno delle rispettive thread e vengono eseguite più volte, fino a terminare le loro mansioni.

Per tenere traccia dei passaggi fatti, dunque, vengono utilizzate variabili create come static, che si comportano esattamente come variabili globali, mantenendo il valore tra una chiamata e l'altra, ma il loro scopo è ristretto alla funzione in cui sono definite (come quelle locali). Per quanto riguarda invece le variabili globali che vengono utilizzate in più thread o che vengono modificate a seguito di interrupt, la keyword volatile è utilizzata, in questa maniera si informa il compilatore che tali variabili non devono essere ottimizzate in alcun modo. Il compilatore infatti può non essere completamente consapevole di come tali variabili vengono modificate e potrebbe quindi rimuovere alcune istruzioni che considera superflue, compromettendo però il corretto funzionamento di alcune procedure.

### 1.2.2 Stati di esecuzione

### file 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).

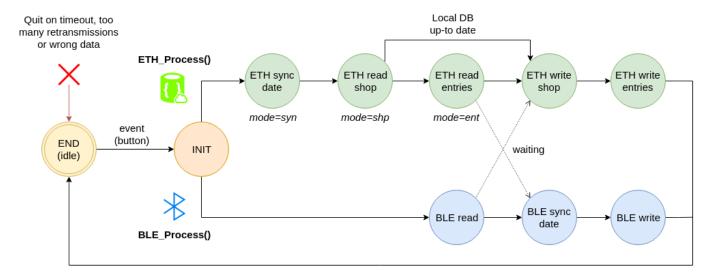


Figura 2: Schema della macchina a stati

### 1.3 Strutture dati

### 1.3.1 Implementazione del database locale

file 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 da ETH i dati vengono salvati all'interno delle strutture puntate da shadow\_ e se 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. Su ogni riga (negozio e vetrina) viene calcolato un checksum utilizzando l'algoritmo BSD su 2 byte, se la sua verifica fallisce si tenterà una ritrasmissione. Questo algoritmo per il checksum non fa altro che calcolare un valore, il più possibile univoco, partendo dai dati che vengono passati: iniziando da zero, ad ogni passo il valore (su 16 bit) subisce una rotazione bitwise a destra e gli viene sommato il dato (8 bit) corrente. In questa maniera sia un cambiamento nei dati che un dato mancante modificherebbero il checksum e in generale è improbabile che una modifica arbitraria generi lo stesso valore. Questo dato viene utilizzato sia

in ETH che leggendo dalla memoria locale (eeprom), un confronto tra il valore calcolato prima (e salvato/ricevuto asseme ai dati effettivi) e quello calcolato in lettura/ricezione rivela con una certa accuratezza se i dati sono corretti.

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.

Le variabili che fanno parte del database contengono i valori iniziali che verranno usati prima del fetch (e qualora questo non andasse a buon fine), i valori del negozio sono settati come default (definite attraverso costanti) mentre quelli delle vetrine sono tutti nulli fatta eccezione per alcuni id della tabella che diventerà active\_ che conterrà le vetrine 1, 2 e 3. Durante la procedura di inizializzazione del database, viene controllata anche la eeprom e vengono acquisiti gli eventuali dati, solo se coerenti. La procedura è simile a quella effettuata per la ricezione da ETH, in cui le tabelle shadow\_ vengono popolate e successivamente attivate se il checksum corrisponde (o in generale il dato letto ha senso). Le funzioni Activate\* utilizzate a tal proposito, salvano anche i dati ricevuti sulla eeprom se questa funzione è abilitata (cosa che capita in ricezione dal server e chiaramente non dopo la lettura dalla stessa eeprom).

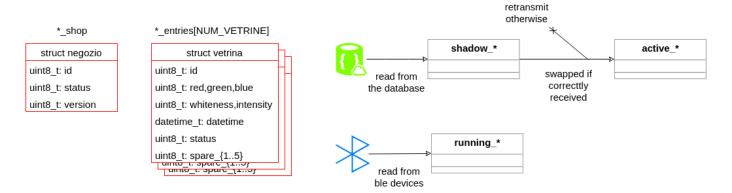


Figura 3: Struttura del database locale

#### 1.3.2 Database remoto e interfaccia web

Il database remoto viene gestito su server apache (o qualunque altra alternativa che supporti php e mysql). È stato preferito il binomio php-mysql a tecnologie più moderne come ad esempio Node.js (accompagnato con database NoSQL) principalmente per la sua semplicità d'uso. In questo progetto l'obiettivo principale non è un'interfaccia web dinamica, dove Node sarebbe particolarmente indicato, ma un accesso rapido dei dati sia da dispositivi embedded (dove verosimilmente non viene impiegato javascript) che da utenti che passano da un browser, vengono quindi privilegiate le funzionalità base di php.

Nella cartella server\_files sono contenute le pagine che devono stare nella cartella pubblica sul server (www) 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 (chiaramente 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 o crearne un'interfaccia dedicata)

Il fatto che le board possano soltanto aggiornare i dati ricevuti dalle vetrine salvati sul server, implica che non si tenga traccia delle vetrine che non hanno dato risposta (semplicemente non ne vengono modificati i dati). Implementare un DELETE potrebbe risultare rischioso o poco pratico, ma è necessario segnare quali vetrine sono effettivamente presenti e attive. Una possibile soluzione sarebbe l'impostazione di una flag DISABLED all'interno del database su tutte le vetrine ad ogni aggiornamento e settarla come falsa per ogni vetrina effettivamente aggiornata. (TODO da implementare)

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 da input).

## 1.4 Gestione firmware e memoria

### 1.4.1 Mappatura delle memorie

I dispositivi basati su processori Cortex-M\* (come la board impiegata in questo progetto), vedono le varie memorie disponibili mappate all'interno dello stesso spazio di indirizzi, indipendentemente dalla loro tecnologia. In questo caso, il processore è in grado di indirizzare 4GB di memoria (da 0x00000000 a 0xFFFFFFFF, potendo utilizzare 32 bit) ma solo alcuni di questi sono effettivamente attivi. Nello specifico sulla board sono presenti nativamente una SRAM, allocata tra 0x20000000 e 0x20030000 (192k) utilizzata come ram principale, una CCRAM, allocata tra 0x10000000 e 0x10010000 (64k) al momento inutilizzata e la flash, allocata tra 0x08000000 e 0x0820000 (2M) utilizzata per contenere il firmware e tutti i dati che non devono essere volatili (e.g. perdersi con uno spegnimento).

### 1.4.2 Emulazione di una EEPROM su memoria flash

La board STM32F429ZI non possiede di base una eeprom, ma ha una memoria flash (dove in genere viene scritto il firmware) di 2MB e nonostante la differente tecnologia che utilizzano le due memorie, può simulare il comportamento della eeprom. La prima difficoltà è rappresentata dal fatto che la flash può essere programmata solo se la scrittura coinvolge un passaggio da 1 a 0, altrimenti è solo possibile la cancellazione dell'intera pagina (o settore, in questo caso grande 16kB). Per ovviare a questo problema, le pagine vengono prima cancellate (settando i bit ad 1) e invece di scrivere word (32 bit) negli indirizzi dedicati, si scrive il dato solamente in mezza word (16 bit) e la restante parte viene utilizzata come indirizzo virtuale. Ogni scrittura successiva aggiungerà una nuova coppia indirizzo dato al fondo della memoria utilizzata (dove è ancora possibile programmare), la lettura dunque andrà a prendere soltanto l'ultimo dei valori con l'indirizzo specificato (sarà dunque necessaria una scansione lineare della pagina). Quando una pagina viene terminata, le coppie ancora attive (le ultime scritte per ogni indirizzo) vengono scritte nella seconda pagina, questa viene segnata come attiva e la precedente viene cancellata, in questo caso le pagine utilizzate sono 2 di cui solo una attiva. Il tutto è gestito utilizzando librerie esterne che implementano questo algoritmo.

In questo caso la simulazione su flash avviene nei primi due settori del secondo banco (12 e 13 a partire da 0x0800000), sufficientemente distanti per essere sicuri di non sovrascrivere settori utilizzati dal firmware (che parte in genere da 0x08000000, all'inizio della memoria) e riprogrammare la scheda con risultati inaspettati (se non addirittura pericolosi). È necessario che entrambi abbiano la stessa grandezza e questa è stata scelta più piccola possibile per evitare di consumare troppa flash quando non necessario.

### 1.4.3 Aggiornamenti FOTA

Per poter apportare modifiche senza dover sempre intervenire fisicamente sulle schede, si è scelto di implementare l'aggiornamento Firmware Over The Air, ovvero il download del firmware tramite connettività (in questo caso ethernet, per le vetrine avverrà lo stesso con il bluetooth). Il firmware viene scritto in una zona inutilizzata della flash, che deve quindi essere abbastanza capiente per poter tenere entrambe le versioni, poi quando l'eseguibile è stato correttamente ricevuto, il dispositivo si riavvia. Appena acceso il dispositivo non esegue immediatamente l'applicazione, ma inizia dal bootloader (scritto all'inizio della memoria flash), questo controlla se è stato effettuato un aggiornamento e in caso positivo copia il contenuto della seconda partizione nella prima, sovrascrivendo dunque la vecchia versione. A questo punto l'applicazione che si trova nella prima partizione può essere eseguita.

#### 1.4.4 Struttura base del bootloader

Il bootloader si trova, per necessità, in un progetto separato, in questo viene inizializzato il poco hardware necessario, vengono fatte le varie copie del firmware per avere l'applicazione nella partizione desiderata e viene preparato l'ambiente per poterla eseguire. Essenzialmente viene deinizializzato l'hardware (che sarà eventualmente ri-inizializzato all'interno dell'applicazione) e viene spostato il program-counter nella posizione preposta all'esecuzione.

Quando la board viene accesa (in condizioni normali), vengono eseguite le prime istruzioni che si trovano in flash (0x0800000), a tal proposito il bootloader deve trovarsi in quella posizione, così da essere eseguito direttamente ad ogni reset o spegnimento. In un'altra posizione ben definita dovrà trovarsi l'applicativo, così da permettere al bootloader di effettuare il 'salto' nella posizione voluta. Di norma un firmware contiene nella prima word (4 byte) la cosiddetta vector table, utilizzata nell'inizializzazione degli interrupt, prima di effettuare il jump, dunque, si dovrà spostare il main-stack-pointer in questa posizione, ma l'esecuzione effettiva partirà dopo la vector table (ADDR+4), si dovrà quindi lanciare l'esecuzione di una funzione a tale indirizzo in flash.

È importante che anche l'applicazione sia consapevole della sua posizione in memoria, principalmente per conoscere dove cercare la vector table e per avere un eseguibile compilato correttamente. A tal proposito va modificato il linker script, impostando uno scostamento nell'inizio della flash (0x20000 in questo caso), in questa maniera la compilazione dell'eseguibile sarà effettuata considerando l'ambiente di destinazione. Inoltre l'applicazione deve contenere tale offset all'interno del proprio codice, così da generare correttamente il puntatore alla vector table, invece di settare un diverso indirizzo iniziale della flash, viene impostata la specifica costante VECT\_TAB\_OFFSET in modo da spostare soltanto tale struttura, senza rendere inaccessibile la prima sezione di memoria flash.

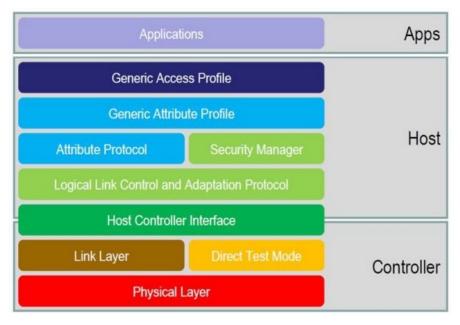
A questo punto l'eseguibile dovrà essere programmato all'indirizzo corretto (con STM32Programmer o via OTA), anche se è ancora possibile la programmazione attraverso la toolchain Atollic, in quanto l'eseguibile elf contiene le informazioni su dove deve essere flashato. Non sarà dunque possibile il debug dal progetto del bootloader (contiene solo la propria sorgente), ma sarà perfettamente funzionale utilizzando quello dell'applicativo.

# 1.5 Collegamento Bluetooth STM32F429ZI

### 1.5.1 Funzionamento delle librerie e dei protocolli

Le librerie per la gestione del BLE (Bluetooth Low Energy), utilizzano i servizi GAP (Generic Access Profile), per quanto riguarda l'apertura/chiusura della connessione, e GATT (Generic ATTribute profile), per lettura/scrittura e altre funzionalità. In entrambi i casi il paradigma utilizzato è quello callback, così le chiamate a funzione lanciano richieste all'estensione bluetooth e quando queste ricevono risposta, un interrupt viene generato e la funzione HCI\_Event\_CB() viene chiamata con argomento il puntatore all'evento (una struct che contiene i dati necessari), in base al tipo di tale evento viene eseguita l'azione corrispondente. Queste sorgenti sono fornite dalla ST come test per l'interfaccia BlueNRG utilizzando alcune delle schede a 64 pin, anche se l'estensione è compatibile in generale con i pin Arduino R3, quindi con qualche modifica è stato possibile effettuare un porting. Nel protocollo gap sono definiti 4 ruoli che i dispositivi possono assumere, in questo caso vengono utilizzati solo quello di *Peripheral* e *Central*. Il primo non fa altro che notificare l'ambiente circostante della sua presenza (advertising) informando delle sue capacità di offrire servizi. Il secondo è in grado di stabilire una connessione con un peripheral e di richiedere servizi, svolge quindi un riolo attivo nella topologia. Questi concetti si trasportano nel protocollo gatt, dove in genere il peripheral coincide con il *Server* mentre il central assume il ruolo di *Client*.

Gli standard gatt, consentono ai server di fornire i propri servizi sotto forma di caratteristiche, strutture dati logicamente omogenee all'interno di un servizio che a loro volta possono contenere una coppia proprietà/valore e dei descrittori. I client connessi sono in grado di leggere o scrivere all'interno di tali caratteristiche e così avvengono i passaggi di dati. Il client, in genere non conosce le varie caratteristiche presenti sul server, a tal proposito esiste una chiamata di discovery per poter appunto capire quali servizi (e caratteristiche) sono disponibili e come accedere ad esse. Il protocollo prevede che ogni caratteristica o servizio abbia un proprio UUID da 128 bit, che dovrebbe essere il più possibile unico, sono anche definiti degli UUID più brevi (16 o 32 bit) per quanto riguarda servizi predefiniti dallo standard. Ad ogni modo, quando il client viene a conoscenza dei vari UUID dei servizi accessibili, è in grado di accedervi attraverso degli indirizzi specifici sul server, detti handle. In questa specifica applicazione, non era necessaria tutta questa genericità, il client è programmato per dialogare solamente con le vetrine, può dunque conoscere a priori la struttura dei servizi e accedervi direttamente attraverso gli handle che sono già conosciuti (evitando dunque tutta la procedura di discovery).



PROFILE

SERVICE

CHARACTERISTIC

CHARACTERISTIC

CHARACTERISTIC

SERVICE

CHARACTERISTIC

CHARACTERISTIC

CHARACTERISTIC

(b) Struttura GATT

(a) Architettura Bluetooth Low Energy

Figura 4: protocolli BLE

### 1.5.2 Comunicazione come client BLE

### funzione BLE Process e file 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...). Le operazioni disponibili sono lettura di un parametro dalla vetrina, scrittura di tale parametro e sincronizzazione della data sulle vetrine (non è altro che una scrittura ma su un attributo particolare che non è incluso nel database).

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 nell'array degli handle), le variabili reading e writing 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). Qualora una vetrina non sia disponibile, nonostante il database la contenga, la scheda tenta di connettersi ma entra ben presto in timeout rendendosi conto di non ricevere risposta. A questo punto ignora la vetrina e passa a quella successiva. Nessuna operazione viene fatta per tener conto di questa condizione, si fa sempre fede al database ricevuto via ETH, se questo dovesse contenere una vetrina in più o si fosse verificato un guasto, ogni ripetizione del processo andrebbe in timeout a quel punto, senza comunque bloccare l'esecuzione per le altre vetrine. (TODO implementare una notifica lato server per le vetrine che non rispondono)

Il led blu (LD2) viene attivato all'inizio della comunicazione con la prima vetrina e spento alla fine dell'ultima, per ogni operazione.

# 1.6 Collegamento Ethernet STM32F429ZI

### 1.6.1 Crittografia con protocolli TLS/HTTPS

Per migliorare la sicurezza della comunicazione, il sistema implementa la libreria di cifratura mbedtls.h (selezionabile da STM32Cube), 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).

Nello specifico ogni trasmissione inizia come connessione TCP, viene dunque effettuato il three-way handshake dove il client chiede di stabilire la connessione attraverso un segmento (vuoto) di SYN, questo viene riconosciuto con un pacchetto SYN, ACK che verrà a sua volta confermato da una ACK dal client (questa flag viene utilizzata per confermare la ricezione di ogni pacchetto all'interno del protocollo TCP). A questo punto può iniziare la procedura di handshake per la connessione criptata (TLS1.2), che in maniera analoga utilizzerà un Client Hello per segnalare l'intenzione di iniziare la connessione (includendo i parametri necessari a concordare la cifratura), seguito da un Server Hello che include il certificato utilizzato e da una (eventuale) conferma lato client di corretta inizializzazione del tunnel TLS (figura 5). Ora i dati possono essere passati in tutta sicurezza come normali pacchetti HTTP che, criptati, saranno leggibili solo dal diretto destinatario, inoltre qualunque pacchetto ricevuto al di fuori dell'ambito di tale connessione sarà scartato (risulterebbe illeggibile dopo una decodifica, non essendo stato creato con la chiave corretta).

#### 1.6.2 Funzionamento delle librerie

La connessione ad internet della board attraverso le librerie mbedtls è effettuata esclusivamente con chiamate procedurali, dunque l'esecuzione è il più possibile consecutiva. Durante l'inizializzazione, vengono impostati i parametri generali necessari alla crittografia (numeri casuali, certificati e varie configurazioni), questo capita solamente quando la scheda viene accesa. Ad ogni esecuzione la connessione viene aperta, la funzione di connessione crea un nuovo socket ed inizia la connessione al server tramite quella struttura, utilizzando le configurazioni tls precedentemente impostate, questo inizia la procedura di handshake (accordi riguardo autenticazioni e crittografia) con il server. Ogni comunicazione avviene tramite normali richieste http criptate secondo i criteri stabiliti durante l'handshake, in maniera analoga vengono ricevute e gestite le risposte. Al termine della procedura la connessione e il relativo socket vengono chiusi.

### 1.6.3 Comunicazione come client web

# funzione ETH Process() e file Src/mbedtls.c, Inc/mbedtls.h

Anche qui le operazioni disponibili sono lettura, per i dati dal server, scrittura, di dati ricevuti sul server e sincronizzazione, per la data corrente presa dal server. Sono disponibili anche funzionalità di download del firmware dal server.

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. 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 byte, che rientra nel limite imposto per il buffer di ricezione (1200, dove c'è da considerare anche l'header). I dati di accesso al database su server sono inclusi nell'header del pacchetto (sempre criptati).

Ogni volta che durante la trasmissione viene registrato un errore non critico (ad esempio un checksum sbagliato), si ritenta la medesima procedura, fino ad un massimo di MAX\_RETRY (impostato a 5), se questo limite viene superato l'intera procedura si interrompe con un errore. Per implementare una ritrasmissione è sufficiente incrementare il valore della variabile retry\_num (statica e utilizzabile solo per ETH) e ritornare dalla funzione, durante la successiva esecuzione (in ogni caso) ci sarà un controllo per verificare se il limite è stato superato, in modo da lanciare l'errore. È importante ri-inizializzare la variabile a 0 ogni volta che la procedura ha successo (essendo statica manterrebbe il valore).

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.

#### 1.6.4 Scaricamento dal server del firmware

### funzione FotaEthUpdate() in Src/mbedtls.c

Il download del firmware viene fatto sempre tramite https, ma viste le notevoli dimensioni non può essere bufferizzato in ram e successivamente scritto in flash, è quindi necessario salvare direttamente in flash ogni pacchetto quando questo viene ricevuto. La scrittura viene effettuata in una partizione apposita della memoria. Per come è configurata mbedTLS, i segmenti tcp hanno un limite di grandezza molto conservativo, utile per mantenere un ridotto utilizzo di RAM, ma problematico in questo caso. A tal proposito il valore di SSL\_MAX\_CONTENT\_LEN andrebbe allargato o disabilitato (di default è 2kB mentre il massimo è intorno ai 16kB ma tale valore introduce un notevole aumento di memoria utilizzata).

Per un trasferimento ottimale è stata utilizzata la codifica di tipo chunked (Transfer-encoding: chunked, dall'inglese a pezzi), il che significa che i dati mandati dal server (sempre nello stesso pacchetto http) vengono divisi in pezzi, ogniuno preceduto dalla propria grandezza in formato esadecimale e da \r\n per separare dall'effettivo chunk, che a sua volta termina con \r\n. Il server così ha tempo di preparare progressivamente ogni chunk e il client riesce a gestire correttamente i pacchetti ricevuti senza introdurre troppo ritardo. Questo è possibile perché il client non chiede direttamente la risorsa (fota\_update.bin) ma una pagina php che, dopo aver controllato le credenziali di accesso, invia tale firmware. Per favorire la divisione in chunk, lo script utilizza fread() su una grandezza ridotta (intorno ai 2kB alla volta) all'interno dello script php, evitando dunque di servire l'intero file in una volta.

Le letture da socket (sul client) avvengono a blocchi grandi al massimo MAX\_ETH\_LEN, ma la lettura (per impostazione automatica) si ferma quando incontra dei fine riga. Questo fa sì che parti del pacchetto che non fanno effettivamente parte del firmware (come header e chunk size) sono lette separatamente, rendendone più semplice l'interpretazione. In ogni caso, per evitare problemi con particolari sequenze riscontrate in un firmware, il valore di grandezza del chunk viene letto dal socket (un byte alla volta, fino a \r\n) e poi viene letto esattamente lo stesso numero di byte dal socket (questa volta a blocchi grandi al massimo 1200 B per limitare l'uso di memoria). Questa sezione farà sicuramente parte del firmware e quindi potrà essere direttamente salvata in flash, calcolando i relativi checksum e dopo questa si potrà passare al chunk successivo.

Utilizzando questo metodo di codifica, non viene passato inizialmente l'header Content-size, per il client è dunque impossibile capire quando sarà arrivato l'ultimo chunk (il socket non andrà mai in EOF), il protocollo quindi prevede che al termine del download la connessione html venga chiusa, il client cioè riceverà un messaggio di PEER\_CLOSE\_NOTIFY. Alla ricezione vengono eseguiti i controlli di dimensione e checksum. Si procede sia facendo riferimento ai byte ricevuti e scritti che sommando i valori di chunk-size e si confrontano entrambi i risultati con un dato ricevuto dal server all'interno dell'header. In maniera analoga il checksum viene calcolato e confrontato con il valore ricevuto alla fine del firmware, calcolato sul server. Visto che il dato è calcolato sul momento questo non può essere incluso nell'header (non è ancora conosciuto in quel momento) e si utilizzano quindi gli ultimi 2 byte per contenere tale checksum in notazione big-endian. Per avere la certezza che tutti i byte siano correttamente scritti, visto che sulla flash vengono scritte solo parole intere (4 byte), viene aggiunto un padding di 3 byte vuoti (0xff) così da far rientrare anche gli ultimi byte spaiati nel firmware all'interno dell'ultima parola scritta in flash (con buona probabilità un firmware non soffre di questo problema in quando le istruzioni sono generalmente contenute in word).

Sono possibili due tipologie di download di firmware via OTA, quello abilitato dallo stato FW\_UP (all'interno di ETH\_Process()) che rappresenta l'aggiornamento del firmware locale e quello relativo a FW\_DL, destinato al download del firmware per le vetrine. Entrambe le esecuzioni sono uguali per quanto riguarda il download, eccetto che per quanto riguarda le vetrine, la query string dl=ble

è aggiunta al pacchetto per richiedere al server il firmware corretto. Il file binario viene scaricato e salvato in flash (in entrambi i casi nella stessa posizione), se si tratta dell'aggiornamento locale, un reset della board viene invocato, così da eseguire il codice del bootloader che installerà e lancerà il nuovo firmware. (**TODO** configurare il passaggio via ble del firmware alle vetrine)

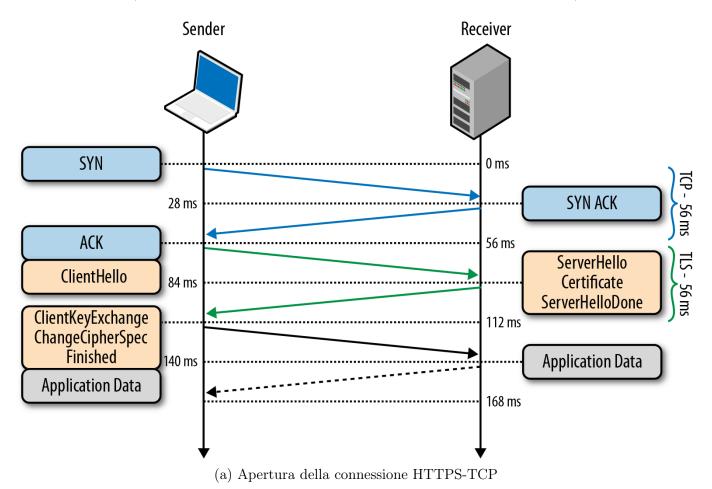


Figura 5: protocollo HTTPS

### 1.7 Note

- Il debug attraverso la console (utilizzando printf) può essere effettuato utilizzando la console SWD (accessibile dal debugger di Atollic) o usando l'UART (necessaria la connessione seriale e un applicazione esterna in grado di leggerne i dati). Nel primo caso basta abilitarla attraverso la configurazione di debug, impostando il clock corretto reperibile in STM32Cube come HCLOCK. Va modificata la funzione \_write(), utilizzata da printf ma definita come \_weak\_ all'interno delle librerie per consentirne l'override in fase di linking con una funzione con lo stesso nome. La si può trovare in Src/syscalls.c, anche se può essere direttamente aggiunta in altre sorgenti senza dover includere l'intero file. In alternativa si può mantenere la definizione di default in syscalls.c, che chiama a sua volta la funzione \_\_io\_putchar() (che definisce la scrittura per carattere invece che per stringa) e sovrascrivere quella con i metodi per SWD/UART.
- Il debug è abilitato attraverso la costante DEBUG, che definisce la funzione PRINTF come printf (altrimenti non definita), è bene disabilitare tale funzione quando si passerà ad un ambiente di release, in quanto introduce funzionalità inutilizzate che potrebbero rallentare il sistema.
- 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 (già implementato nella connessione via http).
- 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 \$\_REQUEST[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 openssi 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.
- Per la connessione HTTPS non vengono utilizzati certificati lato client in quanto non espressamente richiesto dallo standard, potrebbe essere un'ulteriore sicurezza per controllare gli accessi al server (in aggiunta al controllo della password)
- Indipendentemente dal metodo effettivamente implementato nel sistema, ogni stream binario all'interno di questa applicazione che prevede byte multipli, utilizza una notazione big-endian, dove i byte più significativi precedono quelli meno significativi (viene gestito tutto via software, perciò è indipendente dall'architettura), dunque una grandezza a 16 bit con valore Oxabcd viene trasmessa separando i byte come Oxab e Oxcd, la notazione contraria (little-endian) viene invece utilizzata a livello di firmware.
- 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.
- Le variabili del database locale riguardanti le vetrine sono impostate a 0 sfruttando il fatto che la memoria allocata staticamente non può essere parzialmente inizializzata, dunque basta settare alcuni byte (in questo caso gli id dei primi 3 record, il loro valore è ininfluente) per avere il resto della struttura inizializzata automaticamente.
- Per come sono configurate ora, sia vetrine che negozi contengono il proprio id all'interno del firmware (attraverso un #define), questo fa si che ogni board è programmata per essere una specifica vetrina e il suo firmware è unico. Una configurazione di questo tipo può risultare scomoda quando si tratta di effettuare aggiornamenti OTA, perché, in tal caso, si renderebbe necessario un download diverso per ogni vetrina, una soluzione più ottimale sarebbe avere lo stesso firmware che si adatta in base ad un valore salvato in flash (che può essere configurato al primo avvio o programmato prima della distribuzione del dispositivo.
- Gli indirizzi virtuali per la eeprom sono definiti in db\_struct.h, l'indirizzo iniziale è del tutto arbitrario, da lì inizia il negozio, sono poi definiti i numeri di half-word necessari per contenere il negozio e ciascuna vetrina, con questi si compone l'indirizzo finale, dal quale si può capire lo spazio di indirizzi utilizzato.
- Nella flash i dati scritti devono essere allineati, il che vuol dire che possono essere scritte parole (32 bit), mezze parole e byte singoli, ma non è possibile scrivere una parola subito dopo un singolo byte, perché il suo allineamento prevede un indirizzo di partenza multiplo di 4 (per questa ragione le flag di inizio partizione occupano un'intera parola anzi che un singolo bit).
- 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 e non ne determina dunque la fine.
- Le query string utilizzate per le richieste al server sono nella forma mode=abc per poter occupare una dimensione fissa di 8 bit, necessaria alla funzione che compone il pacchetto http e non sempre ricavabile con strlen() (vedi sopra).
- Per ridurre la grandezza dei pacchetti è possibile eliminare o accorciare alcuni header, che comunque non vengono interpretati dalla board. Ad esempio all'interno degli script php si può rimuovere l'header X-Powered-By, che contiene dati sull'interprete utilizzato (qui php). Attraverso le configurazioni del server apache invece, è possibile ridurre i dati presentati nell'header Server alla sola dicitura Apache aggiungendo al file di configurazione (httpd.conf) la linea ServerTokens Prod.
- 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)
- L'operazione di jump del bootloader è delicata, è importante che l'indirizzo per lo stack pointer (\_\_set\_MSP()) sia messo all'inizio della partizione in flash, quello del program counter (chiamato attraverso la funzione Jump(), definita a partire dall'indirizzo) sarà invece impostato 4 byte dopo. [progetto bootloader]

• L'applicazione, che verrà flashata nella partizione dedicata, deve essere programmata adeguatamente, va quindi modificato il linker script STM32F429ZI\_FLASH.1d includendo una corretta ORIGIN per la flash e nel file Src/system\_stm32f4xx.c va settato l'offset per la vector table. Visto che quest'ultima viene inizializzata nell'applicazione, non c'è bisogno di impostarla anche nel bootloader, la flag SET\_VECTOR\_TABLE può dunque essere disabilitata (ripeterebbe l'inizializzazione di SCB->VTOR).

# 2 CHANGELOG DEI PROGETTI

# eth test

# v1.0.0 eth test

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

# 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

# ble merge

## 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 (EXTernal Interrupt)
- Impostazione come client ble (originariamente server)

# $v1.1.0\ bl\_merge$

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

# v1.1.1 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

## 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)

### v1.2.5 shop429

- Aggiustata la pagina per controllare il database del singolo negozio
- Implementate views per rendere accessibile ad ogni utente solo i dati di sua competenza
- Aggiustata la funzione Keep-alive per connessioni al server quando altri utenti sono connessi
- Il controllo di versione tiene conto anche dell'overflow su un byte

### v1.3.0 shop429

- Implementazione della eeprom simulata per salvare le tabelle
- Ad ogni boot vengono ripristinati i dati salvati in rom

### v1.3.1 shop429

- Inizio implementazione del download del firmware via ETH, problemi con pacchetti grandi
- Rimosso limite SSL\_MAX\_CONTENT\_LEN, la connessione va in timeout durante il flash

## v1.3.2 shop429

• Utilizzo di richieste separate al server per ottenere le varie sezioni del file

### v1.3.2 shop429

- Implementazione del trasferimento come *chunked* per sfruttare le potenzialità dell'http (singola richiesta)
- Aggiunto controllo di checksum per il firmware ricevuto

# $v1.3.3\ shop 429$

- Migliorata la gestione con costanti della flash
- $\bullet\,$  Corretta l'interpretazione della codifica chunked per resistere con dei firmware reali

# $v1.3.3 \ shop 429$

•