UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea Triennale in Ingegneria Elettronica

Implementazione della trasmissione tramite MQTT del dato acquisito da sensori con interfaccia CANBUS

Implementation of transmission via MQTT of data acquired from sensors with CANBUS interface

Candidato:
Ciotti Jacopo

Relatore:
Prof. Ciattaglia Gianluca

Correlatore:
Prof. Gambi Ennio

Anno Accademico 2022-2023

"È il lavoro mai incominciato che
impieghi più tempo a finire."
J.R.R. Tolkien

Indice

Introduzione **1**

1 Il protocollo CANBUS **2**
1.1 Introduzione al protocollo CANBUS 2
1.2 Architettura del protocollo 2
1.3 Trasmissione dati 4
1.4 Frame CANBUS 5
1.5 Varianti CANBUS: CAN 2.0A vs CAN 2.0B 7
1.6 Applicazioni pratiche del protocollo: CANBUS in ambito automobilistico 8
1.7 Considerazioni finali 9

2 Il protocollo MQTT **11**
2.1 Introduzione al protocollo MQTT 11
2.2 Principi di comunicazione del protocollo MQTT 11
2.2.1 Architettura di una comunicazione MQTT 11
2.2.2 Modello Publish/Subscribe 12
2.2.3 QoS (Quality of Service) in MQTT 13
2.3 Considerazioni finali 14

3 Hardware e Software utilizzato **15**
3.1 Arduino 15
3.1.1 Arduino Uno 15
3.1.2 Arduino Due 16
3.2 ADXL345 16
3.3 Termistore 18
3.4 CAN-BUS Shield 19
3.4.1 Seeed Studio CAN-BUS Shield v1.0 19
3.4.2 Sparkfun CAN-BUS Shield 20
3.5 WiFi Shield 21
3.5.1 Configurazione shield WiFi 22

4 Descrizione del sistema implementato **23**
4.1 Schema generale del sistema 23
4.2 Comunicazione tra ADXL345 e Arduino 24
4.2.1 Configurazione ADXL345 26
4.2.2 Lettura dei dati dall'ADXL345 29

4.3 Comunicazione tra Arduino e il termistore 31

5 Integrazione con l'interfaccia CANBUS 33

5.1 Configurazione dei nodi CANBUS 33

5.2 Implementazione della trasmissione dati tramite CANBUS 35

6 Implementazione trasmissione MQTT su Cloud IoT 38

6.1 Introduzione Cloud IoT 38

6.2 Scelta del broker MQTT 39

6.3 Implementazione MQTT su ThingSpeak 41

7 Problematiche riscontrate e soluzioni implementate 44

7.1 Incompatibilità tra Arduino Due e lo shield CAN di Seed Studio . . 44

7.2 Problemi con l'architettura di Arduino Due e la libreria dello shield CAN di SparkFun 44

7.3 Disponibilità dei Socket sullo shield WiFi 45

7.4 Intervallo di pubblicazione sul cloud IoT 46

7.5 Considerazioni finali 46

8 Conclusioni 48

Bibliografia 49

Appendice : Codice Arduino 51

1 Codice Trasmettitore 51

2 Codice Ricevitore 57

Introduzione

Negli ultimi decenni, l'evoluzione delle tecnologie di comunicazione e di acquisizione dati ha rivoluzionato numerosi settori, dall'industria all'ambito domestico. In particolare, la crescente diffusione di reti di sensori e l'adozione sempre più ampia di bus di comunicazione come il Controller Area Network (CAN) hanno aperto nuove prospettive per il monitoraggio e la gestione dei sistemi complessi.

Il presente lavoro di tesi si concentra sull'implementazione della trasmissione di dati acquisiti da un sensore digitale e da uno analogico attraverso il protocollo di messaggistica MQTT (Message Queuing Telemetry Transport), con particolare attenzione all'interfaccia CANBUS.

L'obiettivo principale è, infatti, esplorare le potenzialità e le sfide legate all'integrazione delle due tecnologie: la comunicazione tramite CANBUS e la trasmissione di dati attraverso il protocollo MQTT. La combinazione di queste due tecnologie offre una soluzione versatile e efficiente per la trasmissione di dati in tempo reale provenienti da sensori distribuiti su una rete CANBUS.

In particolare questa tesi include la progettazione e l'implementazione di un sistema che acquisisce i valori di accelerazione dal sensore ADXL345 e i valori di temperatura da un termistore, per poi essere trasmessi ad un Arduino. Quest'ultimo, mediante il bus seriale CAN, li invia ad un altro nodo, nel nostro caso un altro Arduino, che li trasmette tramite il protocollo MQTT ad un cloud IoT, dove saranno disponibili all'utente.

Nei primi capitoli saranno esaminati gli aspetti teorici delle tecnologie CANBUS e MQTT, concentrandosi sugli elementi essenziali che le caratterizzano. Verranno descritte le caratteristiche principali dei due protocolli, i principi base di comunicazione, le loro architetture e i vari modelli di comunicazione.

Nei capitoli successivi, saranno, invece, esaminati gli aspetti pratici della progettazione del sistema. Si partirà dalla scelta dei dispositivi hardware e software, per poi procedere alla descrizione dettagliata, attraverso l'ausilio di schemi e codice, dell'implementazione e configurazione dei vari nodi del sistema.

Nei capitoli conclusivi verranno esaminate le criticità emerse durante lo sviluppo del progetto e le soluzioni adottate, fornendo così una sintesi completa del lavoro svolto.

Capitolo 1

Il protocollo CANBUS

1.1 Introduzione al protocollo CANBUS

Il CAN (*Controller Area Network)*, chiamto anche CANBUS, è uno standard seriale per bus, ideato negli anni '80 dalla Robert Bosh GmbH, con lo scopo di consentire la comunicazione tra dispositivi. Si tratta di un protocollo basato su messaggi, progettato originariamente per l'utilizzo in ambiente automotive, ma che può essere utilizzato anche in molti altri contesti, come ad esempio applicazioni industriali di tipo embedded. Per ciascun dispositivo, i dati in un frame vengono trasmessi in serie ma in modo tale che se più di un dispositivo trasmette contemporaneamente, il dispositivo con la priorità più alta può continuare mentre gli altri si ritirano. I frame vengono ricevuti da tutti i dispositivi, ma il dispositivo trasmittente non riceve una conferma diretta della propria trasmissione.

In particolare CAN è un sistema di bus seriale con funzionalità multi-master; cioè, tutti i nodi sono in grado di trasmettere dati e multipli nodi possono richiedere l'accesso al bus simultaneamente. Nella rete CAN non c'è bisogno di indirizzamento dei nodi, ma vengono trasmessi messaggi prioritari. Un trasmettitore trasmette un messaggio a tutti i nodi CAN dopodiché ciascun nodo decide, sulla base dell'identificatore ricevuto, se deve elaborare il messaggio o meno. L'identificatore determina anche la priorità del messaggio nell'accesso al bus. Questa forma di gestione del bus è chiamato Carrier Sense Multiple Access with Collision Detection (CSMA/CD).

1.2 Architettura del protocollo

L'architettura di un sistema CAN è tipicamente organizzata in una topologia a bus, in cui diversi dispositivi possono comunicare tra loro attraverso un singolo bus fisico (Figura 1.1). I due elementi chiave sono:

- *Bus CAN*: si tratta del mezzo fisico attraverso il quale i dispositivi comunicano. Solitamente formato da due fili chiamati CANH (CAN high) e CANL (CAN low) e terminato con un'impedenza caratteristica di $120\,\Omega$.

- *Nodo CAN*: Qualunque dispositivo che partecipa alla comunicazione sul bus CAN (Figura 1.2). Un nodo può essere un componente elettronico come sensori, attuatori, microcontrollori, unità di controllo elettroniche (ECU), ecc...

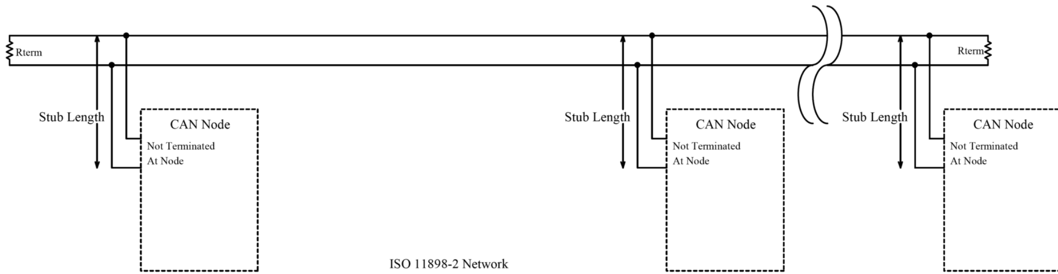

Figura 1.1: Esempio di un'architettura CAN

Per poter comunicare sono necessari due o più nodi sulla rete CAN. In particolare ogni nodo necessita di:

- *CPU*: si occupa di stabilire il significato dei messaggi ricevuti e quali messaggi devono essere trasmessi.
- *CAN-controller*: spesso parte integrante di un microcontrollore.
  - Ricezione: il controller CAN memorizza i bit seriali ricevuti dal bus finchè non costruisce un intero messaggio.
  - Trasmissione: il processore invia i messaggi da trasmettere al controller CAN, che trasmette i bit in maniera seriale sul bus quando è libero.
- *Transceiver*: si occupa di tradurre i segnali di livello logico generati dal CAN controller in segnali fisici adatti per essere trasmessi sul bus fisico CAN e viceversa. In altre parole agisce come interfaccia tra il controller CAN e il bus fisico.

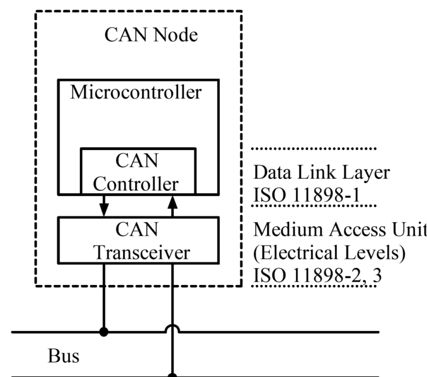

Figura 1.2: Nodo CAN

1.3 Trasmissione dati

Il CAN trasmette dati secondo un modello basato su bit "dominanti" e "recessivi", dove il bit "dominante" è uno 0 logico, mentre il bit "recessivo" è un 1 logico. Lo stato inattivo è rappresentato dal livello recessivo.

Quando il nodo CAN trasmette un segnale dominante (0), il filo CANH viene portato ad un livello di tensione alto (vicino a VCC), mentre il filo CANL viene portato ad un livello di tensione basso (vicino a GND). Quando viene trasmesso il segnale recessivo (1), i fili CANH e CANL si trovano a livelli di tensione simili. Un esempio di tale funzionamento è mostrato in Figura 1.3.

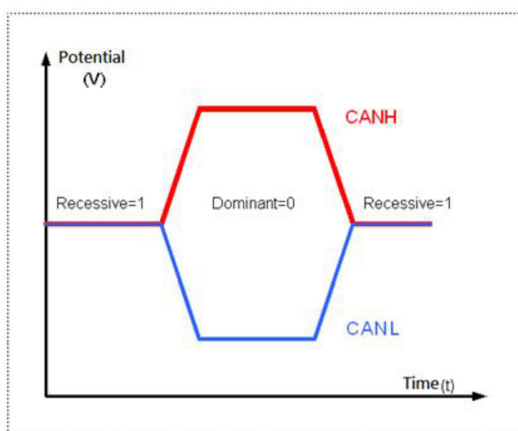

Figura 1.3: Aspetto del segnale di livello fisico

Se un nodo trasmette un bit dominante ed un altro nodo trasmette un bit recessivo si verifica una collisione e prevale il bit dominante (AND logico). Ciò significa che non vi è alcun ritardo nel messaggio con priorità più alta e il nodo che trasmette il messaggio con priorità più bassa tenta automaticamente di ritrasmettere sei tempi di bit dopo la fine del messaggio dominante. Ciò rende CAN molto adatto come sistema di comunicazione con priorità in tempo reale.

Le tensioni esatte per uno 0 o 1 logico dipendono dal livello fisico utilizzato, ma il principio di base del CAN richiede che ciascun nodo ascolti i dati sulla linea CAN, inclusi i nodi trasmittenti stessi.

Quando un nodo trasmette un 1 logico ma vede uno 0 logico, si accorge che c'è un conflitto e smette di trasmettere. Utilizzando questo processo, qualsiasi nodo che trasmette un 1 logico, quando un altro nodo trasmette uno 0 logico, perde la priorità e si ritira. Un nodo che perde la priorità rimette in coda il suo messaggio per una trasmissione successiva e il flusso di bit del frame CAN continua senza errori finché non rimane un solo nodo in trasmissione. Ciò significa che il nodo che trasmette il primo 1 perde la priorità.

Poiché l'identificatore a 11 bit viene trasmesso da tutti i nodi all'inizio del frame CAN, il nodo con l'identificatore più basso trasmette più zeri all'inizio del frame, e questo è il nodo che vince e ha la massima priorità, come mostrato in Figura 1.4.

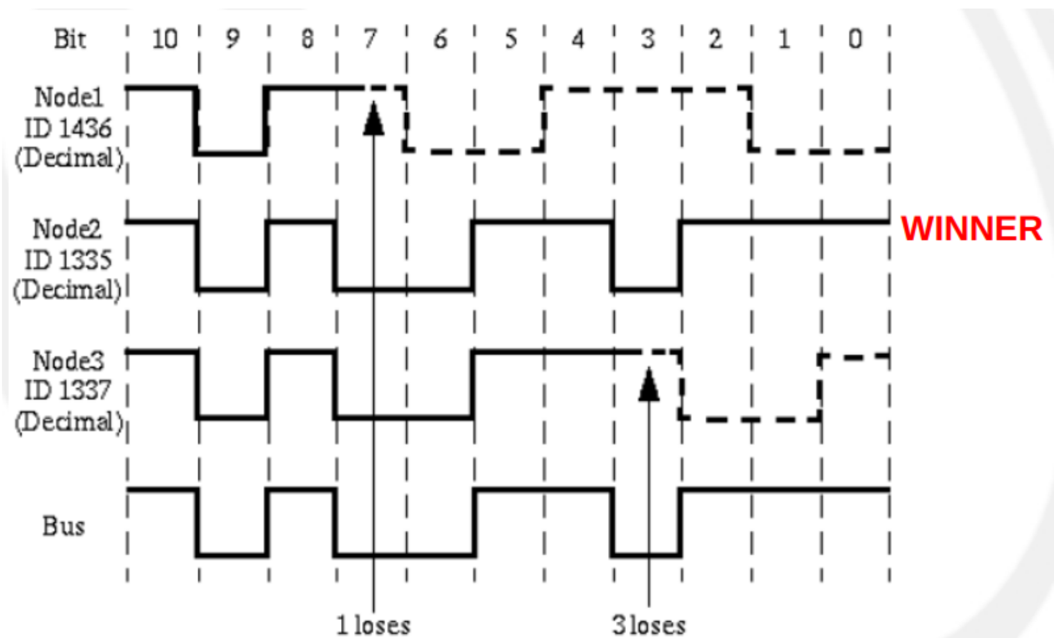

Figura 1.4: Esempio arbitraggio canale

1.4 Frame CANBUS

Tutti i frame (messaggi) iniziano con un bit SOF (Start Of Frame). Il protocollo CAN prevede 4 tipi di frame:

- *Data Frame*: contiene i dati trasmessi da un nodo.
- *Remote Frame*: frame che richiede la trasmissione di un determinato identificatore.
- *Error Frame*: trasmesso da un nodo che ha rilevato un errore.
- *Overload Frame*: frame per inserire un ritardo tra il data frame e il remote frame.

Data Frame

Un frame standard CAN, il cui formato è illustrato in Figura 1.5, consiste dei seguenti campi:

- SOF: bit dominante (0 logico) che indica l'inizio di un frame.
- Arbitration Field: contenente 11 bit di identificatore.
- Remote Transmission Request (RTR) bit: 0 = Data Frame, 1 = Remote Frame.
- Control Field: 6 bit di cui 2 bit riservati per usi futuri e 4 bit di Data Lenght Code (DLC) che indica il numero dei byte nel Data Field.

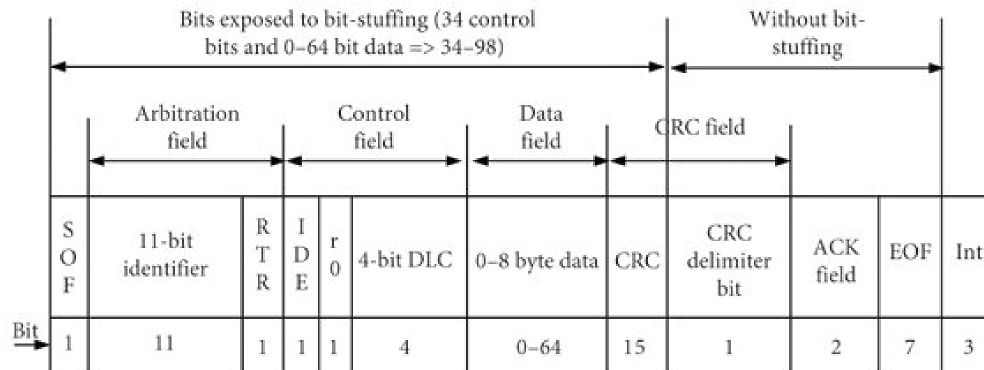

Figura 1.5: Data Frame

- Data Field: può variare da 0 a 8 byte.
- CRC Field: 15 bit di cyclic redundancy check code + 1 bit recessivo come delimitatore.
- Ack: 2 bit di cui il primo è lo Slot Ack che è posto a recessivo da TX ma è sovrascritto con un dominante da ogni stazione che riceve correttamente il messaggio mentre il secondo bit è recessivo e svolge il compito di delimitatore.
- End Of Frame (EOF): 7 bit di valore recessivo.

Remote Frame

Il Remote Frame è identico al Data Frame, eccetto che il bit RTR viene trasmesso come bit recessivo ed in secondo luogo il Remote Frame non contiene nessun campo dati.

Nel caso in cui un frame di dati e un frame remoto con lo stesso identificatore vengano trasmessi contemporaneamente, il frame di dati vince l'arbitraggio a causa del bit RTR dominante che segue l'identificatore.

Error Frame

Un Error Frame è costituito da:

- Error Flag lungo almeno 6 bit recessivi in modo che tutte le altre stazioni rilevino un errore e spediscano anch'esse un Error Flag. Per questo motivo il campo Error Flag nel pacchetto è di lunghezza variabile (max 12 bit) dato dalla sovrapposizione di tutti gli Error Flag spediti.
- Error Delimiter costituito da 8 bit recessivi.

Overload Frame

È molto simile ad un Error Frame, ma consiste di un Overload Flag e di un Overload Delimiter.

Può essere generato da due condizioni:

- Quando un ricevitore ha bisogno di più tempo per processare i dati correnti prima che altri vengano ricevuti.
- Il rilevamento di un bit dominante durante l'Intermission Field (Separa tra loro Data/Remote Frame dai frame precedenti).

1.5 Varianti CANBUS: CAN 2.0A vs CAN 2.0B

Come già detto CANBUS nasce negli anni '80, ma nel 1991 BOSCH rilascia la versione 2.0 che a sua volta si divide in due parti, CAN 2.0A e CAN 2.0B. Le principali differenze tra le due versioni sono:

- Formato del frame:
  - CAN 2.0A: formato di frame standard con il campo identificazione lungo 11 bit.
  - CAN 2.0B: formato di frame esteso con il campo identificazione lungo 29 bit.
- Campo Identificazione:
  - CAN 2.0A: Campo identificazione lungo 11 bit, consente massimo 2048 identificatori univoci.
  - CAN 2.0B: Campo identificazione più lungo (29 bit), consente oltre 500 milioni identificatori univoci.
- Gestione errori:
  - CAN 2.0A: la gestione degli errori non è avanzata. In caso di errore i nodi ritrasmettono i messaggi senza rilevare la natura dell'errore.
  - CAN 2.0B: introduce una gestione migliorata degli errori permettendo maggiore precisione nella rilevazione e segnalazione degli errori.

(a)

(b)

Figura 1.6: CAN 2.0A vs CAN 2.0B

1.6 Applicazioni pratiche del protocollo: CANBUS in ambito automobilistico

Una delle maggiori applicazioni del protocollo CAN è il settore automobilistico. Esso viene infatti utilizzato per facilitare la comunicazione tra i vari sistemi elettronici presenti in un veicolo.

La rete di un automobile deve infatti accumulare quanti più dati possibili durante il movimento per poter funzionare in real-time, in modo tale che possa rispondere preparata ad eventuali influenze esterne. Per realizzare uno scenario del genere c'è bisogno di sensori, telecamere, ed ulteriori dispositivi elettronici che permettono di raccogliere i dati richiesti. Questi dati devono essere trasferiti ad un'unità centrale di controllo attraverso un bus che connette tutti i dispositivi in un unico sistema.

È qui che entra in gioco il protocollo CAN che permette di utilizzare un unico bus per più connessioni, al fine di limitare al minimo la quantità di cavi all'interno del veicolo. L'elemento fondamentale della rete di un'automobile è l'ECU (Electronic Control Unit), che si occupa di monitorare i vari sistemi elettrici. All'interno del veicolo generalmente sono presenti diverse ECU, ciascuna con un compito diverso, come ad esempio l'unità di controllo del motore o l'unità di controllo della velocità. Tutte le ECU sono collegate al bus CAN che permette loro di essere collegate all'unità centrale di controllo per garantire le funzionalità del sistema. I dati vengono infatti inviati sul bus CAN per essere trasportati all'unità centrale di controllo e confrontati con valori specifici. Se si verifica una discrepanza tra i valori, la centrale utilizzerà gli attuatori collegati per regolare fisicamente questa variazione di valori. La complessità dell'unità centrale di controllo dipende dal campo di applicazione. Le unità più elementari sono costituite da un solo chip con un microcontrollore. Le moderne unità di controllo sono spesso sistemi multiprocessore con un'interfaccia grafica utente.

Nonostante l'avvento di nuove tecnologie il protocollo CAN viene tutt'oggi ampiamente utilizzato in questo settore soprattutto grazie ai suoi bassi costi, alla semplicità di implementazione, alla facilità di gestione del protocollo, ma soprattutto

perché consente di rispondere a processi asincroni in pochi millisecondi, rendendo la comunicazione quasi real-time.

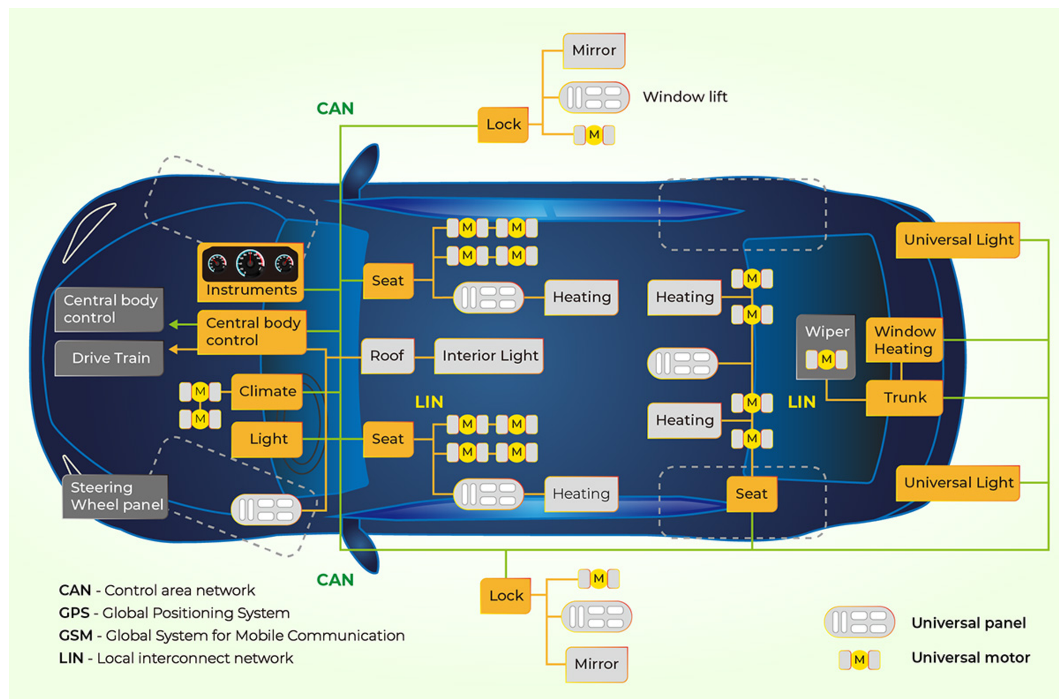

Figura 1.7: Esempio di rete CANBUS all'interno di un automobile

1.7 Considerazioni finali

Il protocollo CAN è stato ed è tuttora fondamentale nello sviluppo di sistemi di comunicazione all'interno dei veicoli ed in altri contesti industriali. La sua enorme diffusione è dovuta alla sua semplicità e flessibilità: i nodi, infatti, non hanno bisogno di un indirizzo che li identifichi, semplificando dunque la loro aggiunta e rimozione senza necessità di riorganizzazione del sistema. Un altro suo punto a favore è l'elevata immunità ai disturbi, consentendo ai dispositivi di comunicare anche in condizioni estreme come l'interruzione di uno dei due fili. Da sottolineare è anche la sua affidabilità nella gestione degli errori; ciascun nodo è in grado di rilevare il proprio malfunzionamento e autoescludersi dal bus in caso di problemi persistenti.

Tuttavia è importante notare che il protocollo CAN presenta alcune limitazioni in termini di velocità di trasmissione dati e dimensioni dati, specialmente confrontato con tecnologie moderne.

La velocità di trasferimento dati del protocollo CAN è, infatti, tipicamente dell'ordine dei kilobit al secondo (Kbps), velocità modeste se comparate ad altre tecnologie utilizzate nell'ambiente automotive come Ethernet che permette di raggiungere velocità dell'ordine dei gigabit al secondo (Gbps), per applicazioni come l'infotainment avanzato e le comunicazioni veicolo-veicolo (V2V), o FlexRay che può supportare

velocità di trasferimento dati di diversi megabit al secondo (Mbps), rendendolo adatto in applicazioni ad alta larghezza di banda come quelli utilizzati nei sistemi avanzati di assistenza alla guida (ADAS).

Nonostante ciò, l'elevata affidabilità e adattabilità del protocollo CAN, continuano a renderlo una scelta diffusa e affidabile in numerosi contesti, confermando il suo ruolo chiave nell'evoluzione dei sistemi di comunicazione industriali e automobilistici.

Capitolo 2

Il protocollo MQTT

2.1 Introduzione al protocollo MQTT

Il protocollo MQTT (*Message Queuing Telemetry Transport*) è un protocollo di messaggistica leggero, progettato da IBM nel 1999 per la comunicazione efficiente tra dispositivi con restrizioni di risorse, come sensori e attuatori.

Creato per garantire una trasmissione di messaggi affidabile e veloce, MQTT si basa su un modello di *publish/subscribe*, consentendo ai dispositivi di inviare e ricevere dati in modo asincrono. La sua flessibilità e la bassa latenza lo rendono ampiamente adottato nelle applicazioni IoT (Internet of Things).

2.2 Principi di comunicazione del protocollo MQTT

2.2.1 Architettura di una comunicazione MQTT

Una tipica architettura MQTT può essere divisa in due componenti principali come mostrato in Figura 2.1. Ciascun componente è brevemente descritto di seguito:

- *Client MQTT*: è un qualsiasi dispositivo, come un server o un microcontrollore, che stabilisce la connessione con il server (*Broker*). Un client MQTT può essere un *Publisher* o un *Subscriber*.

- *Broker MQTT*: il broker MQTT è il server che coordina i messaggi tra i diversi client. È inoltre responsabile della ricezione dei messaggi da parte dei *Publisher* e del filtraggio di essi per essere infine inviati ai client iscritti (*Subscriber*).

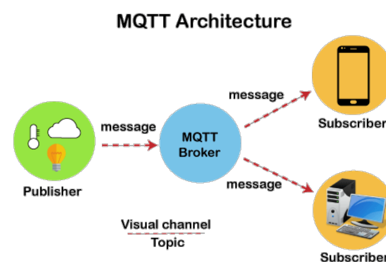

Figura 2.1: Architettura MQTT

L'avvio della comunicazione tra i client e il broker avviene attraverso una connessione MQTT (Figura 2.2). Inizialmente, i client instaurano la connessione inviando un messaggio di tipo CONNECT al broker MQTT, il quale risponde confermando l'avvenuta connessione mediante un messaggio CONNACK. Questa comunicazione richiede l'impiego di uno stack TCP/IP sia da parte dei client MQTT che del broker.

È importante notare che i client non si collegano direttamente tra loro, ma stabiliscono connessioni esclusivamente con il broker, il quale sarà poi responsabile dell'inoltro dei messaggi verso i rispettivi client iscritti.

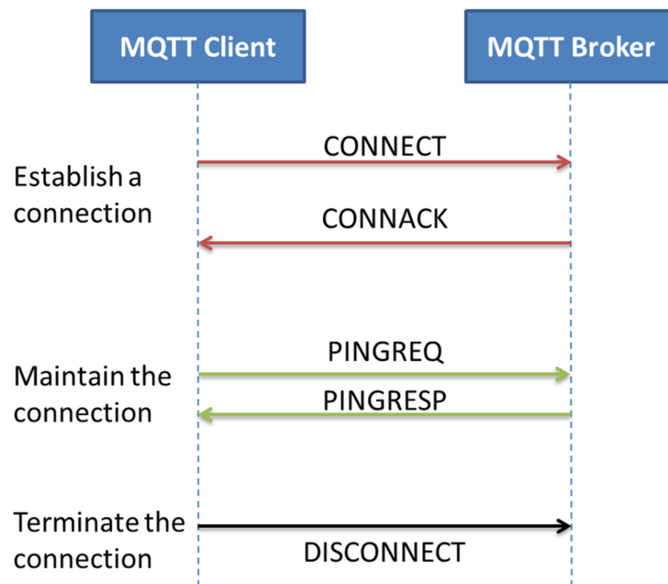

Figura 2.2: Comunicazione MQTT

2.2.2 Modello Publish/Subscribe

Nel modello *Publish/Subscribe*, i client che pubblicano un messaggio vengono disaccoppiati temporalmente da quelli che ricevono il messaggio. I client operano in isolamento l'uno dall'altro, infatti non sanno dell'esistenza degli altri client. Un client può pubblicare messaggi di un tipo specifico, e solo i client interessati a quel tipo di messaggio saranno destinatari delle pubblicazioni corrispondenti.

Il modello *Publish/Subscribe* richiede un server (Broker) per poter funzionare. Tutti i client stabiliscono una connessione con il broker. Il client che invia un messaggio verso il broker è chiamato *Publisher*.

Il broker filtra i messaggi in arrivo e li distribuisce ai client interessati al tipo di messaggio ricevuto.

I client che si iscrivono sul broker perché interessati a tipi specifici di messaggi sono chiamati *Subsciber*.

I client comunicano con il broker attraverso l'invio di messaggi; ogni messaggio contiene un topic, organizzato in una struttura ad albero, al quale i client possono iscriversi o pubblicare. Il broker riceve messaggi pubblicati dai client che contengono il topic e un determinato valore o comando e trasmette le informazioni a ogni client che si è iscritto a quell'argomento specifico. Un esempio è mostrato in Figura 2.3, in cui i client a sinistra inviano messaggi con diversi topic (*"temp"*, *"moisture"*, *"accel-x"*) e diversi valori verso il broker. I client di destra iscritti ai topic *"temp"* e *"accel-x"* riceveranno rispettivamente i valori pubblicati (30°C, 1.8g).

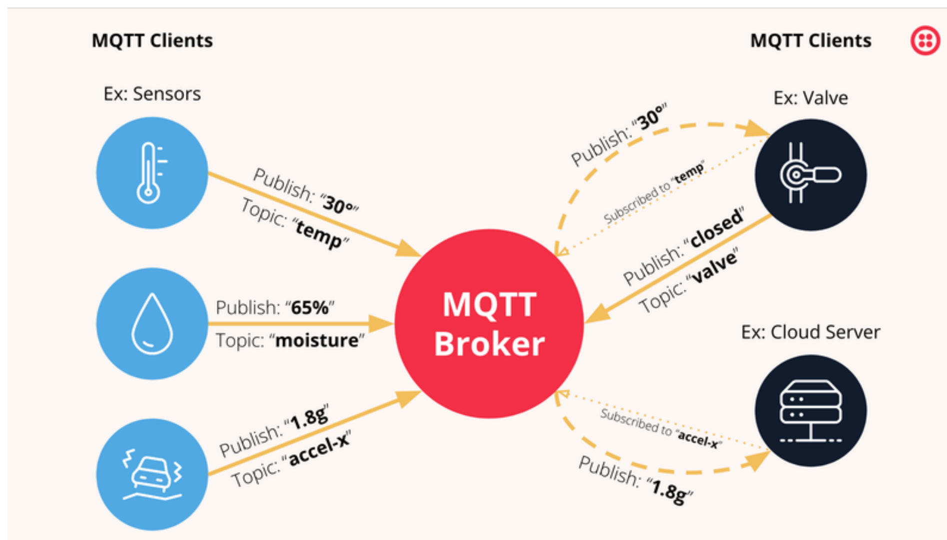

Figura 2.3: Modello publish/subscribe

2.2.3 QoS (Quality of Service) in MQTT

La Quality of Service (QoS) in MQTT, rappresenta un aspetto fondamentale del protocollo e definisce il livello di affidabilità nella consegna dei messaggi tra i client e i broker MQTT. Il livello di QoS è l'accordo tra un mittente e un destinatario di un messaggio sulle garanzie di effettiva consegna del messaggio. Queste garanzie potrebbero includere il numero di volte in cui un messaggio potrebbe arrivare e la presenza o meno di duplicati.

MQTT offre tre livelli di QoS per soddisfare diverse esigenze di affidabilità e tempi di consegna. Ecco una panoramica dei tre livelli di QoS in MQTT:

- *QoS 0, At most once*: questo livello di QoS offre una consegna dei messaggi con un'affidabilità "al massimo una volta". Il mittente invia semplicemente il messaggio alla destinazione senza richiedere una conferma di ricezione. Non c'è garanzia che il messaggio venga consegnato con successo e potrebbe persino essere duplicato. Il vantaggio principale di questo livello di QoS è che presenta il minor overhead possibile rispetto ad altri livelli di QoS. Questo livello è adatto a situazioni in cui la perdita di alcuni messaggi non è critica.

- *QoS 1, At least once*: questo livello QoS aggiunge un requisito di conferma (ACK) alla destinazione che deve ricevere il messaggio. In questo modo, questo livello di QoS fornisce la garanzia che il messaggio verrà consegnato almeno una volta all'abbonato. Uno dei principali svantaggi di questo livello di QoS è che può generare duplicati, ovvero lo stesso messaggio può essere inviato più di una volta alla stessa destinazione. Il mittente memorizza il messaggio finché non riceve una conferma dall'abbonato. Nel caso in cui il mittente non riceva la conferma entro un tempo specifico, il mittente ripubblicherà il messaggio al destinatario. Il destinatario finale deve disporre della logica necessaria per rilevare i duplicati nel caso in cui non sia necessario elaborarli due volte.
- *QoS 2, Exactly once*: questo livello di QoS offre il massimo livello di affidabilità. Esso garantisce che il messaggio venga consegnato solo una volta alla destinazione. Il livello QoS 2 presenta l'overhead più elevato rispetto agli altri livelli QoS. Un messaggio pubblicato con QoS livello 2 viene considerato consegnato con successo dopo che il mittente è sicuro di essere stato ricevuto con successo una sola volta dalla destinazione.

È molto importante capire che possiamo pubblicare con un livello di QoS e iscriverci con un altro livello di QoS. Pertanto, esiste un livello di QoS per il processo di pubblicazione tra il publisher e il server MQTT e un altro livello di QoS per il processo di pubblicazione tra il server MQTT e il subscriber.

Se un publisher lavora con un livello QoS superiore al livello QoS specificato dal subscriber, il server MQTT dovrà ridurre il livello QoS al livello più basso utilizzato dal subscriber specifico quando pubblica il messaggio dal server MQTT a questo subscriber. Ad esempio, se utilizziamo il livello QoS 2 per pubblicare un messaggio dal publisher sul server MQTT ma un subscriber ha richiesto il livello QoS 1 durante l'esecuzione del servizio di iscrizione, la pubblicazione dal server MQTT a questo subscriber utilizzerà il livello QoS 1.

2.3 Considerazioni finali

MQTT si conferma dunque uno dei fondamentali standard di comunicazione nel settore IoT (*Internet of Things*). Come già accennato la sua leggerezza, flessibilità e scalabilità lo rendono ideale per applicazioni in cui risorse come potenza, capacità di calcolo, memoria e banda sono limitate. L'architettura di tipo Publish/Subscribe di MQTT consente una comunicazione asincrona, riducendo la complessità delle connessioni dirette e facilitando l'integrazione di dispositivi diversi. La varietà e flessibilità dei livelli di Quality of Service (QoS) offre un bilanciamento tra l'affidabilità della consegna dei messaggi e la velocità di trasmissione, consentendo di adattare il protocollo alle specifiche esigenze dell'applicazione.

Capitolo 3

Hardware e Software utilizzato

Nel corso di questo capitolo verranno elencati i dispositivi hardware e software impiegati nell'implementazione del sistema in esame.

3.1 Arduino

Per la realizzazione dell'architettura a microcontrollori in esame, si è optato per l'impiego delle schede Arduino, note per la loro versatilità e facilità d'uso. La scelta è motivata non solo dalla robustezza di tali dispositivi, ma anche dalla vasta disponibilità di librerie che coprono le nostre esigenze di studio. La scelta del software, conseguentemente, si è indirizzata verso l'utilizzo di Arduino IDE, che si configura la piattaforma ideale per lo sviluppo del nostro progetto.

3.1.1 Arduino Uno

Arduino Uno (Figura 3.1) è una scheda microcontrollore basata sull'ATmega328P. Dispone di 14 pin di ingresso/uscita digitali (di cui 6 utilizzabili come uscite PWM), 6 ingressi analogici, un clock da 16 MHz (CSTCE16M0V53-R0), una connessione USB, un jack di alimentazione, un header ICSP e un pulsante di reset.

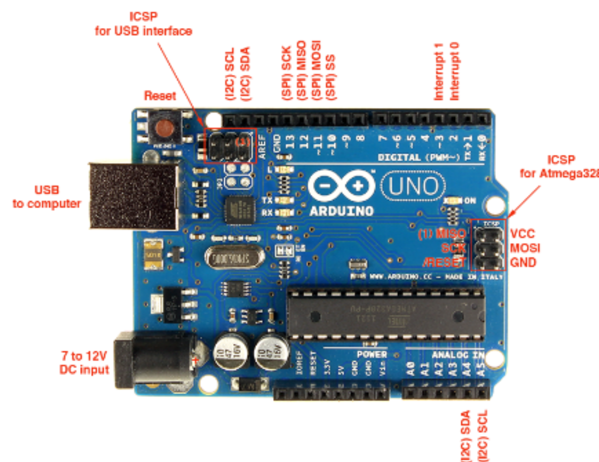

Figura 3.1: Arduino Uno

3.1.2 Arduino Due

Arduino Due (Figura 3.2) è una scheda microcontrollore basata sulla CPU Atmel SAM3X8E ARM Cortex-M3. È la prima scheda Arduino basata su un microcontrollore ARM core a 32 bit. La comunicazione avviene tramite due porte micro Usb, di cui una (Native USB) connessa direttamente al microcontrollore SAM3X MCU, mentre l'altra (Programming USB), con un throughtput minore, è connessa a un convertitore da protocollo USB a Seriale. Dispone di 54 pin di ingresso/uscita digitali (di cui 12 utilizzabili come uscite PWM), 12 ingressi analogici, 4 UART (porte seriali hardware), un clock da 84 MHz, una connessione USB OTG, 2 DAC (da digitale ad analogico) , 2 TWI, un jack di alimentazione, un'header SPI, un'intestazione JTAG, un pulsante di ripristino e un pulsante di cancellazione. A differenza della maggior parte delle schede Arduino, la scheda Arduino Due funziona a 3,3 V. La tensione massima che i pin I/O possono tollerare è 3,3 V. L'applicazione di tensioni superiori a 3,3 V a qualsiasi pin I/O porterebbe a danneggiare la scheda.

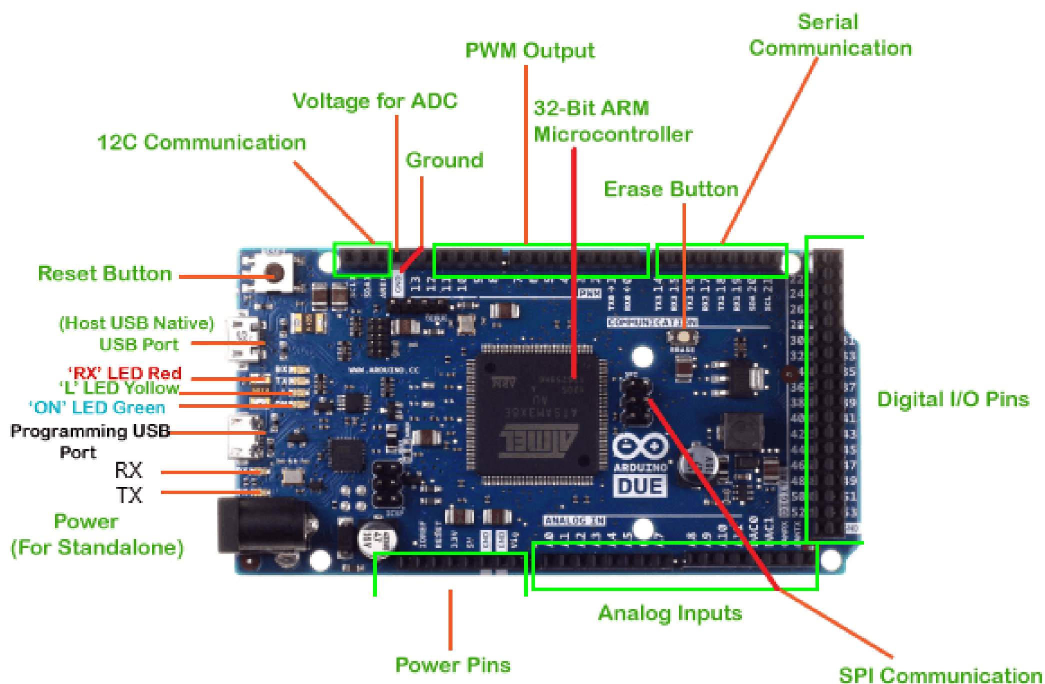

Figura 3.2: Arduino Due

3.2 ADXL345

ADXL345 (Figura 3.3) è un accelerometro digitale a 3 assi con un range massimo di $\pm 16g$, risoluzione a 13 bit e larghezza di banda massima di 3200 Hz. I dati in uscita sono formattati come complemento a due a 16 bit, accessibili tramite interfacce

SPI (a 3 o 4 fili) o I2C. Questo sensore ha un consumo energetico estremamente basso e consuma 23 μA in modalità misurazione e 0.1 μA in modalità standby.

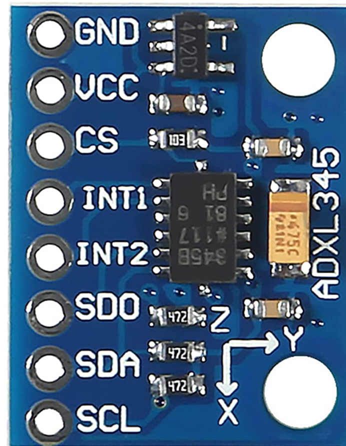

Figura 3.3: ADXL345

Alcune specifiche:

- *Range di misurazione*: ADXL345 può misurare l'accelerazione dei 3 assi con un range selezionabile di $\pm 2g$, $\pm 4g$, $\pm 8g$ e $\pm 16g$. Con un range di misurazione di $\pm 2g$ il sensore è in grado di misurare fino a 19.6 $\frac{m}{s^2}$ ($2 \cdot 9.8$ $\frac{m}{s}$) in tutte e 3 le direzioni lungo gli assi. Con un range di misurazione di $\pm 16g$ il sensore è in grado di misurare fino a 153.6 $\frac{m}{s^2}$ ($16 \cdot 9.8$ $\frac{m}{s}$) in tutte e 3 le direzioni lungo gli assi.

- *Risoluzione d'uscita*: ADXL345 supporta una risoluzione d'uscita a 10 bit per il range di misurazione di $\pm 2g$, 11 bit per $\pm 4g$, 12 bit per $\pm 8g$, 13 bit per $\pm 16g$. La risoluzione di default è 10 bit per tutti i range di misurazione.

- *Sensibilità*: Con la risoluzione standard a 10 bit ADXL345 ha una sensibilità di 3.9 mg/LSB per il range $\pm 2g$, 7.8 mg/LSB per $\pm 4g$, 15.6 mg/LSB per $\pm 8g$ e 31.2 mg/LSB per $\pm 16g$.

- *Data rate in uscita e larghezza di banda*: Sono entrambi selezionabili, da un range di 0.1 Hz fino a 3200 Hz.

- *Tensione di lavoro*: Il sensore necessita di una tensione di lavoro di 2.5 V che può andare da 2.0 V a 3.6 V.

- *Temperatura di lavoro*: ADXL345 ha un range di temperatura di lavoro di -40°C fino +85°C.

- *Valori massimi*: ADXL345 può tollerare un accelerazione massima di 10000g (98 $\frac{km}{s^2}$). Può tollerare una tensione massima di 3.9 V e una temperatura massima di +105°C.

In Figura 3.3 si può notare un modulo ADXL345 in cui non si hanno disponibili tutti i pin, ma solo quelli essenziali per l'interfacciamento con un circuito. Essi sono:

- *GND*: Ground.
- *VCC*: Tensione di alimentazione.
- *CS*: Chip Select, utilizzato per selezionare il dispositivo nella comunicazione SPI.
- *INT1*: Interrupt 1 Output.
- *INT2*: Interrupt 2 Output.
- *SDO*: Serial Data Output (SPI 4 fili)/ Address Select (I2C).
- *SDA*: Serial Data (I2C)/ Serial Data Input (SPI 4 fili)/ Serial Data Input Output (SPI 3 fili).
- *SCL*: Serial CLock.

3.3 Termistore

Un sensore termistore o sonda NTC (Negative Temperature Coefficient) è un resistore il cui valore di resistenza varia con la temperatura (Figura 3.4). Esso è composto da un materiale semiconduttore sintetizzato che, in risposta a una piccola variazione della temperatura, mostra una variazione resistiva. Il sensore possiede un coefficiente di temperatura negativo, facendo si che l'aumento di temperatura provoca la diminuzione in modo esponenziale del valore di resistenza.

Di seguito alcune caratterisriche tecniche:

- Elemento sensibile:
  - Pt100 Ω @ 0°C
  - NTC R(25°C)=10 KΩ ±1%, beta(25/85)=3977
  - NTC R(25°C)=10 KΩ ±1%, beta(25/85)=3435
  - NTC R(25°C)=10 KΩ ±3%, beta(25/85)=3977
  - NTC R(25°C)=2.7 KΩ ±1%, beta(25/85)=3977
  - NTC R(25°C)=10 KΩ, beta (25/85) =3969, tol ±0.2°C (0-70°C)
- Configurazione: 2 fili
- Campo temperatura di funzionamento: $-50 \div 150$ °C

- Resistenza di isolamento: 100 MΩ, 1000 VCC
- Rigidità dielettrica: 3750 Vac

Figura 3.4: Sonda NTC a 2 fili

3.4 CAN-BUS Shield

Al fine di stabilire la comunicazione tra le due schede Arduino è stato scelto di adottare il protocollo CAN. Tuttavia, poiché entrambe le schede non supportano nativamente questo tipo di comunicazione, sono stati impiegati due shield CAN bus. Uno "shield" è una scheda di espansione che di solito viene inserita sopra l'Arduino, ma che può anche essere collegata separatamente, al fine di aumentare le funzionalità della scheda. Per ragioni di disponibilità sono stati utilizzati due shield provenienti da case produttrici differenti: il CAN bus shield di Seeed Studio (Figura 3.5) e quello di Sparkfun (Figura 3.6).

3.4.1 Seeed Studio CAN-BUS Shield v1.0

Questo shield CAN bus adotta il controller CAN MCP2515 con interfaccia SPI e il transceiver CAN MCP2551, per fornire funzionalità CAN all'Arduino. Alcune caratteristiche:

- implementa CAN 2.0B con velocità fino a 1 Mb/s
- interfaccia SPI fino a 10 MHz
- Standard (11 bit) e Extended (29 bit) Data e Remote frame
- connessione CAN tramite il connettore DB-9
- 2 buffer in ricezione con memorizzazione dei messaggi prioritari
- inidicatori LED

Figura 3.5: CAN bus shield Seeed Studio

3.4.2 Sparkfun CAN-BUS Shield

Anche questo shield utilizza il controller CAN MCP2515 con il transceiver CAN MCP2551. Oltre alla comunicazione CAN, lo shield dispone di uno slot per schede uSD, un connettore seriale LCD e di un connettore per un modulo GPS EM506. Alcune caratteristiche:

- implementa CAN 2.0B con velocità fino a 1 Mb/s
- interfaccia SPI fino a 10 MHz
- Standard (11 bit) e Extended (29 bit) Data e Remote frame
- connessione CAN tramite il connettore DB-9
- Socket per un modulo GPS EM506
- porta Micro SD
- tasto di RESET
- controllo della navigazione del menu tramite joystick
- 2 indicatori LED

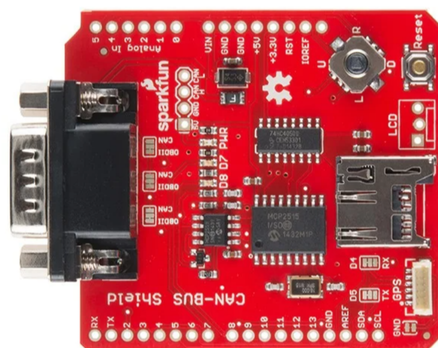

Figura 3.6: CAN bus shield SparkFun

3.5 WiFi Shield

Per abilitare la connesione ad Internet si è scelto di impiegare uno shield WiFi (Figura 3.7). Arduino WiFi Shield consente ad una scheda Arduino di connettersi a Internet utilizzando la specifica wireless 802.11 (WiFi). Si basa sul System in-Package HDG204 Wireless LAN 802.11b/g. Un AT32UC3 fornisce uno stack di rete (IP) in grado di supportare sia TCP che UDP.

Arduino comunica con il processore del Wifi Shield utilizzando il bus SPI. Questo è sui pin digitali 11, 12 e 13 dell'Arduino Uno. Sulla scheda, il pin 10 viene utilizzato per selezionare l'HDG204. Questo pin non può essere utilizzato per I/O generali. Il pin digitale 7 viene utilizzato come pin di handshake tra lo shield WiFi e Arduino e non deve essere utilizzato. Alcune caratteristiche:

- Tensione di lavoro 5 V
- compatibile con Arduino Due
- Connessione: reti 802.11 b/g
- Encryption types: WEP e WPA2 Personal
- Connessione all'Arduino via SPI
- slot micro-SD
- header ICSP
- connessione FTDI per debugging seriale
- Mini-USB per aggiornare il firmware

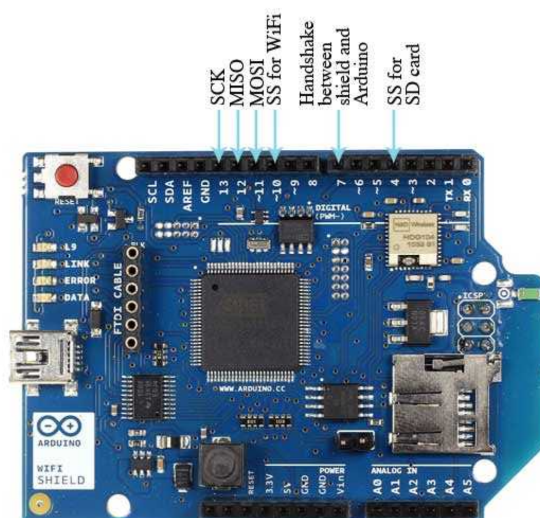

Figura 3.7: Arduino WiFi shield

3.5.1 Configurazione shield WiFi

Lo shield WiFi consente ad una scheda Arduino di connettersi a Internet utilizzando la libreria *WiFi.h*.

Esso è in grado di connettersi sia a reti aperte, che a quelle protette da crittografia WEP e WPA2 Personal. Tuttavia, lo shield non supporta la connessione a reti che utilizzando la crittografia WPA2 Enterprise.

Per una rete aperta (non crittografata), è necessario solo l'SSID.

Per le reti che utilizzano la crittografia WPA/WPA2 Personal, sono necessari sia l'SSID che la password.

La configurazione dello shield WiFi su Arduino è mostrato in Listing 3.1

```
#include <SPI.h>
#include <WiFi.h>

char ssid[] = "yourNetwork"; //  your network SSID
char pass[] = "secretPassword"; // your network password
int status = WL_IDLE_STATUS; // the WiFi radio's status

void setup() {
  // check for the presence of the shield:
  if (WiFi.status() == WL_NO_SHIELD) {
    Serial.println("WiFi shield not present");
    // don't continue:
    while (true);
  }
  // attempt to connect to WiFi network:
  while (status != WL_CONNECTED) {
    Serial.print("Attempting to connect to WPA SSID: ");
    Serial.println(ssid);
    // Connect to WPA/WPA2 network:
    status = WiFi.begin(ssid, pass);
    //wait 5 seconds for connection:
    delay(5000);
   }
  // you're connected now, so print out the data:
  Serial.print("Connected to WiFi");
}
```

Listing 3.1: Configurazione WiFi shield

Capitolo 4

Descrizione del sistema implementato

4.1 Schema generale del sistema

In Figura 4.1 è rappresentato lo schema a blocchi del sistema preso in esame. Esso può essere suddiviso in tre sezioni principali:

- *Sezione Slave*: composto da Arduino Due, dall'accelerometro ADXL345 e dal termistore.
- *Sezione CANBUS*: composto dai due shield CAN e dal bus fisico.
- *Sezione Master+MQTT*: composto da Arduino Uno, dallo shield WiFi e dal cloud IoT.

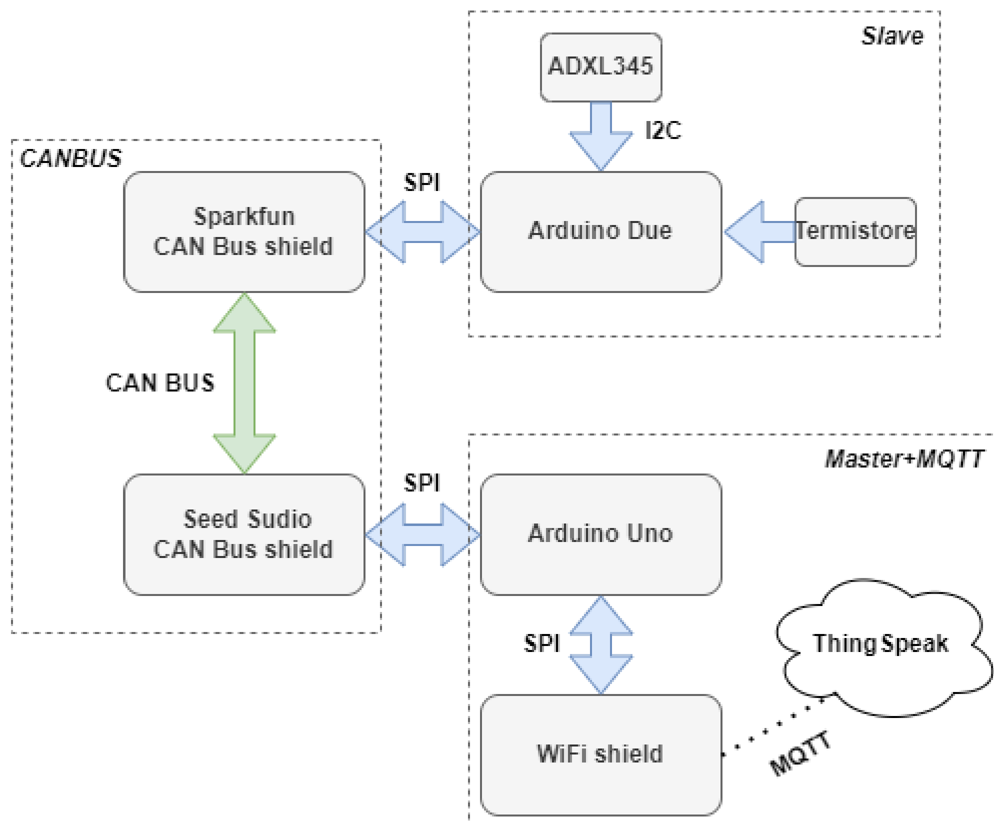

Figura 4.1: Schema a blocchi

L'accelerometro ADXL345 trasmette periodicamente i dati digigtali all'Arduino Due (Slave) attraverso l'interfaccia seriale I2C. Il termistore fornisce, invece, valori analogici di tensione, che verranno convertiti dalla scheda in valori di temperatura.

L'Arduino Due per poter inviare i dati al Master tramite il bus CAN ha bisogno dello shield CAN (in questo caso quello della SparkFun). La comunicazione tra loro avviene mediante l'interfaccia seriale SPI.

A questo punto, i dati possono essere correttamente inviati attraverso il bus CAN dove raggiungono Arduino Uno (Master). Anche quest'ultimo è connesso tramite interfaccia seriale SPI al suo shield CAN (quello della Seeed Studio).

I dati vengono quindi memorizzati, ma prima di essere inviati al broker tramite il protocollo MQTT, è necessario collegare la scheda allo shield WiFi.

Completato anche questo passo, Arduino Uno è in grado di connettersi ad Internet ed inviare i messaggi al broker utilizzando il protocollo MQTT. Successivamente il broker inoltrerà i dati alla piattaforma cloud Thingspeak, affinchè possano essere consultati dall'utente.

Lo schema definitivo dei dispositivi elettronici e delle relative connessioni è illustrato in Figura 4.2, dove lo shield WiFi è stato omesso per semplificare la visualizzazione.

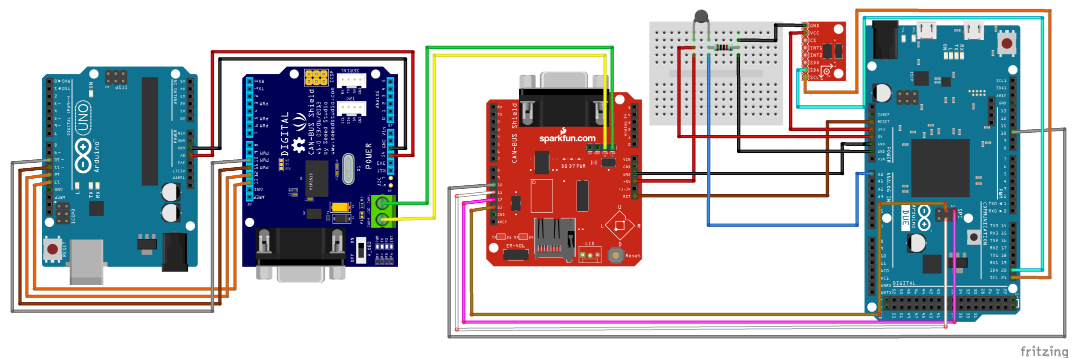

Figura 4.2: Schema Elettrico

4.2 Comunicazione tra ADXL345 e Arduino

Nel presente paragrafo si espone il procedimento mediante il quale Arduino Due acquisisce i dati provenienti dall'accelerometro ADXL345. Come accennato nella sezione 3.2, l'accelerometro può comunicare con l'ambiente esterno attraverso due interfacce seriali: SPI (a 3 o 4 fili) e I2C.

Nel nostro studio, la scelta è ricaduta su I2C a causa della sua semplicità di implementazione. A differenza di SPI, infatti, necessita di due sole linee, ovvero SDA e SCL, in confronto alle quattro utilizzate da SPI (MISO, MOSI, CS, SCLK).

Arduino Due supporta la comunicazione I2C per cui si possono collegare direttamente i pin I2C del sensore ADXL345 ai pin I2C della scheda Arduino. Il sensore

inoltre tollera una tensione massima di 3.9 V, pertanto esso sarà alimentato dal pin 3.3 V di Arduino. Il circuito in Figura 4.3 mostra come collegare ADXL345 ad Arduino Due.

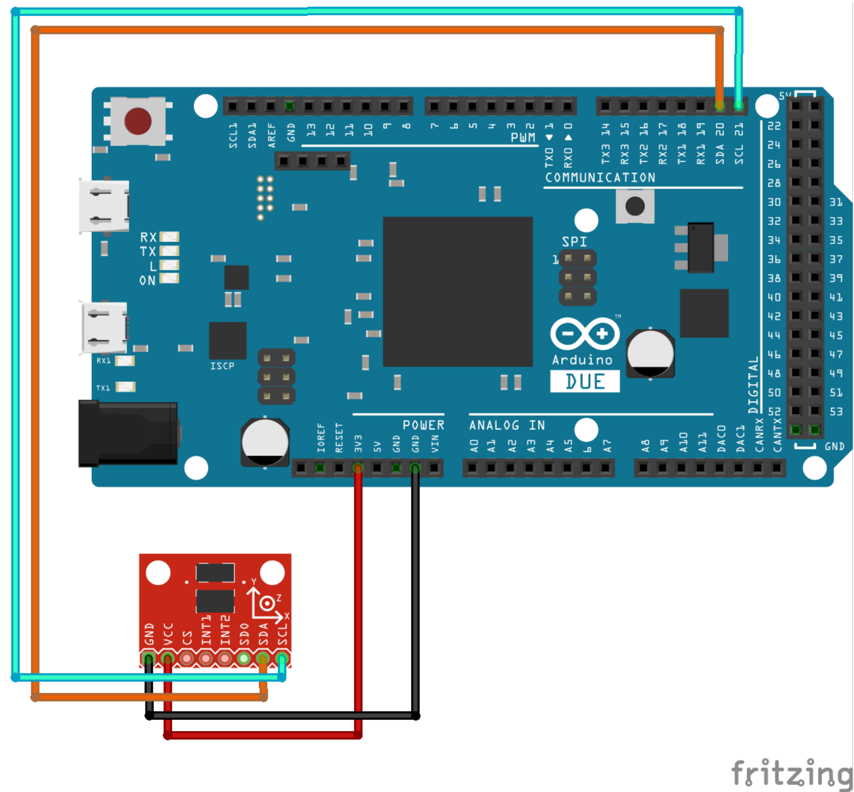

Figura 4.3: Arduino Due e ADXL345

Per leggere i dati dal sensore digitale Arduino deve essere configurato come Master I2C. Questo viene fatto attraverso l'ausilio della libreria *Wire.h* e del comando *Wire.begin()*, come mostrato in Listing 4.1.

```
#include <Wire.h>

void setup(){
    Wire.begin();
}
```

Listing 4.1: Configurazione Arduino come Master

Arduino può trasmettere dati ad uno slave tramite i comandi *Wire.write()*, *Wire.beginTransmission()* e *Wire.endTransmission()*. Può invece richiederne utilizzando il metodo *requestFrom()*, e recuperare i dati richiesti tramite il comando *Wire.read()*. Un esempio è mostrato in Listing 4.2.

```
void loop(){
  //Read from the register DATAX0
  Wire.beginTransmission(ADXL345_ADDRESS);
  Wire.write(DATAX0);
```

```
  Wire.endTransmission();
  Wire.requestFrom(ADXL345_ADDRESS, 1); //requesting 1 byte from
    ADXL345_ADDRESS
  byte x0 = Wire.read();
}
```

Listing 4.2: Esempio di comunicazione tra Arduino e ADXL345

4.2.1 Configurazione ADXL345

In questo paragrafo vengono elencati gli step per configurare l'accelerometro per la comunicazione seriale.

1. Impostare la power mode e il data transfer scrivendo sul registro 0x2C (Figura 4.4).

Register 0x2C—BW_RATE (Read/Write)

D7	D6	D5	D4	D3	D2	D1	D0
0	0	0	LOW_POWER	Rate			

Figura 4.4: Registro 0x2C

Se LOW_POWER è impostato a 0, ADXL345 lavora in normal mode. Se settato ad 1, il sensore lavora in power mode ridotta, dove il rumore è più alto.

I bit D3,D2,D1 e D0 selezionano il data rate.

2. Selezionare il formato dei dati scrivendo sul registro 0x31 (Figura 4.5).

Register 0x31—DATA_FORMAT (Read/Write)

D7	D6	D5	D4	D3	D2	D1	D0
SELF_TEST	SPI	INT_INVERT	0	FULL_RES	Justify	Range	

Figura 4.5: Registro 0x31

Se il bit SELF-TEST è impostato su 1, al sensore viene applicata una forza di self-test, provocando uno shift nei dati di uscita. Se è impostato su 0, la forza di self-test è disabilitata.

Se il bit SPI è impostato su 1, ADXL345 utilizza la modalità SPI a tre fili. Se è impostato su 0, utilizza la modalità SPI a quattro fili.

Se il bit INT_INVERT è impostato su 0, imposta gli interrupt su attivo ALTO e se è impostato su 1, imposta gli interrupt su attivo BASSO.

Se il bit FULL_RES è impostato su 1, il sensore fornisce un valore alla massima risoluzione. Altrimenti, se è impostato su 0, viene utilizzato il valore predefinito a 10 bit.

Se il bit di giustificazione è impostato su 1, i valori di accelerazione nei registri da 0x32 a 0x37 vengono giustificati a sinistra. Se è impostato su 0, questi valori sono giustificati a destra.

La Figura 4.6 mostra il formato dei registri dei valori dei dati con le giustificazioni sinistra e destra e le posizioni di LSB o MSB per diversi intervalli di misurazione.

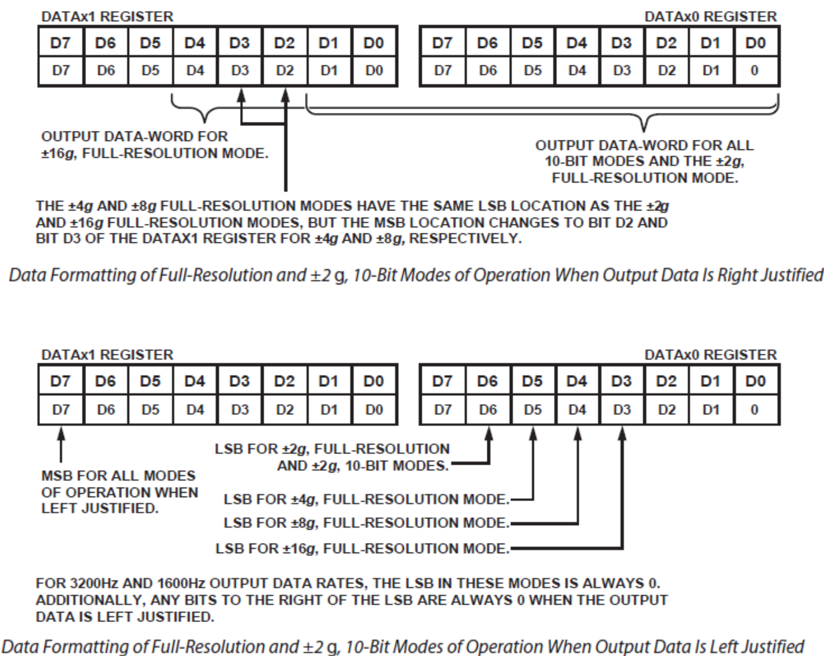

Figura 4.6: Formattazione dati e giustificazione

3. Impostare le opzioni di power-saving nel registro 0x2D (Figura 4.7).

Register 0x2D—POWER_CTL (Read/Write)

D7	D6	D5	D4	D3	D2	D1	D0
0	0	Link	AUTO_SLEEP	Measure	Sleep	Wakeup	

Figura 4.7: Registro 0x2D

Se il bit di Link è impostato su 1, le funzioni di attività e inattività sono collegate in serie (il che significa che la funzione di attività viene ritardata finché non viene rilevata la funzione di inattività). Se è impostato su 0, entrambe le funzioni sono simultanee. La funzione di inattività si riferisce ad una situazione

in cui l'accelerazione è inferiore al valore THRESH_INACT (registro 0x25) per almeno il tempo indicato dal TIME_INACT (registro 0x26).

Se il bit di Link è impostato a 1 e il bit AUTO_SLEEP è impostato su 1, abilita la funzionalità di auto sleep. In questa modalità, l'ADXL345 passa automaticamente alla modalità di sleep se la funzione di inattività è abilitata e viene rilevata inattività. Se anche l'attività è abilitata, l'ADXL345 si risveglia automaticamente dalla modalità di sleep dopo averla rilevata e torna a funzionare alla velocità impostata nel registro BW_RATE.

Se il bit AUTO_SLEEP è impostato a 0, disabilita il passaggio automatico alla modalità sleep. Se il bit di Link non è impostato, la funzione AUTO_SLEEP è disabilitata e l'impostazione del bit AUTO_SLEEP non ha alcun impatto sul funzionamento del dispositivo.

Se il bit di Measure è impostato su 1, l'ADXL345 funziona in modalità di misura. Se è impostato su 0, l'ADXL345 funziona in modalità standby.

Se il bit Sleep è impostato su 1, l'ADXL345 funziona in modalità sleep. Se è impostato su 0, l'ADXL345 funziona in modalità normale. La modalità di sleep sopprime DATA_READY, cioè interrompe la trasmissione dei dati al registro FIFO e cambia la frequenza di campionamento su quella specificata dai bit di Wakeup.

La configurazione dell'ADXL345 utilizzata nel nostro progetto è presentata in Listing 4.3.

```
void setup(){
  //Set the power mode and data transfer
  Wire.beginTransmission(ADXL345_ADDRESS);
  Wire.write(0x2C); //Selecting the register
  Wire.write(0x08); //Setting the register
  Wire.endTransmission();

  //Set the data format---Full res. 10 bit, values right-justified,
    range +/- 2g
  Wire.beginTransmission(ADXL345_ADDRESS);
  Wire.write(0x31); //Selecting the register
  Wire.write(0x08); //Setting the register
  Wire.endTransmission();

  //Set the power saving features: Measure mode
  Wire.beginTransmission(ADXL345_ADDRESS);
  Wire.write(0x2D); //Selecting the register
  Wire.write(0x08); //Setting the register
  Wire.endTransmission();
  delay(500);
}
```

Listing 4.3: Configurazione ADXL345

4.2.2 Lettura dei dati dall'ADXL345

I valori di accelerazione forniti dal sensore sono letti:

- asse X, nei registri 0x32 e 0x33.
- asse Y, nei registri 0x34 e 0x35.
- asse Z, nei registri 0x36 e 0x37.

I valori vengono forniti a 16 bit in complemento a due. La risoluzione, il tipo di giustificazione e il range di misurazione vengono impostati a seconda del valore del registro 0x31.

In qualunque range di misurazione, la risoluzione a 10 bit può essere selezionata. Il valore a 10 bit di accelerazione può poi essere giustificato a destra o a sinistra. Se il valore è giustificato a destra, i valori di accelerazione a 10 bit possono essere ottenuti mascherando il secondo byte (che viene letto in 0x33 per l'asse x, 0x35 per l'asse y e 0x37 per l'asse z) con il valore 0x03 e shiftandolo a destra otto volte. Quindi, i due byte vengono combinati (0x32 e 0x33; 0x34 e 0x35; 0x36 e 0x37) in un numero intero a 16 bit.

Se il valore è giustificato a sinistra, i valori di accelerazione a 10 bit possono essere ottenuti shiftando a destra il primo byte (che viene letto in 0x32 per l'asse x, 0x34 per l'asse y e 0x36 per l'asse z) sei volte, mascherando il secondo byte (che viene letto in 0x33 per l'asse x, 0x35 per l'asse y e 0x37 per l'asse z) con il valore 0x3F in un byte temporaneo. Il byte temporaneo viene shiftato a sinistra due volte, per poi essere combinato al primo byte. Quindi, il primo byte viene shiftato a sinistra sei volte e combinato con il secondo byte (0x32 e 0x33; 0x34 e 0x35; 0x36; e 0x37) per un numero intero a 16 bit.

Il valore a 10 bit varierà da 0 a 1024. L'accelerazione viene misurata in entrambe le direzioni lungo l'asse. Quindi, se il valore è maggiore di 511, bisogna sottrare 1024 per ottenere un valore negativo che indica l'altra direzione dell'asse.

Infine per una risoluzione a 10 bit, il valore dell'accelerazione nell'unità di misura della gravità può essere derivato moltiplicando questo valore per 4 mg (0.004) per l' intervallo di $\pm 2g$, 7.8 mg (0.0078) per l'intervallo di $\pm 4g$, 15.6 mg (0.0156) per l'intervallo di $\pm 8g$ o 31.25 mg (0.03125) per l'intervallo di $\pm 16g$.

Il seguente codice Arduino (Listing 4.4) illustra i passi appena descritti per acquisire i dati forniti dall'ADXL345. Per motivi di chiarezza visiva, viene mostrato il codice relativo all'acquisizione solamente del valore di accelerazione lungo l'asse x; gli stessi passaggi devono, ovviamente, essere replicati per ottenere i valori lungo gli assi y,z.

```
//Defining ADXL345 address
#define ADXL345_ADDRESS 0x53
//Defining ADXL345 data register
#define DATAX0 0x32
#define DATAX1 0x33
```

```
#define DATAY0 0x34
#define DATAY1 0x35
#define DATAZ0 0x36
#define DATAZ1 0x37

void loop(){
  //Read from DATAX0
  Wire.beginTransmission(ADXL345_ADDRESS);
  Wire.write(DATAX0); //Selecting the register
  Wire.endTransmission();
  Wire.requestFrom(ADXL345_ADDRESS, 1); //requesting 1 byte from
    ADXL345_ADDRESS
  byte x0 = Wire.read();
  //Read from DATAX1
  Wire.beginTransmission(ADXL345_ADDRESS);
  Wire.write(DATAX1); //Selecting the register
  Wire.endTransmission();
  Wire.requestFrom(ADXL345_ADDRESS, 1); //requesting 1 byte from
    ADXL345_ADDRESS
  byte x1 = Wire.read();

  // Since the value is right-adjusted, the 10-bit acceleration is
    obtained by masking the second byte (which is read from 0x33 for
    the x-axis) with 0x03, right-shifting it eight times.
  //Then, the two bytes are combined (0x32 and 0x33) to a 16-bit
    integer.
  x1 = x1 & 0x03;

  // Shifting and combining the two bytes
  uint16_t x = (x1 << 8) | x0;
  int16_t xf = x;

  //The 10-bit value will range from 0 to 1024.
  //The acceleration is measured in both directions along the axis.
  //So, if the value is greater than 511, subtract 1024 from it to get
     a negative value which indicates the other direction of the axis.
  if(xf > 511)
  {
    xf = xf - 1024;
  }
  //For a 10-bit resolution, the value of the acceleration in a
    gravity unit can be derived by multiplying this value with 4 mg
    (0.004) for +/- 2g range.
  float xa = xf * 0.004;
  Serial.print("X = ");
  Serial.print(xa);
  Serial.print(" g");
  Serial.println();
  delay(500);
}
```

Listing 4.4: Acquisizione dati da ADXL345

4.3 Comunicazione tra Arduino e il termistore

Come accennato nella sezione 3.3, il sensore fornisce dei valori analogici di tensione, che verranno convertiti in temperatura.

Siccome il termistore non è altro che un resistore variabile, necessitiamo di conoscere il suo valore prima di calcolare la temperatura. Arduino non può calcolare il valore di resistenza direttamente, ma solo la tensione ai suoi capi.

Per tale ragione si utilizza un partitore di tensione con una resistenza di valore noto, ad esempio 10 kΩ, per conoscere il valore di resistenza del termistore, come mostrato in Figura 4.8.

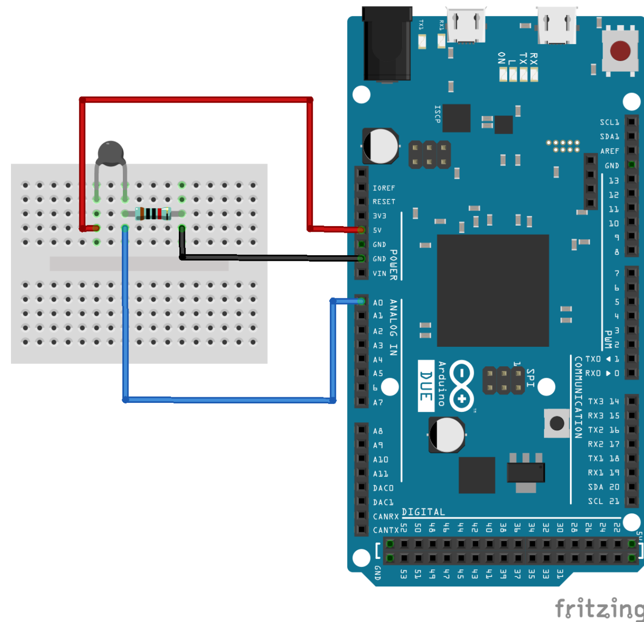

Figura 4.8: Schema di collegamento tra Arduino Due e il termistore

A questo punto per calcolare la resistenza del termistore viene ricavata la seguente equazione dalla formula generale del partitore di tensione:

$$R = R_1 \cdot (\frac{V_{\text{CC}}}{V_{\text{out}}} - 1)$$

dove:

- V_{CC}: tensione di alimentazione (5 V)
- V_{out}: tensione tra il termistore e la resistenza nota
- R_1: valore della resistenza nota
- R: valore di resistenza del termistore

Infine attraverso il valore di resistenza del termistore è possibile calcolare la temperatura tramite la formula di Steinhart-Hart:

$$\frac{1}{T} = a + b \cdot \ln(R) + c \cdot \ln^3(R)$$

dove a,b e c sono i coefficienti di Steinhart-Hart e vanno specificati per ciascun dispositivo, T è la temperatura in kelvin e R è la resistenza del termistore in ohm.

I termistori NTC possono essere anche caratterizzati con un'equazione più semplice detta equazione con parametro B o beta value (che in essenza è l'equazione di Steinhart-Hart con c=0):

$$\frac{1}{T} = \frac{1}{T_0} + \frac{1}{B} \cdot \ln\left(\frac{R}{R_0}\right)$$

dove le temperature sono in kelvin (K) e R_0 è la resistenza alla temperatura T_0 (di solito 25°C=298,15 K).

L'acquisizione dei valori di temperatura in Arduino del sistema sopra menzionato viene mostrato in Listing 4.5.

```
#include <math.h>
int ThermistorPin = 0;
int Vo;
float R1 = 10000; //known resistor
float logR2, R2, T;
float c1 = 1.009249522e-03, c2 = 2.378405444e-04, c3 = 2.019202697e
    -07; //defining coefficient

void setup() {
Serial.begin(9600);
}

void loop() {
  Vo = analogRead(ThermistorPin);
  R2 = R1 * (1023.0 / (float)Vo - 1.0);
  logR2 = log(R2);
  T = (1.0 / (c1 + c2*logR2 + c3*logR2*logR2*logR2));
  T = T - 273.15;
  Serial.print("Temperature: ");
  Serial.print(T);
  Serial.println(" C");
  delay(500);
}
```

Listing 4.5: Acquisizione temperatura dal termistore

Capitolo 5

Integrazione con l'interfaccia CANBUS

Questo capitolo si concentra sulla sezione CANBUS, affrontando tematiche come la configurazione dei dispositivi e l'implementazione della trasmissione.

5.1 Configurazione dei nodi CANBUS

Il primo nodo da configurare è l'Arduino Due che lavora come Slave nel nostro sistema.

I dati acquisiti dai sensori devono essere, infatti, formattati in pacchetti adatti ad essere trasmessi tramite lo shield sul bus CAN.

Lo shield CAN di SparkFun presenta 4 linee dedicate alla trasmissione CAN che sono:

- 5 V
- GND
- CANH
- CANL

Lo shield comunica con Arduino attraverso l'interfaccia seriale SPI, necessitando pertanto di 4 linee (MISO, MOSI, SCLK, CS) per lo scambio di dati tra i due dispositivi.

In Figura 5.1 viene mostrata la connessione tra lo shield e Arduino Due. Il pin di CS (Chip Select) dello shield è posizionato sul pin 10, che verrà collegato al medesimo pin di Arduino. Le linee di MOSI, MISO e SCLK sono posizionate rispettivamente sui pin 11, 12 e 13 dello shield e devono essere connesse all'header SPI di Arduino Due come mostrato in Figura 5.1.

Nella figura, oltre alle linee di alimentazione, GND e RESET, si possono osservare le due linee circuitali fondamentali (gialla e verde) che costituiscono il bus CAN. Naturalmente, le due linee CANH e CANL dovranno essere collegate in ricezione al secondo shield, nelle rispettive linee CAN di quest'ultimo.

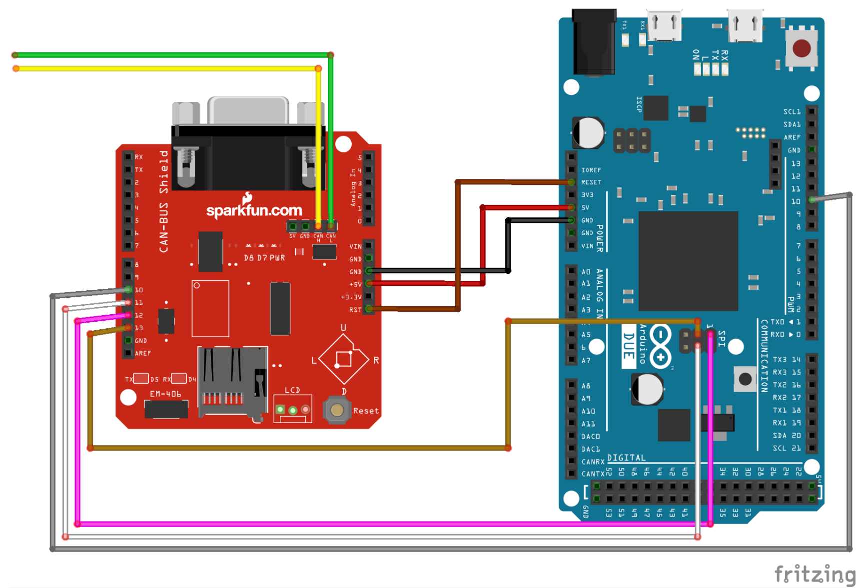

Figura 5.1: Connessione SparkFun CAN bus shield ad Arduino Due

Il secondo nodo da configurare è ovviamente Arduino Uno, il Master del nostro sistema, equipaggiato con lo shield CAN di Seeed Studio. La sua configurazione è molto simile alla precedente; infatti anche questo shield utilizza l'interfaccia seriale SPI per comunicare con la scheda. L'unica differenza sta nel fatto che Arduino Uno presenta i pin designati per la comunicazione SPI sui pin digitali 10, 11, 12 e 13, rispettivamente per CS, MOSI, MISO e SCLK. La Figura 5.2 mostra il collegamento tra lo shield CAN di Seeed Studio e Arduino Uno.

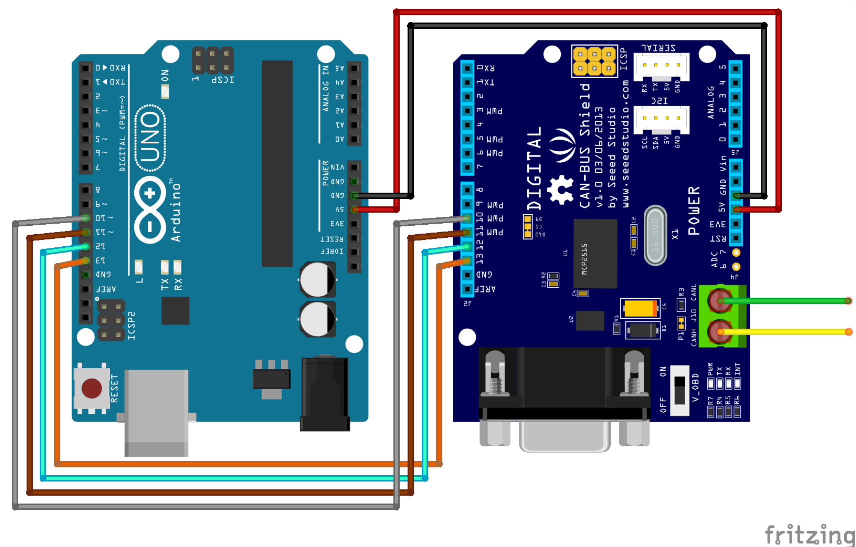

Figura 5.2: Connessione Seeed Studio CAN bus shield ad Arduino Uno

5.2 Implementazione della trasmissione dati tramite CANBUS

Come accennato nel Capitolo 3 entrambi gli shield adottano il controller CAN MCP2515. Pertanto, l'effettiva implementazione della comunicazione CAN tra i due dispositivi è resa possibile grazie all'utilizzo della libreria *mcp2515_can.h*.

Attraverso questa libreria è infatti possibile definire i parametri chiave della comunicazione, tra cui la scelta del pin designato al CS (Chip Select), la scelta del baudrate, il formato dei messaggi e la configurazione dell'identificatore del pacchetto.

Per selezionare la velocità di trasmissione attraverso il bus CAN, la libreria mette a disposizione il comando *CAN.begin()*, che consente di selezionare un baudrate compreso tra 5 kbps e 1000 kbps, in base alle specifiche esigenze. Un esempio di applicazione di questa impostazione è illustrato in Listing 5.1.

```
#include <SPI.h>

#include "mcp2515_can.h"

const int SPI_CS_PIN = 10; //Define CS pin

mcp2515_can CAN(SPI_CS_PIN); // Set CS pin

void setup() {
  while(CAN_OK != CAN.begin(CAN_25KBPS)) {// init can bus : baudrate =
     25kbs
        Serial.println("CAN init fail, retry...");
        delay(100);
    }
    Serial.println("CAN init ok!");
    delay(100);
}
```

Listing 5.1: Inizializzazione nodo CAN

Naturalmente questa procedura deve essere eseguita su entrambi i nodi, sia sul trasmettitore che sul ricevitore.

Focalizzandosi per il momento sul lato trasmittente, è importante notare che l'accelerometro ADXL345 fornisce tre valori di accelerazione, uno per ogni asse x,y e z, mentre il termistore un singolo valore di temperatura. Per tale ragione si è deciso di implementare uno script che generi quattro pacchetti distinti, ciascuno con identificatore univoco, contenente i dati acquisiti dai sensori.

Il valore di accelerazione e temperatura è un valore di tipo *float*, tuttavia, la versione standard del protocollo CAN consente la trasmissione seriale di 8 byte. Pertanto, prima della trasmissione, i valori acquisiti devono essere convertiti in byte e inseriti in un array lungo 8 byte. Questo può essere fatto tramite il comando *memcpy()* come mostrato in Listing 5.2.

```
//Read 3-axis values
float X = readX();
float Y = readY();
float Z = readZ();
//Read Temperature
float T = readT();
//create arrays
byte XArray[8];
byte YArray[8];
byte ZArray[8];
byte Temp[8];
//convert float into byte
memcpy(XArray, &X, sizeof(float));
memcpy(YArray, &Y, sizeof(float));
memcpy(ZArray, &Z, sizeof(float));
memcpy(Temp, &T, sizeof(float));
```

Listing 5.2: Conversione da float a byte

A questo punto i valori sono pronti per essere trasmessi tramite il comando *CAN.sendMsgBuf(id, ext, len, buf)*, il quale consente di configurare l'identificatore del pacchetto, il tipo di pacchetto, la lunghezza del pacchetto e il vettore che contiene il dato da trasmettere. Un esempio di trasmissione dei valori di accelerazione è mostrato in Listing 5.3.

```
void loop(){
  //Send data
  CAN.sendMsgBuf(0x100, 0, 8, XArray); //x value: ID 0x100, standard
    frame, length 8 byte
  delay(100);
  CAN.sendMsgBuf(0x101, 0, 8, YArray); //y value: ID 0x101, standard
    frame, length 8 byte
  delay(100);
  CAN.sendMsgBuf(0x102, 0, 8, ZArray); //z value: ID 0x102, standard
    frame, length 8 byte
  delay(100);
  CAN.sendMsgBuf(0x103, 0, 8, Temp); //Temperature value: ID 0x103,
    standard frame, length 8 byte
  delay(100);
}
```

Listing 5.3: Trasmissione dei tre valori di accelerazione

Lato ricezione è possibile rilevare la presenza di messaggi sul bus mediante l'utilizzo del comando *CAN.checkReceive()*. La lettura di quest'ultimi è, poi, possibile tramite l'operazione *CAN.readMsgBuf()*.

È possibile, inoltre, filtrare specifici pacchetti in base al loro identificatore utilizzando *CAN.getCanId()*. Il codice di seguito (Listing 5.4) mostra la lettura dei soli pacchetti contenenti i valori di accelerazione lungo l'asse x.

```
void loop(){
  unsigned char len = 0;
  unsigned char buffer[8];

  unsigned char xbuf[8];

  if (CAN_MSGAVAIL == CAN.checkReceive()) {          // check if data
    coming

     CAN.readMsgBuf(&len, buffer);    // read data,  len: data length,
    buffer: data buffer
     unsigned long canId = CAN.getCanId();
     if (canId == X_ID)
         {
           Serial.println("-----------------------------");
           Serial.print("Get data from ID: 0x");
           Serial.println(canId, HEX);
           for (int i = 0; i < len; i++) {
             xbuf[i] = buffer[i];
             Serial.print(xbuf[i], HEX);
             }
         Serial.println();
         /********* Form bytes to float *********/
         memcpy(&X,xbuf,sizeof(float));
         Serial.print("X = ");
         Serial.print(X);
         Serial.print(" g");
         Serial.println();
         }
    }
}
```

Listing 5.4: Lettura dei valori di accelerazione lungo x

Capitolo 6

Implementazione trasmissione MQTT su Cloud IoT

Dopo la presentazione teorica del protocollo MQTT nel Capitolo 2, verranno ora esaminate concretamente le fasi coinvolte nell'implementazione della trasmissione MQTT su di un Cloud IoT.

Questo capitolo, si concentrerà sull'analisi pratica degli elementi chiave della trasmissione, tra cui la scelta del broker MQTT e la sua configurazione, la gestione dei topic e la connessione dei client MQTT.

6.1 Introduzione Cloud IoT

IoT (*Internet of Things*) rappresenta il concetto di una rete aperta di dispositivi smart (*Things*) che hanno la capacità di condividere informazioni, dati e risorse nell'intera infrastruttura.

L'architettura generale dell'IoT (Figura 6.1) si basa su tre concetti:

- *Things*: i dispositivi (sistemi embedded, sensori, attuatori, dispositivi smart, ecc...) che si connettono via cavo o wireless alla rete Internet.
- *Rete*: l'infrastruttura che connette i dispositivi al cloud; essa permette lo scambio di informazioni, dati e risorse tra quest'ultimi.
- *Cloud*: il server remoto in grado di immagazzinare i dati forniti dai dispositivi. È qui che, solitamente, l'utente può monitorare i dispositivi e analizzare le informazioni ricevute.

Figura 6.1: Esempio di architettura IoT

Il termine "Cloud IoT" si riferisce all'utilizzo del cloud computing nel contesto dell'Internet of Things (IoT). L'integrazione del cloud computing nell'IoT offre diversi vantaggi:

- Archiviazione e Elaborazione dei dati: i dispositivi IoT generano enormi quantità di dati, spesso in tempo reale. Utilizzando servizi cloud, è possibile archiviare e analizzare questi dati su una piattaforma centralizzata, consentendo una gestione più efficiente.
- Scalabilità: il cloud consente la scalabilità delle risorse in base alle esigenze dell'IoT. Quando il numero di dispositivi cresce o diminuisce, è possibile adattare le risorse di elaborazione e archiviazione di conseguenza.
- Accessibilità Remota: i dati generati dai dispositivi IoT possono essere accessibili da qualsiasi luogo attraverso Internet, facilitando il monitoraggio e la gestione remota.

Di solito, l'accesso ad un cloud per le applicazioni IoT è reso possibile grazie all'utilizzo di piattaforme IoT dedicate, che offrono il proprio servizio di cloud.

Naturalmente esistono numerosissime piattaforme IoT, come ad esmepio Google Cloud, AWS IoT (Amazon Web Service IoT), HiveMQ, ThingSpeak, ecc..., a seconda delle specifiche necessità.

6.2 Scelta del broker MQTT

Il nostro sistema prevede che i dati acquisiti dai sensori vengano inviati ad un cloud IoT tramite il protocollo MQTT; per questa ragione è essenziale utilizzare una piattaforma che supporti tale comunicazione.

Due piattaforme adatte per la nostra applicazione sono ThingSpeak e HiveMQ, poiché entrambe mettono a disposizione un broker MQTT pubblico. La ragione per il quale è stato scelto ThingSpeak, a discapito di HiveMQ, sta nel fatto che esso rende possibile monitorare e visualizzare i dati ricevuti dal sensore attraverso grafici configurabili.

ThingSpeak è, infatti, una piattaforma di analisi IoT fornita da MathWorks, che offre la possibilità di archiviare, visualizzare e analizzare flussi di dati in tempo reale nel cloud. ThingSpeak fornisce immediate visualizzazioni dei dati pubblicati dai dispositivi o apparecchiature, oltre a consentire l'esecuzione di codice MATLAB direttamente sulla piattaforma per effettuare analisi ed elaborazioni online dei dati appena ricevuti.

I dati vengono archiviati in canali e ciascun canale può memorizzare fino a otto campi dati. La visualizzazione dei canali può essere resa pubblica o privata. La comunicazione tra la piattaforma e i dispositivi è possibile mediante l'utilizzo di chiavi API (*Application Programming Interface*); esse infatti consentono di scrivere

(*writeAPIkey*) dati su un canale privato, o di leggere dati (*readAPIkey*) da un canale privato. Queste chiavi vengono generate automaticamente alla creazione di un canale.

Come accennato precedentemente ThingSpeak è la piattaforma adatta per l'implementazione di una trasmissione MQTT; essa mette infatti a disposizione un broker MQTT gratuito all'indirizzo: *mqtt3.thingspeak.com* e alla porta *1883*, garantendo un livello di QoS di tipo 0 (At most once).

Il broker ThingSpeak supporta sia la pubblicazione MQTT che la sottoscrizione MQTT, come mostrato in Figura 6.2 e in Figura 6.3.

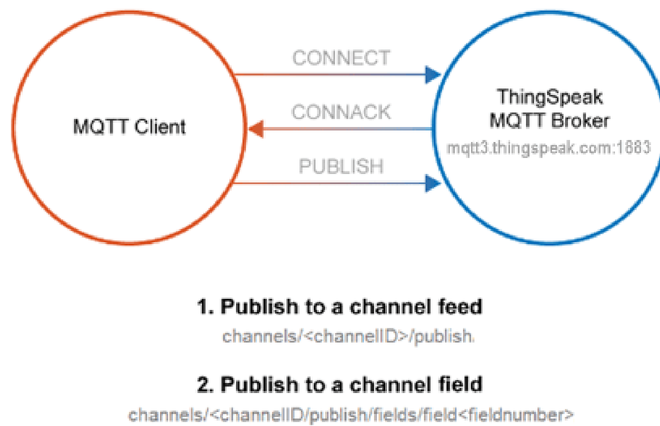

Figura 6.2: Pubblicazione MQTT con ThingSpeak

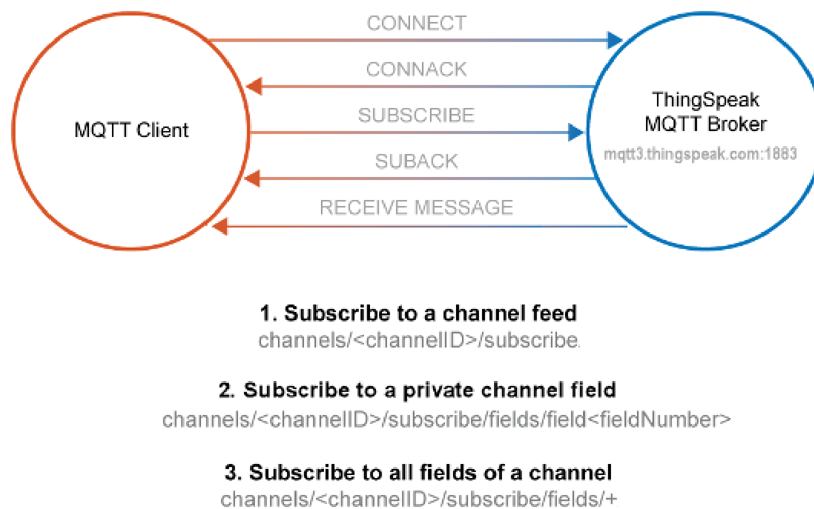

Figura 6.3: Sottoscrizione MQTT con ThingSpeak

6.3 Implementazione MQTT su ThingSpeak

Il primo passo consiste nell'abilitare la comunicazione MQTT su ThingSpeak, creando dunque il dispositivo adibito alla comunicazione e il canale desiderato per la visualizzazione dei dati. In questa fase è possibile configurare il dispositivo specificando le autorizzazioni e assegnando le relative credenziali.

Per pubblicare un messaggio MQTT si deve specificare un topic; su ThingSpeak questo viene fatto attraverso la stringa

"*channels/<channelID>/publish*"

se si vuole pubblicare su tutti i campi, o con

"channels/<channelID>/publish/fields/field<fieldnumber>"

per aggiornare un solo campo.

Il messaggio pubblicato sarà, invece, una stringa del formato:

"field1=xvalue&field2=yvalue&field3=zvalue&field4=temp"

composto da quattro campi perché si stanno pubblicando i tre valori di accelerazione e il valore di temperatura.

Passando ad Arduino, la comunicazione MQTT viene interamente gestita dalla libreria *PubSubClient.h*, che mette a disposizione una vasta gamma di comandi. Tra questi, troviamo metodi per selezionare server e porta, altri per verificare lo stato della connessione di un client, ma soprattuto istruzioni per la pubblicazione di messaggi e sottoscizione a argomenti specifici.

Un esempio della configurazione di un client MQTT è illustrato in Listing 6.1

```
#include <PubSubClient.h>

const char* mqtt_server = "mqtt3.thingspeak.com";
char mqtt_username[] = "your_mqtt_username";
char mqtt_password[] = "your_mqtt_password";
char clientID[] = "your_clientID";
int mqtt_port = 1883;
long channelID = your_channelID;
char* Topic = "channels/channelID/publish";

// Initialize the WiFi client library
WiFiClient wificlient;
PubSubClient client(wificlient);

void setup() {
    client.setServer(mqtt_server, mqtt_port);
    delay(500);
}
```

Listing 6.1: Esempio di configurazione di un client MQTT

Dopo aver configurato il client, bisogna stabilire la sua connessione al broker MQTT mediante l'uso del comando *client.connect()*; in caso di fallimento, è possibile verificare lo stato del client e il relativo codice di ritorno (rc) tramite il comando *client.state()*. Il codice di ritorno specifica il motivo della mancata connessione del client e può assumere diversi valori:

- -4 : MQTT_CONNECTION_TIMEOUT - il server non ha risposto nel tempo prestabilito
- -3 : MQTT_CONNECTION_LOST - la connessione è stata interrotta
- -2 : MQTT_CONNECT_FAILED - la connessione non è riuscita
- -1 : MQTT_DISCONNECTED - il client si è disconnesso
- 0 : MQTT_CONNECTED - il client è connesso
- 1 : MQTT_CONNECT_BAD_PROTOCOL - il server non supporta la versione richiesta di MQTT
- 2 : MQTT_CONNECT_BAD_CLIENT_ID - il server ha rifiutato l'identificatore del client
- 3 : MQTT_CONNECT_UNAVAILABLE - il server non è in grado di accettare la connessione
- 4 : MQTT_CONNECT_BAD_CREDENTIALS - username/password incorretti
- 5 : MQTT_CONNECT_UNAUTHORIZED - il client non è autorizzato alla connessione

In Listing 6.2 è mostrato un esempio della connessione di un client al broker MQTT.

```
void loop()
{
  //Check if client connected
  if (!client.connected()) {
    // Loop until we're connected
    while (!client.connected()) {
        // Attempt to connect
        Serial.print("Attempting MQTT connection...");
        //check client credentials
        if (client.connect(clientID, mqtt_username, mqtt_password)) {
        Serial.println("connected");
        }
        else {
        Serial.print("failed, rc=");
```

```
        Serial.print(client.state());
        Serial.println(" try again in 5 seconds");
        delay(5000);  // Wait 5 seconds before retrying
        }
    }
  }
  client.loop();
}
```

Listing 6.2: Connessione Client MQTT

L'ultimo passaggio è la pubblicazione dei dati acquisiti dal sensore. Dopo aver dunque creato la stringa da trasmettere, il messaggio viene pubblicato con il comando *client.publish()*, come mostrato in Listing 6.3

```
void loop()
{
    publishMessage(X,Y,Z,T);
}

/**** Method for Publishing MQTT Messages **********/
void publishMessage(float x, float y, float z, float t)
{
  //Building the message
  String data = "field1=";
  data += String(x);
  data += "&field2=";
  data += String(y);
  data += "&field3=";
  data += String(z);
  data += "&field4=";
  data += String(t);
  int lunghezza_dato = data.length();
  char messaggio[lunghezza_dato];
  data.toCharArray(messaggio, lunghezza_dato + 1);
  Serial.println(messaggio);
  //publish message
  client.publish(Topic,messaggio);
  Serial.println("Message published ["+String(Topic)+"]: "+messaggio);
}
```

Listing 6.3: Pubblicazione messaggi MQTT

Capitolo 7

Problematiche riscontrate e soluzioni implementate

In questo capitolo verranno esaminate le criticità emerse in fase di sviluppo del progetto e le soluzione adottate. Saranno poi forniti dei grafici raffiguranti l'acquisizione dei valori di accelerazione e temperatura.

7.1 Incompatibilità tra Arduino Due e lo shield CAN di Seed Studio

Nella fase iniziale del progetto, si era considerato l'utilizzo di due schede Arduino Due come nodi CAN. Navigando in rete, però, è emerso che la versione 1.0 dello shield CAN di Seeed Studio, impiegata nel nostro lavoro, presenta problemi di incompatibilità con la scheda Arduino Due, impedendo l'utilizzo della libreria *mcp2515_can.h* indispensabile nel progetto. Questo è uno dei motivi per il quale la versione 1.0 è stata rapidamente sostituita dalla versione 1.2.

Disponendo però solo della versione 1.0 dello shield CAN, la soluzione è stata modificare un nodo, sostituendo Arduino Due con un Arduino Uno. Quest'ultimo, infatti, non presenta alcun problema di incompatibilità, consentendo l'utilizzo della libreria necessaria all'implementazione della trasmissione CAN.

7.2 Problemi con l'architettura di Arduino Due e la libreria dello shield CAN di SparkFun

La problematica precedente impone l'uso dello shield CAN di SparkFun in combinazione con Arduino Due. Lo shield, dispone di una libreria dedicata che supporta esclusivamente architetture AVR, presenti, ad esempio, nelle schede Arduino Uno, Arduino Nano, Arduino Mega, ecc..., ma non in Arduino Due. Quest'ultimo utilizza, infatti, un microcontrollore SAMx8E basato su un'architettura ARM, risultando, perciò, incompatibile con libreria specifica dello shield.

La soluzione a questo problema, è stata quella di non montare lo shield direttamente su Arduino Due, bensì di collegrali separatamente, come descritto nel Capitolo 5, e di utilizzare la libreria fornita da Seeed Studio. Tutti e due gli shield montano,

infatti, lo stesso CAN controller (MCP2515), il che consente l'utilizzo della libreria di Seeed Studio per entrambi gli shield.

7.3 Disponibilità dei Socket sullo shield WiFi

Il termine "socket" nel contesto del WiFi Shield si riferisce ai canali di comunicazione che consentono al dispositivo di stabilire connessioni di rete con altri dispositivi o server tramite il protocollo TCP/IP. I socket vengono identificati da numeri di porta e sono utilizzati per instradare i dati tra i dispositivi connessi.

Lo shield WiFi utilizzato è dotato di quattro socket, il che significa che supporta al massimo quattro connessioni simultanee.

Una volta che i socket sono esauriti, è necessario liberarli, al fine di rendere disponibili nuove connessioni. Tipicamente questo si realizza tramite i comandi *client.stop()* e *client.flush()*. Tuttavia la libreria PubSubClient gestisce automaticamente la connessione e la disconnessione del client, rendendo inefficaci tali comandi.

All'esecuzione del codice, infatti, si verificava che dopo quattro connessioni MQTT consecutive, si manifestava l'errore:

No Socket Available Connection Failed

Per risolvere questa problematica, è stato necessario intervenire direttamente sulla libreria *WiFi.h*, implementando due funzioni aggiuntive: una per monitorare la disponibilità dei socket e l'altra per liberare l'eventuale socket.

Grazie a questa aggiunta, dopo ogni connessione MQTT, il socket viene liberato, eliminando il rischio di esaurire i socket disponibili.

Le due funzioni utilizzate sono mostrate in Listing 7.1

```
/*********** Show Socket Status ******/
void ShowSockStatus() {
  for(int x = 0; x < MAX_SOCK_NUM; x++) {
    Serial.print(F("Socket #"));
    Serial.print(x,DEC);
    if(WiFi._state[x] == -1) Serial.print(F(": available  port "));
    else Serial.print(F(": used  port "));

    Serial.println(WiFi._server_port[x]);
  }
}
/************ Make Socket Available *************/
void SetSockStatus() {
  for(int x = 0; x < MAX_SOCK_NUM; x++) {
    WiFi._state[x] = -1;
  }
}
```

Listing 7.1: Funzioni per gestire i socket

7.4 Intervallo di pubblicazione sul cloud IoT

L'ultima problematica è legata ad una limitazione tecnologica del cloud. La versione gratuita di ThingSpeak consente, infatti, la pubblicazione dei dati sui canali ogni 15 secondi. Sebbene concettualmente non rappresenti un grande vincolo, perché permette comunque di implementare la comunicazione MQTT e di visualizzare i dati in appositi grafici, diventa una limitazione in applicazioni real-time. Ad esempio in scenari in cui dei sensori acquisiscono dati in intervalli estremamente rapidi e quest'ultimi devono essere resi disponibili per essere monitorati nell'immediato, questo intervallo di pubblicazione diventa una restrizione significativa.

Nel nostro caso i sensori acquisiscono i dati in tempi molto brevi, per cui tutti quei valori acquisiti dopo una pubblicazione e prima che passino 15 secondi non verranno mai inviati al broker e quindi pubblicati sul cloud.

Per far si che trasmettitore e ricevitore siano sincronizzati si è deciso di intervenire via software implementando uno script che permetta l'invio e la pubblicazione dei dati ogni 15 secondi tramite l'utilizzo della funzione *millis()* e di appositi contatori.

Il ricevitore invierà poi un pacchetto particolare di conferma (ACK), per informare il trasmettitore che ha ricevuto i valori, li ha pubblicati sul cloud ed è pronto a ricevere dei nuovi valori.

Il trasmettitore, non trasmetterà nuovi valori acquisiti dai sensori finché non riceverà il messaggio di conferma da parte del ricevitore.

L'implementazione del codice, sopra, menzionato viene esposto nell'Appendice.

7.5 Considerazioni finali

In conclusione, nonostante le limitazioni, l'adozione delle soluzioni sopra indicate, ha consentito di monitorare i valori di accelerazione e temperatura sul cloud ThingSpeak. Le Figura 7.1, Figura 7.2, Figura 7.3 e Figura 7.4, mostrano quattro grafici raffiguranti un esempio di acquisizione dei quattro valori forniti dai sensori, dopo la pubblicazione di 64 valori.

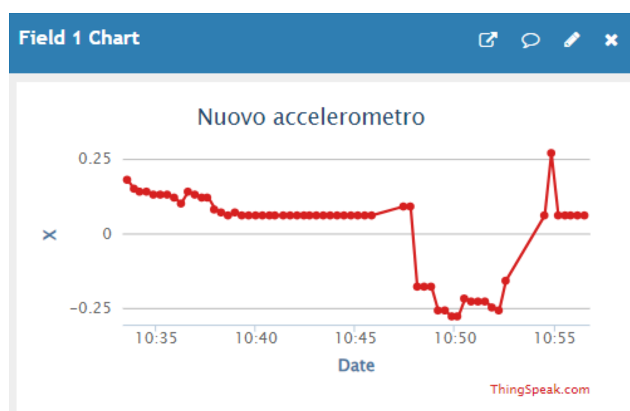

Figura 7.1: Valori asse X

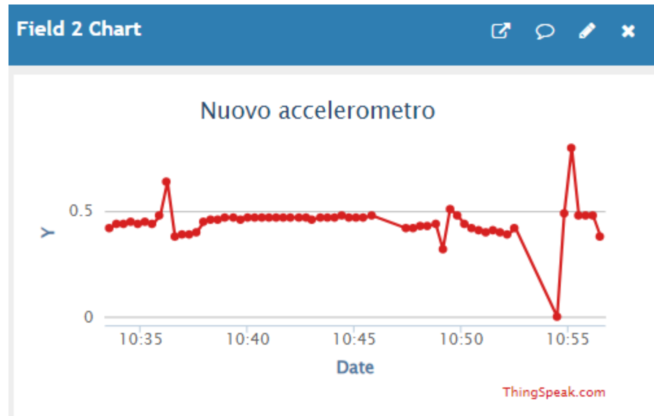

Figura 7.2: Valori asse Y

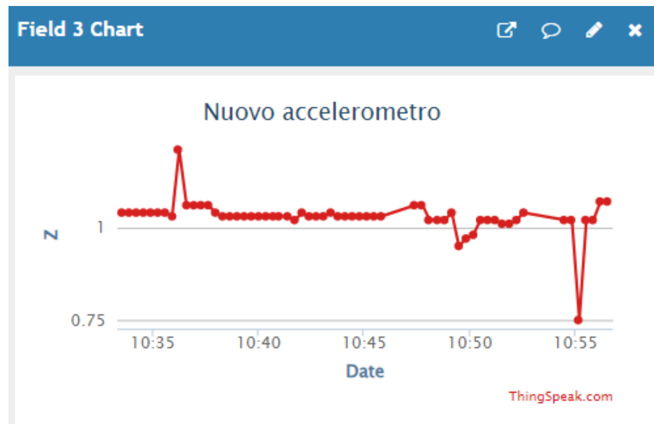

Figura 7.3: Valori asse Z

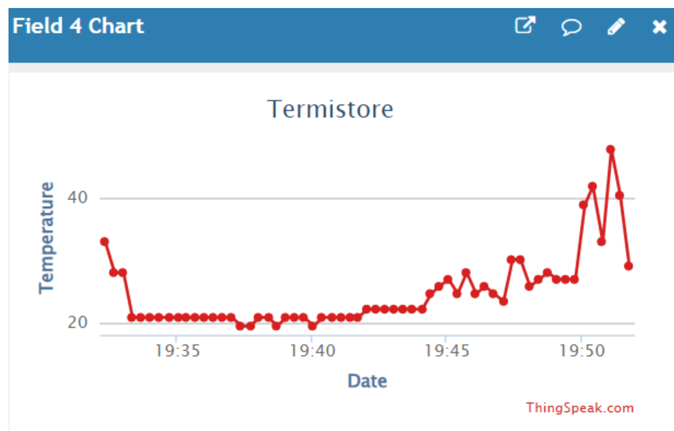

Figura 7.4: Valori di Temperatura

Capitolo 8

Conclusioni

La presente tesi si è focalizzata sull'implementazione della trasmissione dei dati acquisiti dai sensori attraverso l'utilizzo del protocollo MQTT, con particolare attenzione all'interfaccia seriale CANBUS.

Dopo una breve introduzione teorica sui protocolli utilizzati, il lavoro si è concentrato maggiormente su tutto ciò che riguarda la configurazione dei nodi CAN e l'effettiva implementazione della trasmissione MQTT, piuttosto che sul tipo specifico di dati trasmessi.

Lo studio mostra, infatti, l'implementazione di un sistema flessibile e versatile, che consente, con le dovute accortezze, la modifica e l'aggiunta di ulteriori sensori per ottenere un apparato in grado di acquisire una maggiore quantità di dati, che saranno poi disponibili per essere monitorati dall'utente.

I risultati sperimentali hanno dimostrato come l'approccio proposto sia in grado di soddisfare gli obiettivi prefissati, offrendo una trasmissione dati efficiente, leggera e con una bassa latenza. L'architettura risultante si è rivelata, infine, adatta per applicazioni in cui è richiesta una raccolta dati quasi in tempo reale da sensori distribuiti su una rete CANBUS.

Bibliografia

[1] Daniel Kraus, Erich Leitgeb, Thomas Plank, and Markus Löschnigg. Replacement of the controller area network (can) protocol for future automotive bus system solutions by substitution via optical networks. In *2016 18th International Conference on Transparent Optical Networks (ICTON)*, pages 1–8, 2016.

[2] M. van Osch and S.A. Smolka. Finite-state analysis of the can bus protocol. In *Proceedings Sixth IEEE International Symposium on High Assurance Systems Engineering. Special Topic: Impact of Networking*, pages 42–52, 2001.

[3] Dipa Soni and Ashwin Makwana. A survey on mqtt: a protocol of internet of things (iot). In *International conference on telecommunication, power analysis and computing techniques (ICTPACT-2017)*, volume 20, pages 173–177, 2017.

[4] Gaston C Hillar. *MQTT Essentials-A lightweight IoT protocol*. Packt Publishing Ltd, 2017.

[5] Dan Dinculeană and Xiaochun Cheng. Vulnerabilities and limitations of mqtt protocol used between iot devices. *Applied Sciences*, 9(5):848, 2019.

[6] Partha Pratim Ray. A survey of iot cloud platforms. *Future Computing and Informatics Journal*, 1(1-2):35–46, 2016.

[7] Hong-Linh Truong and Schahram Dustdar. Principles for engineering iot cloud systems. *IEEE Cloud Computing*, 2(2):68–76, 2015.

[8] Abdur Rahim Biswas and Raffaele Giaffreda. Iot and cloud convergence: Opportunities and challenges. In *2014 IEEE World Forum on Internet of Things (WF-IoT)*, pages 375–376. IEEE, 2014.

[9] Somayya Madakam, Vihar Lake, Vihar Lake, Vihar Lake, et al. Internet of things (iot): A literature review. *Journal of Computer and Communications*, 3(05):164, 2015.

[10] https://it.wikipedia.org/wiki/Controller_Area_Network.

[11] https://en.wikipedia.org/wiki/CAN_bus.

[12] https://www.dia.uniroma3.it/autom/Reti_e_Sistemi_Automazione/PDF/CAN%20(Applicazioni%20Automobilistiche)%20Di%20Galanti%20Antonello.pdf.

[13] https://www.corsi.univr.it/documenti/OccorrenzaIns/matdid/matdid488545.pdf.

[14] https://www.winstar.com.tw/it/can-bus-interface-communication.

[15] https://mlando.xoom.it/ferrari/can.htm.

[16] https://aws.amazon.com/it/what-is/mqtt/.

[17] https://store.arduino.cc/products/arduino-due.

[18] https://store.arduino.cc/products/arduino-uno-rev3.

[19] https://www.analog.com/media/en/technical-documentation/data-sheets/adxl345.pdf.

[20] https://it.wikipedia.org/wiki/Termistore.

[21] https://it.omega.com/prodinfo/termistori.html.

[22] https://www.circuitbasics.com/arduino-thermistor-temperature-sensor-tutorial/.

[23] https://wiki.seeedstudio.com/CAN-BUS_Shield_V1.2/.

[24] https://www.sparkfun.com/products/13262.

[25] https://docs.arduino.cc/retired/shields/arduino-wifi-shield.

[26] https://docs.arduino.cc/retired/getting-started-guides/ArduinoWiFiShield.

[27] https://www.engineersgarage.com/adxl345-accelerometer-arduino-i2c/.

[28] https://www.engineersgarage.com/adxl345-accelerometer-sensor-how-to-use/.

[29] https://it.mathworks.com/help/thingspeak/mqtt-basics.html.

[30] https://pubsubclient.knolleary.net/api.html#connect.

Appendice

In questa sezione verranno illustrati i codici Arduino completi di entrambi i nodi.

1 Codice Trasmettitore

```
#include <SPI.h>
#include "mcp2515_can.h"
#include <Wire.h>
// Set SPI CS Pin according to your hardware
// For Arduino MCP2515 Hat:
// the cs pin of the version after v1.1 is default to D9
// v0.9b and v1.0 is default D10
const int SPI_CS_PIN = 10;
const int CAN_INT_PIN = 2;

mcp2515_can CAN(SPI_CS_PIN); // Set CS pin

//Defining ADXL345 address
#define ADXL345_ADDRESS 0x53

//Defining ADXL345 data register
#define DATAX0 0x32
#define DATAX1 0x33
#define DATAY0 0x34
#define DATAY1 0x35
#define DATAZ0 0x36
#define DATAZ1 0x37

//Defining thermistor variables
int ThermistorPin = 0;
int Vo;
float R1 = 10000; //known resistor
float logR2, R2, T;
float c1 = 1.009249522e-03, c2 = 2.378405444e-04, c3 = 2.019202697e
    -07;

//Defining publish interval varaibles
unsigned long lastUploadedTime = 0;
const unsigned long postingInterval = 20L * 1000L; // Post data every
    20 seconds.

//*************Setup Loop**************//
void setup() {
```

```
  Serial.begin(9600);
  //Initialize I2C comunication
  Wire.begin();
  //Initialize ADXL345
  setupADXL_345();
  //Initialize CAN transmission
  while (CAN_OK != CAN.begin(CAN_25KBPS)) {
  //init can bus : baudrate = 25k
        Serial.println("CAN init fail, retry...");
        delay(100);
    }
    Serial.println("CAN init ok!");
    delay(100);
}

//***************Main Loop****************//
void loop()
{
  if (millis() - lastUploadedTime >= postingInterval){ // The
   uploading interval must be > 15 seconds
    //read acceleration values
    float X = readX();
    float Y = readY();
    float Z = readZ();
    //Read temperature
    float T = readT();

    //print the values
    printValues(X,Y,Z,T);

    //Create arrays that will be transmitted
    byte XArray[8];
    byte YArray[8];
    byte ZArray[8];
    byte Temp[8];

    //convert float into byte
    memcpy(XArray, &X, sizeof(float));
    memcpy(YArray, &Y, sizeof(float));
    memcpy(ZArray, &Z, sizeof(float));
    memcpy(Temp, &T, sizeof(float));
    //print the arrays
    for (int i = 0; i < sizeof(XArray); i++)
    {
      Serial.print(XArray[i], HEX);
      Serial.print(' ');
    }
    Serial.println(" ");
    for (int i = 0; i < sizeof(YArray); i++)
    {
      Serial.print(YArray[i], HEX);
```

```
      Serial.print(' ');
    }
    Serial.println(" ");
    for (int i = 0; i < sizeof(ZArray); i++)
    {
      Serial.print(ZArray[i], HEX);
      Serial.print(' ');
    }
    for (int i = 0; i < sizeof(Temp); i++)
    {
      Serial.print(Temp[i], HEX);
      Serial.print(' ');
    }
    Serial.println(" ");

    //Send data
    CAN.sendMsgBuf(0x100, 0, 8, XArray);
    delay(500);
    CAN.sendMsgBuf(0x101, 0, 8, YArray);
    delay(500);
    CAN.sendMsgBuf(0x102, 0, 8, ZArray);
    delay(500);
    CAN.sendMsgBuf(0x103, 0, 8, Temp);
    delay(500);
    Serial.println("");

    // Wait ACK from receiver
    while (CAN_MSGAVAIL != CAN.checkReceive()) {
    delay(10);
    }
    unsigned char len;
    unsigned char buffer[8];
    //Read ACK
    CAN.readMsgBuf(&len, buffer);
    if (CAN.getCanId() == 0x555)  //Check ACK
    {
      Serial.println("Pronto a ritrasmettere...");
      lastUploadedTime = millis();
    }
  }
}

/********** Initialize ADXL_345 **********/
void setupADXL_345()
{
  //Set the power mode and data transfer
  Wire.beginTransmission(ADXL345_ADDRESS);
  Wire.write(0x2C);
  Wire.write(0x08);
  Wire.endTransmission();
```

```
  //Set the data format---Full res. 10 bit, values right-justified,
    range +/- 2g
  Wire.beginTransmission(ADXL345_ADDRESS);
  Wire.write(0x31);
  Wire.write(0x08);
  Wire.endTransmission();

  //Set the power saving features: Measure mode
  Wire.beginTransmission(ADXL345_ADDRESS);
  Wire.write(0x2D);
  Wire.write(0x08);
  Wire.endTransmission();

  delay(500);
}

/********* Printing X,Y,Z Values ********/
void printValues(float x,float y,float z,float t)
{
  //Printing values
  //X-axis
  Serial.print("X = ");
  Serial.print(x);
  Serial.print(" g");
  Serial.println();

  //Y-axis
  Serial.print("Y = ");
  Serial.print(y);
  Serial.print(" g");
  Serial.println();

  //Z-axis
  Serial.print("Z = ");
  Serial.print(z);
  Serial.print(" g");
  Serial.println();
  Serial.println();

  //Temperature
  Serial.print("Temperature: ");
  Serial.print(t);
  Serial.println(" C");
  Serial.println();
  Serial.println();
}

/************ Read X-axis ************/
float readX()
{
  //Read from DATAX0
```

```
  Wire.beginTransmission(ADXL345_ADDRESS);
  Wire.write(DATAX0);
  Wire.endTransmission();
  Wire.requestFrom(ADXL345_ADDRESS, 1); //requesting 1 byte from
    ADXL345_ADDRESS
  byte x0 = Wire.read();

  //Read from DATAX1
  Wire.beginTransmission(ADXL345_ADDRESS);
  Wire.write(DATAX1);
  Wire.endTransmission();
  Wire.requestFrom(ADXL345_ADDRESS, 1);
  byte x1 = Wire.read();
  // Since the value is right-adjusted, the 10-bit acceleration is
    obtained
  // by masking the second byte (which is read from 0x33 for the x-
    axis) with 0x03, right-shifting it eight times.
  // Then, the two bytes are combined (0x32 and 0x33) to a 16-bit
    integer.
  x1 = x1 & 0x03;

  // Shifting and combining the two bytes
  uint16_t x = (x1 << 8) | x0;
  int16_t xf = x;

  //The 10-bit value will range from 0 to 1024.
  //The acceleration is measured in both directions along the axis.
  //So, if the value is greater than 511, subtract 1024 from it to get
     a negative value which indicates the other direction of the axis.

  if(xf > 511)
  {
    xf = xf - 1024;
  }

  //For a 10-bit resolution, the value of the acceleration in a
    gravity unit can be derived by multiplying this value with 4 mg
    (0.004) for +/- 2g range.
  float xa = xf * 0.004;
  return xa;
}

/*********** Read Y-axis ************/
float readY()
{
  //Read values from DATAY0
  Wire.beginTransmission(ADXL345_ADDRESS);
  Wire.write(DATAY0);
  Wire.endTransmission();
  Wire.requestFrom(ADXL345_ADDRESS, 1);
  byte y0 = Wire.read();
```

```
  //Read values from DATAY1
  Wire.beginTransmission(ADXL345_ADDRESS);
  Wire.write(DATAY1);
  Wire.endTransmission();
  Wire.requestFrom(ADXL345_ADDRESS, 1);
  byte y1 = Wire.read();
  y1 = y1 & 0x03;

  uint16_t y = (y1 << 8) + y0;
  int16_t yf = y;
  if(yf > 511)
  {
    yf = yf - 1024;
  }
  float ya = yf * 0.004;

  return ya;
}

/********** Read Z-axis **********/
float readZ()
{
  //Read values from DATAZ0
  Wire.beginTransmission(ADXL345_ADDRESS);
  Wire.write(DATAZ0);
  Wire.endTransmission();
  Wire.requestFrom(ADXL345_ADDRESS, 1);
  byte z0 = Wire.read();

  //Read values from DATAZ1
  Wire.beginTransmission(ADXL345_ADDRESS);
  Wire.write(DATAZ1);
  Wire.endTransmission();
  Wire.requestFrom(ADXL345_ADDRESS, 1);
  byte z1 = Wire.read();
  //z1 = z1 & 0x03;

  uint16_t z = (z1 << 8) + z0;
  int16_t zf = z;
  if(zf > 511)
  {
   zf = zf - 1024;
  }
  float za = zf * 0.004;
  return za;
}
/************** Read Temperature ***********/
float readT()
{
  Vo = analogRead(ThermistorPin);
```

```
  R2 = R1 * (1023.0 / (float)Vo - 1.0);
  logR2 = log(R2);
  T = (1.0 / (c1 + c2*logR2 + c3*logR2*logR2*logR2));
  T = T - 273.15;
  return T;
}
```

Listing 1: Codice Trasmettitore

2 Codice Ricevitore

```
#include <SPI.h>
#include "mcp2515_can.h"
#include <WiFi.h>
#include <PubSubClient.h>

/*********** WiFi definitions **********/
int status = WL_IDLE_STATUS;

/****** WiFi Connection Details *******/
const char* ssid = "your_ssid";
const char* password = "your_password";

/******* MQTT Broker Connection Details *******/
const char* mqtt_server = "mqtt3.thingspeak.com";
char mqtt_username[] = "your_mqtt_username";
char mqtt_password[] = "your_mqtt_password";
char clientID[] = "your_mqtt_clientID";
int mqtt_port = 1883;
long channelID = your_channelID;
char* Topic = "channels/channelID/publish";

/***** Defining Publishing interval ***********/
unsigned long lastUploadedTime = 0;
const unsigned long postingInterval = 20L * 1000L; // Post data every
    20 seconds.

// Initialize the WiFi client library
WiFiClient wificlient;
PubSubClient client(wificlient);

/*************** CAN bus definition *************/
// Set SPI CS Pin according to your hardware
// For Arduino MCP2515 Hat:
// the cs pin of the version after v1.1 is default to D9
// v0.9b and v1.0 is default D10
const int SPI_CS_PIN = 10;
const int CAN_INT_PIN = 2;

//defining packets identificator
```

```
#define X_ID 0x100
#define Y_ID 0x101
#define Z_ID 0x102
#define ACK_ID 0x555

mcp2515_can CAN(SPI_CS_PIN); // Set CS pin

//Initializing functions
void setup_wifi();
void printWifiStatus();
void callback();
void reconnect();
void publishMessage();
void SetSockStatus();
void ShowSockStatus();

unsigned char ack[8]={0,0,0,0,0,0,0,0};

/********** Setup Loop *********/
void setup() {
    Serial.begin(9600);
    setup_wifi();
    client.setServer(mqtt_server, mqtt_port);
    client.setCallback(callback);
    while (CAN_OK != CAN.begin(CAN_25KBPS)) { // init can bus :
    baudrate = 500k
        Serial.println("CAN init fail, retry...");
        delay(100);
    }
    Serial.println("CAN init ok!");
    CAN.sendMsgBuf(ACK_ID,0,8,ack);
    delay(1000);
}
//Defining 3-axis variables
float X ;
float Y ;
float Z ;
//Defining Temperature variable
float T ;

/********* Main Loop *******/
void loop()
{
  //Check if client connected
  if (!client.connected()) { //if not
    reconnect();
  }
  client.loop();
  //Creating variables for buffers
  unsigned char len = 0;
  unsigned char buffer[8];
```

```
unsigned char xbuf[8];
unsigned char ybuf[8];
unsigned char zbuf[8];
unsigned char tbuf[8];

//Set interval for uploading
if (millis() - lastUploadedTime > postingInterval) // The uploading
  interval must be > 15 seconds
{
  if (CAN_MSGAVAIL == CAN.checkReceive()) //Check coming packets
  {
      for (int j=0;j<4;j++) //Read 4 packets X,Y,Z,T
      {
          CAN.readMsgBuf(&len, buffer);    // read data,  len: data
  length, buf: data buf
          unsigned long canId = CAN.getCanId();
          // Check if payload is non-zero
          if (canId == X_ID) //X packet
          {
              Serial.println("-----------------------------");
              Serial.print("Get data from ID: 0x");
              Serial.println(canId, HEX);
              for (int i = 0; i < len; i++) {
                  xbuf[i] = buffer[i];
                  Serial.print(xbuf[i], HEX);
                  Serial.print("\t");
              }
              Serial.println();
              /********** Form bytes to float **********/
              memcpy(&X,xbuf,sizeof(float));
              Serial.print("X = ");
              Serial.print(X);
              Serial.print(" g");
              Serial.println();
          }
          if (canId == Y_ID) //Y packet
          {
              Serial.println("-----------------------------");
              Serial.print("Get data from ID: 0x");
              Serial.println(canId, HEX);
              for (int i = 0; i < len; i++) { // print the data
                ybuf[i] = buffer[i];
                Serial.print(ybuf[i], HEX);
                Serial.print("\t");
              }
              Serial.println();
              /********** Form bytes to float **********/
              memcpy(&Y,ybuf,sizeof(float));
              Serial.print("Y = ");
              Serial.print(Y);
```

```
                Serial.print(" g");
                Serial.println();
            }
            if (canId == Z_ID)  //Z packet
            {
              Serial.println("-----------------------------");
              Serial.print("Get data from ID: 0x");
              Serial.println(canId, HEX);
              for (int i = 0; i < len; i++) { // print the data
                zbuf[i] = buffer[i];
                Serial.print(zbuf[i], HEX);
                Serial.print("\t");
              }
              Serial.println();
              /********* Form bytes to float **********/
              memcpy(&Z,zbuf,sizeof(float));
              Serial.print("Z = ");
              Serial.print(Z);
              Serial.print(" g");
              Serial.println();
            }
            if (canId == T_ID)  //Temp packet
            {
              Serial.println("-----------------------------");
              Serial.print("Get data from ID: 0x");
              Serial.println(canId, HEX);
              for (int i = 0; i < len; i++) { // print the data
                tbuf[i] = buffer[i];
                Serial.print(tbuf[i], HEX);
                Serial.print("\t");
              }
              Serial.println();
              /********* Form bytes to float **********/
              memcpy(&T,tbuf,sizeof(float));
              Serial.print("T = ");
              Serial.print(T);
              Serial.print(" C");
              Serial.println();
            }
            delay(500);
        }
    }
    Serial.println(X);
    Serial.println(Y);
    Serial.println(Z);
    Serial.println(T);
    //send ACK
    CAN.sendMsgBuf(ACK_ID,0,8,ack);
    //publish
    publishMessage(X,Y,Z,T);
    //free the socket
```

```
    SetSockStatus();
    lastUploadedTime = millis();
  }
}

/************* Connect to MQTT Broker ***********/
void reconnect() {
  // Loop until we're reconnected
  while (!client.connected()) {
    // Attempt to connect
    Serial.print("Attempting MQTT connection...");
    //Initialize Socket
    SetSockStatus();
    if (client.connect(clientID, mqtt_username, mqtt_password)) {
      Serial.println("connected");
      ShowSockStatus();
    }
    else {
      Serial.print("failed, rc=");
      Serial.print(client.state());
      Serial.println(" try again in 5 seconds");
      // Wait 5 seconds before retrying
      delay(5000);
    }
  }
}

/**** Method for Publishing MQTT Messages **********/
void publishMessage(float x, float y, float z, float t)
{
  //Creating the string
  String data = "field1=";
  data += String(x);
  data += "&field2=";
  data += String(y);
  data += "&field3=";
  data += String(z);
  data += "&field4=";
  data += String(t);
  int lunghezza_dato = data.length();
  char messaggio[lunghezza_dato];
  data.toCharArray(messaggio, lunghezza_dato + 1);
  Serial.println(messaggio);
  //publish message
  client.publish(Topic,messaggio);
  Serial.println("Message published ["+String(Topic)+"]: "+messaggio);
}

/************* Connect to WiFi ***********/
void setup_wifi() {
  delay(10);
```

```
 // attempt to connect to WiFi network:
  while (status != WL_CONNECTED) {
    Serial.print("Attempting to connect to SSID: ");
    Serial.println(ssid);
    // Connect to WPA/WPA2 network. Change this line if using open or
    WEP network:
    status = WiFi.begin(ssid, password);
    // wait 5 seconds for connection:
    delay(5000);
  }
  Serial.println("Connected to WiFi");
  printWifiStatus();
}

/************ Print WiFi status ****************/
void printWifiStatus() {
  // print the SSID of the network you're attached to:
  Serial.print("SSID: ");
  Serial.println(WiFi.SSID());

  // print your WiFi shield's IP address:
  IPAddress ip = WiFi.localIP();
  Serial.print("IP Address: ");
  Serial.println(ip);

  // print the received signal strength:
  long rssi = WiFi.RSSI();
  Serial.print("signal strength (RSSI):");
  Serial.print(rssi);
  Serial.println(" dBm");
}

/*********** Show Socket Status ******/
void ShowSockStatus() {
  for(int x = 0; x < MAX_SOCK_NUM; x++) {
    Serial.print(F("Socket #"));
    Serial.print(x,DEC);
    if(WiFi._state[x] == -1) Serial.print(F(": available  port "));
    else Serial.print(F(": used  port "));

    Serial.println(WiFi._server_port[x]);
  }
}

/************ Make Socket Available *************/
void SetSockStatus() {
  for(int x = 0; x < MAX_SOCK_NUM; x++) {
    WiFi._state[x] = -1;
  }
}
```

Listing 2: Codice Ricevitore