Università Politecnica delle Marche

Facoltà di Ingegneria

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

Tesi di Laurea

Analisi e prototipazione di una soluzione embedded per la misura di dati accelerometrici strutturali su device constrained con comunicazione radio

Analysis and prototyping of an embedded solution for the measurement of structural accelerometric data on devices constrained with radio communication

Relatore

Prof. Domenico Ursino

Candidato

Claudio Menotta

Correlatore

Ing. Emiliano Anceschi

Anno Accademico 2020-2021

Indice

Introduzione . . . 9

1 Definizione del contesto . . . 13
- 1.1 Monitoraggio strutturale . . . 13
- 1.2 Panoramica dell'infrastruttura digitale . . . 14

2 Analisi del Gateway . . . 21
- 2.1 Standard di Messaggistica MQTT . . . 21
  - 2.1.1 Client e broker . . . 22
  - 2.1.2 Messaggi fondamentali . . . 23
- 2.2 Sviluppo del Gateway . . . 24
- 2.3 Configurazione in servizi e librerie da installare nel Gateway . . . 33

3 Rete di comunicazione LoRaWAN . . . 35
- 3.1 Protocollo di comunicazione LoRAWAN . . . 35
  - 3.1.1 LoRa e LoRaWAN . . . 35
  - 3.1.2 Architettura del protocollo LoRaWAN . . . 36
  - 3.1.3 Tipi di messaggio . . . 38
  - 3.1.4 Messa in servizio di un dispositivo . . . 38
  - 3.1.5 Classi A, B e C . . . 40
- 3.2 Configurazione di Chirpstack . . . 41
- 3.3 Doppia funzionalità del Gateway . . . 44

4 Analisi End-Node . . . 47
- 4.1 STM32 BL072Z . . . 47
  - 4.1.1 Ricezione messaggi LoRaWAN . . . 47
  - 4.1.2 Invio dei messaggi tramite UART . . . 52
- 4.2 Collegamento fisico tra STM32 e Raspberry all'interno dell'End-Node 57
- 4.3 Scelta del convertitore MCC172 . . . 57
- 4.4 Raspberry Pi 3B+ e MCC172 . . . 61
- 4.5 L'accelerometro ADXL354 . . . 70

5 Progettazione della componente software dell'End-Node 73
5.1 Analisi della libreria `mcc172.c` 73
5.2 Valutazione dei risultati con un'unica HAT 84
5.3 Valutazione dei risultati con 2 HAT 89
5.4 Prova al passaggio del kernel in Real Time 91
5.5 Modifica di `mcc172.c` per alleggerire il carico dell'acquisizione 93

6 Implementazione della componente software sull'End-Node 103
6.1 Aggiunta dei segnali di sincronizzazione 103
6.2 Aggiunta delle funzioni per il salvataggio dei dati nel pre e post TRIGGER 106
6.3 Script per il caricamento del `.csv` sull'FTP server 107
6.4 Configurazione in servizi e librerie da installare nell'End-Node 109

7 Testing del sistema 111
7.1 Avvio dell'acquisizione da ALGIZ.STRUCT 111
7.2 Avvio dell'applicazione su End-Node 113
7.3 Presentazione del file `.csv` creato 115

Conclusioni 121

Riferimenti bibliografici 123

Ringraziamenti 125

Elenco delle figure

1.1 Schema a blocchi dell'infrastruttura digitale 14
1.2 Raspberry Pi 3 B+ 15
1.3 Stack tecnologico LoRaWAN 15
1.4 Elementi della rete LoRaWAN 16
1.5 LoRa Picocell Gateway 17
1.6 STM32 BL072Z 18

2.1 Architettura di pubulish/subscribe MQTT 22
2.2 Attributi di un messaggio *PUBLISH* 24
2.3 Lista dei possibili template per un progetto 25
2.4 Possibile configurazione di un template 26

3.1 Tecnologie utilizzate per la trasmissione di dati wireless 36
3.2 Flusso di messaggi per l'Over-The-Air Activation (OTAA) 39
3.3 Pre-condivisione di DevAddr e delle chiavi di sessione per l'Activation By Personalization (ABP) 40
3.4 Configurazione del Network Server in Chirpstack 42
3.5 Configurazione del Gateway in Chirpstack 42
3.6 Configurazione di un nuovo *Device-profile* in Chirpstack 43
3.7 Configurazione della procedura di join per gli End-Node in Chirpstack 43
3.8 Definizione del *Device address*, della *Network session key* e della *Application session key* in Chirpstack 44
3.9 Definizione dei parametri del Multicast in Chirpstack 44

4.1 Architettura del pacchetto di espansione I-CUBE-LRWAN 48
4.2 Struttura dei file di progetto 50
4.3 Convertitore MCC172 collegato alla Raspberry Pi 57
4.4 Stack di 4 HAT MCC172 58
4.5 Diagramma a blocchi che descrive la HAT MCC172 59
4.6 Elenco delle funzioni principali della libreria mcc172.c 62
4.7 Accelerometro ADXL354 71
4.8 Verso degli assi di misurazione dell'accelerometro ADXL354 71
4.9 Diagramma funzionale dell'header P1 71

4.10 Diagramma funzionale dell'header P2 72

5.1 Campioni disponibili su *info→scan_buffer* dopo l'attivazione della funzione *mcc172_a_in_scan_read()* 87
5.2 Visualizzazione, tramite oscilloscopio, di una delle interruzioni dell'SPI 88
5.3 Campioni disponibili su *info→scan_buffer*, per entrambe le HAT, dopo l'attivazione della funzione *mcc172_a_in_scan_read()* 90
5.4 Campioni disponibili su *info→scan_buffer*, per entrambe le HAT, dopo l'attivazione della funzione *mcc172_a_in_scan_read()*, utilizzando la versione Real Time del sistema operativo Raspbian Lite 93
5.5 Schema a blocchi per riassumere il processo di acquisizione dei campioni 94
5.6 Campioni inviati dalla MCC172 alla Raspberry in ciascuna lettura, per entrambe le HAT, utilizzando la versione Real Time del sistema operativo Raspbian Lite e la versione modificata della libreria `mcc172.c` 102

7.1 Output prodotto dallo script `send_synch.py` nel file `output.txt` 112
7.2 Output prodotto dallo script `send_synch.py` nel file `logfile.txt` . . . 113
7.3 Output sullo schermo all'avvio dell'applicazione sulla Raspberry 113
7.4 Output sullo schermo della Raspberry all'aggiunta dell'End-Node su ALGIZ.STRUCT 114
7.5 Output sullo schermo della Raspberry all'arrivo dei dati di configurazione del template 114
7.6 Output sullo schermo della Raspberry all'arrivo del segnale di TRIGGER 115
7.7 Output sullo schermo della Raspberry all'arrivo dei segnali di BEACON e di END 115
7.8 Visualizzazione delle prime 5 righe del dataset 116
7.9 Visualizzazione della tensione analogica acquisita sul canale CH0 della HAT0, proporzionale all'accelerazione misurata sull'asse x 117
7.10 Visualizzazione della tensione analogica acquisita sul canale CH1 della HAT0, proporzionale all'accelerazione misurata sull'asse y 117
7.11 Visualizzazione della tensione analogica acquisita sul canale CH0 della HAT1, proporzionale all'accelerazione misurata sull'asse z 117
7.12 Numero di campioni inviati, in ciascuna lettura, dalla HAT0 e dalla HAT1 alla Raspberry 118
7.13 Visualizzazione del conteggio dei BEACON, determinando, per entrambe le HAT, l'esatto istante di campionamento di arrivo di quest'ultimo 119

Elenco dei listati

2.1 Codice presente nel file `main.py` del Gateway 26
2.2 Codice presente nel file `Template_Model.py` del Gateway 27
2.3 Codice presente nel file `send_synch.py` del Gateway 29
2.4 Configurazione del servizio *gateway* 33
2.5 Configurazione del servizio *packet_forwarder* 33
2.6 Configurazione del servizio *wvdial* 34
2.7 Avvio dei servizi 34
4.1 Codice inserito nella funzione *Lora_Init()* per abilitazione e configurazione del Multicast 51
4.2 Definizione delle funzioni utilizzate per l'inizializzazione dell'UART . . 52
4.3 Implementazione della funzione *Lora_RxData()* utilizzata per elaborare i frame in arrivo dal Gateway 52
4.4 Definizione della funzione per la verifica della presenza dell'ID della scheda all'interno della lista ricevuta 55
4.5 Implementazione del *main()* in esecuzione nella Raspberry presente nell'End-Node 64
4.6 Implementazione del thread *Measurement* in esecuzione nella Raspberry presente nell'End-Node 65
4.7 Implementazione della funzione *Init()* presente nel file `Measurement.c` 65
4.8 Implementazione della funzione *Check_Acquisition()* presente nel file `Measurement.c` 66
4.9 Implementazione del thread *UART_thread()* presente nel file `UART_thread.c` 68
5.1 Implementazione della funzione *mcc172_a_in_scan_start()* presente nella libreria `mcc172.c` 74
5.2 Implementazione della funzione *_scan_thread()* presente nella libreria `mcc172.c` 77
5.3 Implementazione della funzione *mcc172_a_in_scan_read()* presente nella libreria `mcc172.c` 81
5.4 Utilizzo della funzione *mcc172_a_in_scan_read()* nell'esempio di acquisizione continua su una sola HAT `continuous_scan.c` 85

5.5 Inserimento della funzione *mcc172_a_in_scan_status()* nel loop di lettura, nell'esempio di acquisizione continua su una sola HAT `continuous_scan.c` . . . 87
5.6 Utilizzo della funzione *mcc172_a_in_scan_read()* nell'esempio di acquisizione continua con 2 HAT nel file `continuous_scan_multi_hat.c` . . . 89
5.7 Download del codice del kerner Real-Time . . . 91
5.8 Compilazione del kerner Real-time con impostazioni di default . . . 91
5.9 Apertura del menù per la configurazione del kernel . . . 92
5.10 Modifica per visualizzare quale kernel è in esecuzione . . . 92
5.11 Installazione del kernel . . . 92
5.12 Modifica del codice della funzione *mcc172_a_in_scan_start()* all'interno della libreria `mcc172.c` . . . 95
5.13 Modifica del codice della funzione *_scan_thread* all'interno della libreria `mcc172.c` . . . 96
5.14 Funzione *mcc172_save_data()* aggiunta all'interno della libreria `mcc172.c` . . . 98
5.15 Modifica della funzione *mcc172_close()* all'interno della libreria `mcc172.c` . . . 100
5.16 Codice da inserire dopo la funzione *mcc172_a_in_scan_start()* per testare la modifica della libreria `mcc172.c` . . . 101
6.1 Allocazione in memoria per il vettore *info→time* all'interno di *mcc172_a_in_scan_start()* . . . 104
6.2 Codice aggiunto a *_scan_thread* per la scrittura del vettore *info→time* 104
6.3 Dichiarazione delle funzioni per la gestione del *beacon_mutex* all'interno della libreria `mcc172.c` . . . 106
6.4 Definizione della funzione *mcc172_save_data_post_t()* all'interno della libreria `mcc172.c` . . . 106
6.5 Riga di codice aggiunta nel thread *Measurement* per attivare lo script Python che si occuperà del caricamento del file . . . 107
6.6 Script `data_upload.py` usato per il caricamento del file sull'FTP server . . . 108
6.7 Codice presente all'interno del `Makefile` . . . 109
6.8 Configurazione del servizio *node-accelerometro* . . . 109
6.9 Installazione della libreria `daqhats` . . . 110
6.10 Disabilitazione del bluetooth . . . 110

Introduzione

Il problema delle vibrazioni degli edifici ha assunto, negli ultimi anni, sempre maggiore importanza, sia in relazione alla diversa tipologia strutturale delle costruzioni moderne, più snelle e più leggere grazie ad un più razionale utilizzo dei materiali con migliori caratteristiche di resistenza meccanica, sia in relazione al moltiplicarsi delle fonti di vibrazione, in special modo quelle generate dalle attività dell'uomo, come quelle derivanti dalle attività di cantiere, dal funzionamento di macchine e dal traffico stradale e ferroviario, che possono essere causa di disturbo e apprensione degli occupanti di edifici. Ciò può portare alla necessità di verificare se le vibrazioni siano tali da indurre o meno danni alla costruzione, soprattutto in presenza di evidenti danni architettonici generati da altre cause.

La misurazione delle vibrazioni degli edifici può essere finalizzata a diversi obiettivi:

- *Riconoscimento del problema*: per valutare se i livelli di vibrazione riscontrati possano determinare danni all'edificio o limitarne la funzionalità specifica, rendendo, quindi, necessario un approfondimento dello studio.
- *Verifiche o controlli*: per rapportare il livello delle vibrazioni ai limiti suggeriti o imposti da normative specifiche, relative per esempio alle condizioni di esercizio delle apparecchiature.
- *Caratterizzazione a scopo di diagnostica*: per verificare, rispetto a ipotesi progettuali o a condizioni precedentemente note, l'insorgenza di modifiche strutturali dovute a carichi accidentali severi (per esempio terremoti) o al degrado dei materiali.
- *Caratterizzazione a scopo di previsione*: per valutare l'attitudine dell'edificio a sopportare carichi dinamici accidentali, quali le raffiche di vento, o per stimare l'efficacia di provvedimenti per l'attenuazione dei fenomeni vibratori. Una tale caratterizzazione può, anche, essere effettuata al solo scopo di ottenere informazioni sulle proprietà strutturali dell'edificio, attraverso la stima dei suoi parametri dinamici.

A tali scopi, è stato realizzato il progetto che verrà spiegato e analizzato in questa tesi. In particolare, l'obiettivo è quello di riuscire a raccogliere, quando desiderato, i valori di accelerazione lungo i tre assi cartesiani, quindi le vibrazioni, in più punti

dell'edificio, sincronizzando tra di loro questi dati e ottenendo, così, un modello vibrazionale completo dell'edificio.

In questa tesi non verrà trattata la parte riguardante le proprietà strutturali e le conclusioni derivanti dall'analisi dei dati, di competenza di ingegneri strutturali. Verrà, invece, presentata e analizzata l'intera infrastruttura hardware e software utilizzata. Essa è stata pensata e strutturata per rispondere alle esigenze richieste.

In particolare, l'esigenza principale è quella di poter avviare l'acquisizione delle vibrazioni tramite un portale web, comandato dall'utente. Da esso si può determinare su quale edificio effettuare il monitoraggio e quali dei punti di acquisizione abilitare, tra quelli disponibili. Nell'applicazione web si effettua, semplicemente, la configurazione di una o più acquisizioni, che verrà, poi, inoltrata, tramite un broker MQTT, ad un Gateway, costituito da una Raspberry Pi 3B+, in grado di elaborare queste informazioni e schedulare delle attività di acquisizione. Il Gateway, inoltre, svolge la funzione di Gateway LoRaWAN, tramite l'utilizzo di un'antenna Picocell SX1308, in grado di mettere in comunicazione il Network Server LoRaWAN con gli End-Node LoRaWAN, che sono, appunto, tutti i nodi di acquisizione.

Proprio il protocollo LoRaWAN è stato utilizzato per inviare le informazioni dal Gateway agli End-Node. Nel mondo IoT e in quello del monitoraggio strutturale esistono diverse soluzioni per far comunicare elementi posti a diverse decine di metri l'uno dall'altro. In questo caso è stato scelto il protocollo LoRaWAN per i suoi bassi costi di sviluppo, essendo completamente open source, per il lungo raggio di lavoro, che raggiunge anche i $15Km$, e per i suoi bassi requisiti di potenza.

Gli End-Node, ovvero tutti i dispositivi che si occupano di effettuare l'acquisizione vera e propria, sono attivati da segnali radio LoRaWAN provenienti dal Gateway. Gli End-Node, dal momento della loro accensione, acquisiscono le accelerazioni sui tre assi cartesiani, con una frequenza di campionamento di 1000 Hz, e, nel momento di arrivo del segnale di attivazione da parte del Gateway, iniziano il salvataggio dei dati acquisiti, memorizzando anche i 20 secondi di acquisizioni precedenti all'arrivo del segnale. La durata dal campionamento è determinata nell'applicazione web, e viene, poi, gestita dal Gateway. Quest'ultimo, oltre al segnale di avvio, fornisce periodicamente dei segnali per la sincronizzazione delle acquisizioni degli End-Node attivati e, una volta trascorso il tempo prefissato, viene mandato il segnale di stop.

La misura delle vibrazioni viene effettuata attraverso l'utilizzo dell'accelerometro ADXL354 e di due schede di acquisizione Analogico-Digitale MCC172 ad alta precisione, che vengono montate sopra una Raspberry Pi 3B+. Inoltre, nell'End-Node è presente anche un microcontrollore STM32 BL072Z, in grado di gestire la ricezione dei messaggi radio LoRaWAN e di comunicarli, tramite protocollo UART, alla Raspberry.

Ovviamente, i dati acquisiti devono essere resi disponibili all'utente e non rimanere confinati all'interno dell'End-Node; perciò, essi vengono caricati all'interno di un FTP server aziendale. Ciò è possibile grazie all'utilizzo di un Modem 4G che fornisce la connessione ad Internet a tale componente.

In questa tesi non è stata affrontata l'intera progettazione dell'infrastruttura hardware e software appena presentata. In particolare, lo sviluppo dell'applicazione web, del Gateway e della comunicazione tra i due è stato effettuato da altre persone presenti all'interno di Smart Space, sede del tirocinio e dello sviluppo del progetto.

Il progetto ha avuto come obiettivo principale lo sviluppo dell'End-Node, con le funzionalità sopra descritte, e della gestione della comunicazione tra i due microcontrollori presenti al suo interno. Il tutto ha superato, ovviamente, i test di validazione prefissati.

Prima di procedere nella realizzazione di quanto detto sopra, nel corso del tirocinio si è speso diverso tempo per analizzare quanto era già stato sviluppato e per studiare le tecnologie utilizzate, per riuscire a completare al meglio la parte principale del progetto, ovvero l'ideazione dell'End-Node. Perciò, nella presente tesi, prima di affrontare nel dettaglio la parte sviluppata, viene fatta una breve analisi del Gateway e del protocollo di comunicazione LoRaWAN utilizzato.

La seguente tesi è sviluppata come di seguito specificato:

- Nel Capitolo 1 viene descritta in maniera generale l'intera infrastruttura hardware e software, indicando le principali funzionalità di ciascun elemento.
- Nel Capitolo 2 viene effettuata una breve analisi dell'applicazione web ALGIZ.STRUCT, viene descritto il protocollo MQTT utilizzato per la comunicazione di quest'ultima con il Gateway, ed, infine, viene analizzato il codice presente all'interno del Gateway.
- Nel Capitolo 3 viene descritto il protocollo LoRaWAN e viene mostrata la configurazione effettuata in Chirpstack per la realizzazione del Network Server e, per la gestione del Gateway LoraWAN.
- Nel Capitolo 4 viene effettuata un'analisi completa dell'End-Node, con i codici presenti all'interno del microcontrollore STM32 e all'interno della Raspberry. Inoltre, viene descritto il protocollo utilizzato per la loro comunicazione, e vengono mostrate le principali caratteristiche dell'ADC MCC172 e dell'accelerometro ADXL354.
- Nel Capitolo 5 viene mostrata la fase di progettazione della componente software dell'End-Node, insieme alle problematiche prestazionali riscontrate e alle soluzioni proposte e sviluppate.
- Nel Capitolo 6 viene descritta l'implementazione del software progettato per l'End-Node. In particolare, la libreria e le funzioni progettate devono essere implementate all'interno del contesto generale di funzionamento.
- Nel Capitolo 7 vengono mostrati i risultati dei test effettuati.
- Infine, vengono tratte le conclusioni e vengono presentati dei possibili sviluppi futuri.

1

Definizione del contesto

In questo primo capitolo verrà descritto il contesto di lavoro all'interno del quale si andrà ad operare, ovvero quello del monitoraggio strutturale. Successivamente verrà illustrata una panoramica dell'infrastruttura informatica e digitale utilizzata per lo sviluppo del progetto, formata da due Nodi comunicanti tramite protocollo LoRa, che sono il Gateway e l'End-Node.

1.1 Monitoraggio strutturale

La diagnostica delle costruzioni è una disciplina che, attraverso l'applicazione di differenti tecniche di indagine, si pone l'obiettivo di raccogliere le informazioni utili all'individuazione delle cause dei fenomeni patologici di cui le costruzioni sono affette. Essa regola lo studio e i rilievi sugli edifici al fine di individuare degrado e vulnerabilità che possono compromettere la stabilità dell'edificio o ridurre la sicurezza per le persone che ci vivono.

La diagnostica delle costruzioni si avvale di diversi strumenti per il rilievo, tra cui si trovano gli accelerometri per l'analisi delle vibrazioni, come nel caso preso in considerazione. Le competenze di questo settore riguardano il campo della tecnologia dei materiali, la scienza e la tecnica delle costruzioni, la fisica degli edifici, le tecniche analitiche e di misura, i metodi di simulazione e di elaborazione dei dati [4].

Pur servendosi di metodi rigorosi e scientificamente sperimentati, la diagnostica degli edifici richiede necessariamente l'intervento di tecnici specializzati e ingegneri strutturali che possano sintetizzare le informazioni ottenute al termine dell'analisi per emettere un giudizio professionale, attraverso un'analisi dei dati molto approfondita.

In particolare in questo progetto l'obiettivo è quello di riuscire a raccogliere, quando desiderato, i valori di accelerazione lungo i tre assi cartesiani, e quindi le vibrazioni, in più punti dell'edificio, e riuscire a sincronizzare questi dati ottenuti in più punti per sviluppare un modello vibrazionale dell'edificio.

Il risultato dell'indagine consente di individuare i difetti di costruzione e i fenomeni di degrado e dissesto che possono interessare i singoli materiali o i prodotti del processo di costruzione durante l'intero ciclo di vita, permettendo di proporre soluzioni adeguate per il problema rilevato.

In questa tesi non verrà trattata la parte riguardante le proprietà strutturali e le conclusioni derivanti dall'analisi dei dati, di competenza di ingegneri strutturali. Verrà, invece, presentata e analizzata l'intera infrastruttura hardware e software utilizzata. In linea generale, essa è composta principalmente da due tipi di Nodi, ovvero il Gatway e l'End-Node, e dalla parte di gestione di comunicazione tramite protocollo LoRa che li collega. In particolare, ciascun End-Node acquisirà in maniera continua i dati di accelerazioni nel punto in cui è posizionato e, nel momento che riceverà il segnale di TRIGGER da parte del Geteway, inizierà il salvataggio dei dati, che comprenderà i 20 secondi precedenti all'arrivo del segnale e i successivi 20 minuti. Durante questo tempo, il Gateway continuerà a mandare dei segnali di sincronizzazione, definiti BEACON, per poter poi sincronizzare i dati acquisiti dai vari End-Node. I dati relativi all'acquisizione che arrivano al Gateway vengono impostati da un operatore attraverso un portale web.

1.2 Panoramica dell'infrastruttura digitale

Per comprendere pienamente la parte di progetto sviluppata è importante definire prima nel dettaglio le varie componenti hardware e software che costituiscono il Gateway e l'End-Node, per poi vedere, nei successivi capitoli, un'analisi dettagliata dei codici sviluppati.

In figura 1.1 è rappresentata l'intera infrastruttura digitale, con le componenti principali sia del Gateway che dell'End-Node.

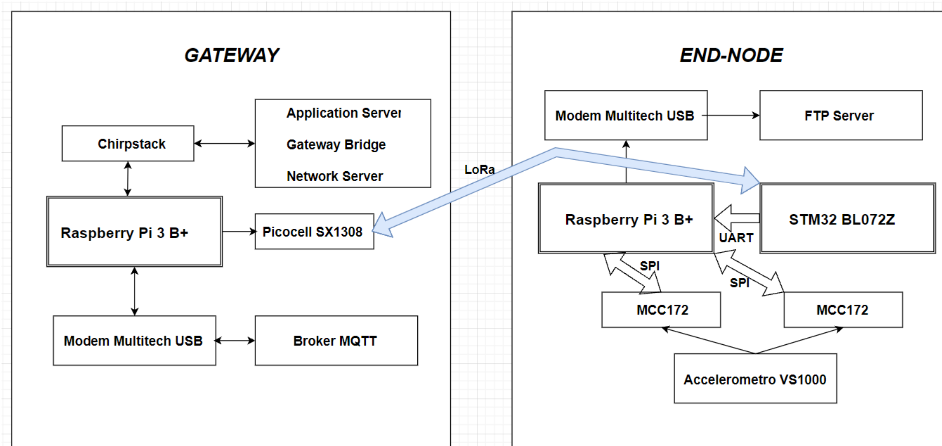


Figura 1.1. Schema a blocchi dell'infrastruttura digitale

Il Nodo Gateway è costituito da un unico microcontrollore, ovvero una Raspberry Pi 3 B+ (Figura 1.2), un'antenna Picocell SX1308 e da un Modem Multitech 4G.

La presenza del Gatetway è necessaria per avere un nodo centrale in grado di comunicare con tutti gli End-Node, che possono appartenere anche a edifici diversi

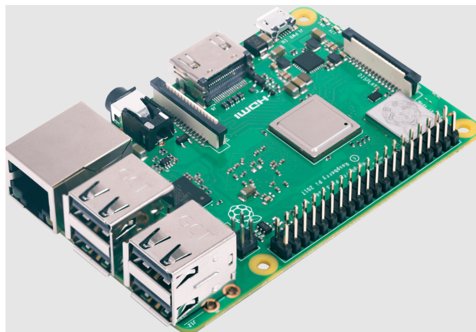


Figura 1.2. Raspberry Pi 3 B+

(ovvero diversi progetti), in modo da inoltrare tutte le informazioni necessarie per l'acquisizione e dare la possibilità di sincronizzare i campioni acquisiti su ciascun nodo. Nel mondo IoT esistono diverse tecnologie di rete che vengono tradizionalmente usate come soluzione per connettere due Nodi (in questo caso, Gateway ed End-Node), tra cui, anche, il protocollo LoRaWAN. LoRa è una tecnologia di modulazione RF (Radio Frequency) per reti WAN (Wide Area Network) a bassa potenza (LPWAN). Creato da Semtech per standardizzare le LPWAN, LoRa fornisce comunicazioni a lungo raggio, ovvero fino a cinque chilometri nelle aree urbane e fino a 15 chilometri o più nelle aree rurali [11].

Una caratteristica chiave delle soluzioni basate su LoRa sono i requisiti di potenza estremamente bassi, che consentono la creazione di dispositivi alimentati a batteria che possono durare fino a 10 anni. Una rete basata sul protocollo LoRaWAN è perfetta per applicazioni che richiedono comunicazioni a lungo raggio o penetranti all'interno dell'edificio tra un gran numero di dispositivi che hanno requisiti di bassa potenza e che scambiano piccole quantità di dati.

In questo capitolo non verranno affrontati nel dettaglio la modulazione LoRa e le sue proprietà, che verranno viste, invece, nel Capitolo 3. È importante però per capire appieno il progetto, dare uno sguardo fin da subito allo stack tecnologico LoRaWAN.

Come mostrato nella Figura 1.3, LoRa è il livello fisico (PHY), ovvero la modulazione wireless utilizzata per creare il collegamento di comunicazione a lungo raggio.

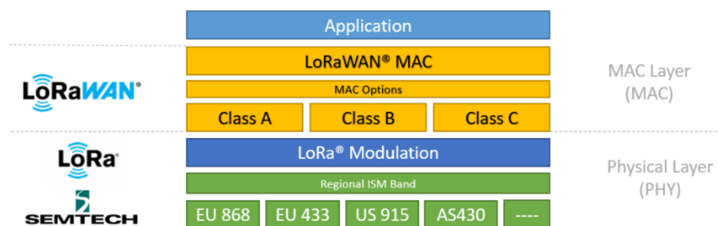


Figura 1.3. Stack tecnologico LoRaWAN

LoRaWAN è, invece, il protocollo di rete aperto che fornisce servizi di comunicazione bidirezionali, mobilità e localizzazione sicuri, standardizzati e gestiti da LoRa Alliance. Esso è il livello MAC (Media Access Control) basato sulla modulazione LoRa, ovvero è il livello software che definisce come i dispositivi utilizzano l'hardware LoRa, ad esempio quando trasmettono, e il formato dei messaggi.

Nella Figura 1.4 sono descritti quali sono gli elementi principali che definiscono una rete basata sul protocollo LoRaWAN [15].

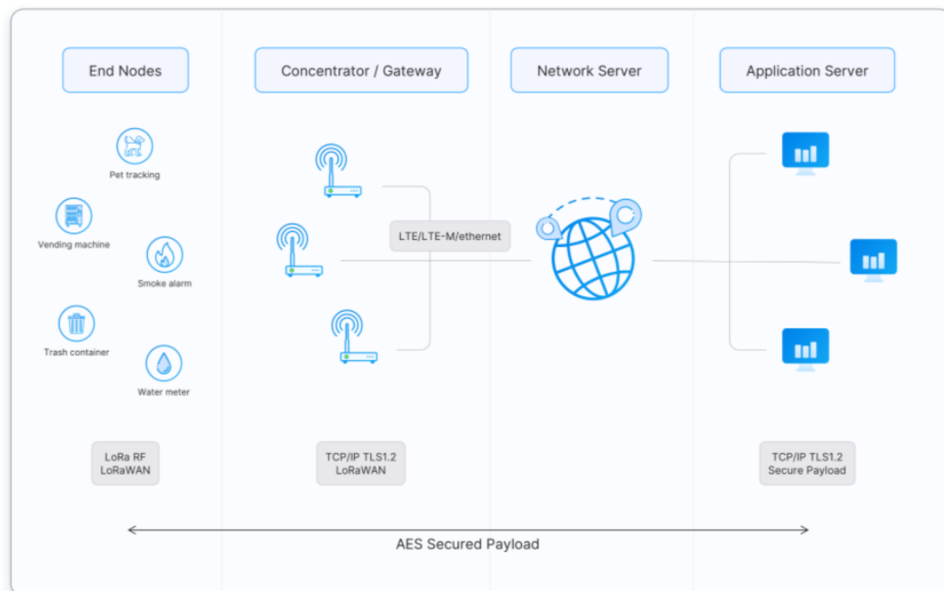


Figura 1.4. Elementi della rete LoRaWAN

Tra di essi si vedono subito gli End-Device, ovvero sensori o attuatori connessi in modalità wireless a una rete LoRaWAN, che, nel progetto preso in esame, sono appunto gli End-Node prima citati. Vi sono poi i Gateway, che, in generale, possono essere più di uno, ma, in questo caso, sono formati da un singolo elemento. Il loro compito è quello di mettere in comunicazione il Network Server con gli End Device, comunicando con questi ultimi attraverso messaggi in Radio Frequenza modulati tramite LoRa. La comunicazione tra Gateway e Network Server può essere realizzata tramite Wi-Fi, Ethernet cablata o connessione cellulare. I Gateway LoRaWAN operano interamente a livello fisico e, in sostanza, non sono altro che inoltri di messaggi radio LoRa.

Il Network Server si occupa di gestire l'intera rete controllando dinamicamente i parametri di rete, garantendo l'autenticità di ogni sensore sulla rete e l'integrità di ogni messaggio. L'Application Server, invece, è responsabile della gestione e dell'interpretazione dei dati che arrivano dai sensori [11].

Per la realizzazione dei diversi server è stato utilizzato Chirpstack, che fornisce componenti open source per le reti LoRaWAN [14]. Insieme formano una soluzione pronta all'uso che include un'interfaccia web intuitiva per la gestione dei dispositivi

e API per l'integrazione. L'architettura modulare consente l'integrazione all'interno di infrastrutture esistenti.Sono forniti i seguenti componenti:

- *ChirpStack Gateway Bridge*, che gestisce la comunicazione con i Gateway LoRaWAN,
- *ChirpStack Network Server*, che è un'implementazione del Network Server LoRaWAN,
- *ChirpStack Application Server*, che è un'implementazione dell'Application Server LoRaWAN.

Le tre componenti offerte da Chirpstack andranno installate nella Raspberry presente nel Gateway principale. Connesso a quest'ultimo vi sarà il Gateway LoraWAN Picocell SX1308 (Figura 1.5), ovvero un ricetrasmettitore multicanale ad alte prestazioni.

Figura 1.5. LoRa Picocell Gateway

Avendo definito come avviene in linea di massima la comunicazione tra i vari End-Node e il Gateway, si può analizzare come avviene il caricamento dei dati relativi alle acquisizioni nel Gateway.

Ogni acquisizione viene gestita attraverso un portale aziendale con l'utilizzo dell'applicazione ALGIZ.STRUCT. Al suo interno è possibile definire diversi progetti, uno per ogni edificio che si vuole monitorare; per ciascuno di essi si possono specificare diversi template, nel quale vengono dichiarati tutti i parametri relativi all'acquisizione. Tra di essi i più importanti sono il codice del progetto e il codice del template per identificare edificio e tipo di acquisizione, frequenza di campionamento, durata del campionamento, funzionamento in Broadcast o Multicast, e anche Cron expression per determinare un'acquisizione schedulata.

Con il termine Multicast nelle reti si indica la distribuzione simultanea di informazioni verso un gruppo di destinatari, cioè la possibilità di trasmettere la medesima informazione a più dispositivi finali, senza dover indirizzare questi ultimi singolarmente e senza avere, quindi, la necessità di duplicare, per ciascuno di essi, l'informazione da diffondere [5]. In questo caso è proprio il Gateway a mandare lo stesso messaggio a tutti gli End-Node, con la lista degli ID dei dispositivi a cui è destinato. Sarà, quindi, ciascun End-Node a verificare se esso appartiene a tale lista e, in caso contrario, scarterà il messaggio ricevuto.

L'invio al Gateway dei dati contenuti all'interno del template avviene attraverso l'utilizzo di un Broker MQTT, che verrà visto nel dettaglio nel Capitolo 2. In linea generale, quello che accade è che uno o più client si sottoscrivono (operazione di Subscribe) ad un topic del Broker, nel quale vengono pubblicati messaggi dai client iscritti (operazione di Publish). In questo modo tutti i client che sono iscritti al topic riceveranno in tempo reale il messaggio che viene pubblicato.

In questo caso sarà appunto il Gateway (client MQTT) che considera i dati ricevuti nel topic e li salva all'interno di un database. Vi è, inoltre, la possibilità di vedere se la ricezione del messaggio è andata a buon fine, attraverso un flag di sincronizzazione mandato dal Gateway.

Infine, per attivare l'acquisizione dei dati accelerometrici, attraverso un'acquisizione ondemand o schedulata, verrà lanciato uno script Python che si occuperà di effettuare inizialmente un check su eventuali template già in corso e stabilire se vi sono dei conflitti, per poi proseguire inviando un messaggio con durata del template, codice del template, frequenza di acquisizione e lista dei dispositivi associati al template a tutti gli End-Node attraverso il protocollo LoRaWAN. Successivamente, verrà inviato il segnale di TRIGGER per avviare il salvataggio dell'acquisizione, mentre ogni 20 secondi verranno inviati i segnali di BEACON per la sincronizzazione, per poi terminare con il segnale di END. Tutti i 3 messaggi contengono anche la lista dei dispositivi a cui sono rivolti per permettere agli End-Node di scartarli o meno.

Il Nodo End-Node è costituito da due board, ovvero una STM32 BL072Z (Figura 1.6) e una Raspberry Pi 3 B+, da due convertitori Analogici-Digitali MCC172, da un accelerometro ADXL354 e da un Modem Multitech 4G. Nella struttura che verrà analizzata, sia essa un edificio piuttosto che un ponte, come già detto, ci saranno più punti di acquisizione, e perciò vi saranno più End-Node.

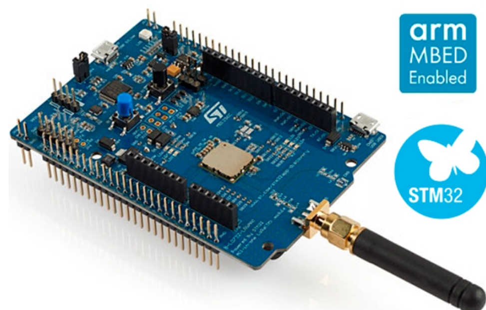


Figura 1.6. STM32 BL072Z

La board STM32 BL072Z viene utilizzata per lo sviluppo di soluzioni basate su tecnologia LoRa; essa comprende un microcontrollore STM32L072CZ e un ricetrasmettitore SX1276. Il ricetrasmettitore è dotato del modem a lungo raggio LoRa, che fornisce comunicazioni a spettro esteso a lungo raggio e un'elevata immunità alle interferenze, riducendo al minimo il consumo di corrente.

ST fornisce già un pacchetto di espansione software chiamato I-CUBE-LRWAN, costituito da una serie di librerie ed esempi di applicazioni per i microcontrollori serie

STM32L0, in grado di supportare trasmissioni in Classe A, B e C per comunicazioni tramite protocollo LoRaWAN. Nello sviluppo del progetto, la struttura del codice fornito da ST non è stata modificata eccezion fatta per delle piccole modifiche volte a soddisfare le esigenze di Multicast e l'invio dei comandi tramite UART alla Raspberry. I messaggi che la board riceve contengono tutti i dati necessari per la configurazione del template che deve eseguire; tali dati sono:

- l'elenco degli identificativi ID;
- codice del progetto;
- la durata del tempo di acquisizione;
- il sample rate con cui effettuare il campionamento delle accelerazioni;
- il codice del template;
- i messaggi di sincronizzazione (TRIGGER, BEACON, END).

La board STM32 invierà poi tramite comunicazione UART alla Rasperry il codice progetto e ID, e i dati del template che sta per essere eseguito, ovvero template code e sample rate. Una volta terminato l'invio dei dati iniziali, la board STM32 invierà il segnale di TRIGGER appena riceverà tale messaggio dal Gateway, con il quale si darà il via al salvataggio dei dati acquisiti. Successivamente si procederà con l'invio dei BEACON ogni 20 secondi per tutto l'arco dell'acquisizione, finché non verrà inviato il segnale di END. Entrambi, come precedentemente descritto, vengono prima inviati dal Gateway alla STM32. Essi vengono utilizzati, una volta raccolti i dati accelerometrici, per sincronizzare i campioni raccolti dai vari End-Node. Infatti, non è detto che tutti gli End-Node inizino l'acquisizione nello stesso istante seppur ricevano il segnale di TRIGGER tutti dallo stesso Gateway. Inoltre, nell'arco dell'acquisizione, si potrebbero verificare delle piccole derive di clock da parte degli End-Node, che fanno cambiare leggermente la frequenza di acquisizione e il numero di campioni totali raccolti.

L'altro componente dell'End-Node è, appunto, la Raspberry Pi. Essa viene utilizzata per l'acquisizione dei dati vera e propria, attraverso due schede ADC MCC172 montate sul lato superiore. Al proprio interno è sviluppato un codice multithread che parallelizza le due funzioni principali che la scheda deve eseguire, ovvero

- rimanere in ascolto sull'UART per ricevere i comandi dalla board STM32 BL072Z;
- acquisire costantemente i dati e avviare il salvataggio una volta che l'acquisizione è finita.

Inizialmente i dati vengono salvati nella Raspberry, all'interno della cartella `home/pi/Measurement` e con il nome che identifica il codice progetto, ID e template. Una volta salvato il file `.csv`, viene richiamato uno script Python per il suo trasferimento dalla cartella prima citata all'opportuna cartella dell'FTP server. Per il caricamento nella giusta cartella, le informazioni necessarie (codice template, ID Raspberry, codice progetto) sono estrapolate dal nome del `.csv` prima creato. Le due acquisizioni generano un file `.csv`, contenente i dati relativi ai 20s precedenti all'arrivo del TRIGGER e l'intera acquisizione di 20 minuti effettuata dopo tale segnale. Il `.csv` è costituito da 5 colonne fondamentali, nelle quali vi sono, rispettivamente, le accelerazioni lungo gli assi x,y,z. Vengono riportate anche due colonne chiamate $time0$ e $time1$, una per ogni scheda MCC172 utilizzata, che contengono i seguenti valori:

- $time0$ e $time1 = beacon_count$ se in quei campioni, acquisiti, rispettivamente, dalla prima o seconda scheda MCC172, è arrivato il segnale di BEACON;
- $time0$ e $time1 = 0$ per tutti gli altri campioni.

Qui, *beacon_count* assume un valore crescente che parte da 0 e, all'arrivo di ciascun BEACON, viene incrementato di 1. Inoltre i segnali x,y,z vengono misurati, rispettivamente, nei canali di acquisizione 0 e 1 della HAT0 (ovvero prima board partendo dalla Raspberry) e nel canale 0 della HAT1 (ovvero seconda board partendo dalla Raspberry).

E' possibile caricare il file csv sul FTP server perché alla Raspberry è connesso un Modem Multitech 4G.

2

Analisi del Gateway

In questo capitolo verrà brevemente descritto lo standard di messaggistica MQTT utilizzato per ricevere nel Gateway i dati relativi all'acquisizione, impostati dall'utente. Verranno, inoltre, presentati alcuni esempi di template che vengono definiti nella piattaforma ALGIZ.STRUCT per configurare le acquisizioni. Infine, verrà svolta una breve analisi dei codici presenti all'interno della Raspberry del Gateway.

2.1 Standard di Messaggistica MQTT

MQTT è l'acronimo di Message Queuing Telemetry Transport e indica un protocollo di trasmissione dati TCP/IP basato su un modello di pubblicazione e sottoscrizione che opera attraverso un apposito Message-Broker. È leggero, aperto, semplice e progettato in modo da essere facile da implementare. Tali caratteristiche lo rendono ideale per l'uso in molte situazioni, come per la comunicazione in contesti di Machine to Machine (M2M) e dell'Internet of Things (IoT). Un altro aspetto importante del protocollo MQTT è che esso è estremamente facile da implementare "lato client". La facilità d'uso è stata una prerogativa chiave nello sviluppo di MQTT, e ciò lo rende perfetto per i dispositivi con delle risorse limitate [12].

Il modello publish/subscribe (noto anche come pub/sub) fornisce un'alternativa alla tradizionale architettura client-server. Nel modello client-sever, un client comunica direttamente con un endpoint. Il modello pub/sub disaccoppia il client che invia un messaggio (il publisher) dal client o dai client che ricevono i messaggi (i subscriber). I publisher e i subscriber non si contattano mai direttamente. In realtà, non sono nemmeno consapevoli dell'esistenza l'uno dell'altro. Il collegamento tra di loro è gestito da un terzo componente, ovvero il broker. Il compito di quest'ultimo è quello di filtrare tutti i messaggi in arrivo e distribuirli correttamente ai subscriber. In Figura 2.1 è raffigurato tale processo.

L'aspetto più importante di pub/sub è il disaccoppiamento dell'editore del messaggio (publisher) dal destinatario (subscriber). Questo disaccoppiamento ha diverse dimensioni:

- *Disaccoppiamento spaziale*: publisher e subscriber non sono tenuti a conoscersi; ad esempio, non vi è nessuno scambio di indirizzo IP e porta.

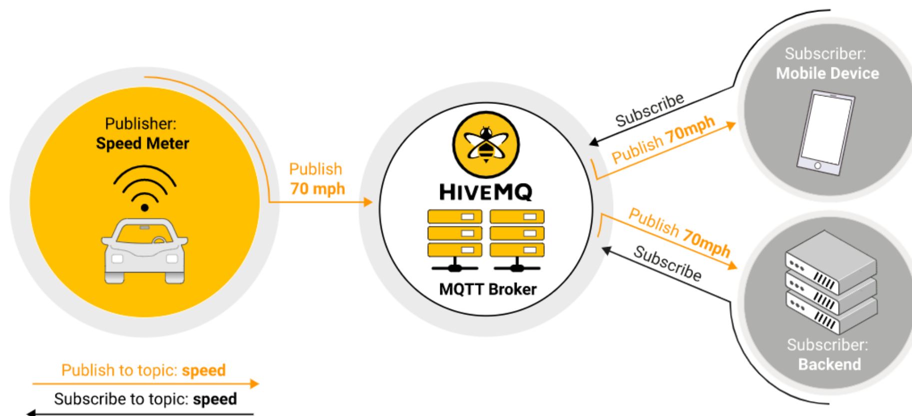


Figura 2.1. Architettura di pubulish/subscribe MQTT

- *Disaccoppiamento temporale*: publisher e subscriber non sono tenuti ad essere eseguiti contemporaneamente.
- *Disaccoppiamento della sincronizzazione*: le operazioni su entrambi i componenti non hanno bisogno di essere interrotte durante la pubblicazione, la ricezione o l'attesa di un messaggio.

Il metodo pub/sub è più scalabile del tradizionale approccio client-server. Questo perché le operazioni sul broker possono essere altamente parallelizzate e i messaggi possono essere elaborati in modo guidato dagli eventi.

Il broker si occupa di svolgere un'attività di filtraggio dei messaggi che riceve, determinando per ciascuno di essi, a quale subscriber deve essere inviato. In questo modo ogni client sottoscritto al broker riceve solo i messaggi di suo interesse. Tale filtraggio può avvenire con vari metodi; nel caso MQTT vi è un filtraggio in base all'oggetto. Questo filtro si basa sull'oggetto o sul topic che fa parte di ogni messaggio. Il client in ricezione si iscrive al broker nei topic di interesse. Da quel momento in poi, il broker assicura che il client ricevente ottenga tutti i messaggi pubblicati nei topic sottoscritti. In generale, un topic è una stringa con una struttura gerarchica che permette di effettuare il filtraggio.

A seconda di ciò che si desidera ottenere, MQTT incarna tutti gli aspetti di pub/sub che sono stati menzionati.

2.1.1 Client e broker

Sia i publisher che i subscriber sono client MQTT. Le due etichette si riferiscono al fatto che il client stia attualmente pubblicando messaggi o si sia iscritto per ricevere messaggi (la funzionalità di pubblicazione e sottoscrizione può essere implementata anche nello stesso client MQTT). Un client MQTT può essere un qualsiasi dispositivo, da un microcontrollore fino a un server completo, che esegue una libreria MQTT e si connette a un broker MQTT su una rete. Ad esempio, il client MQTT può essere un dispositivo molto piccolo con risorse limitate che si connette su una

rete wireless e dispone di una libreria minima; esso, però, può anche essere un tipico computer che esegue un client MQTT grafico a scopo di test. Fondamentalmente, qualsiasi dispositivo che parla MQTT su uno stack TCP/IP può essere chiamato client MQTT.

Le librerie client MQTT sono disponibili per un'ampia varietà di linguaggi di programmazione, e in questo progetto è stata usata la libreria `paho` in Python.

La controparte del client MQTT è il broker MQTT. Il broker è al centro di qualsiasi protocollo di pubblicazione/sottoscrizione. A seconda dell'implementazione, un broker può gestire fino a milioni di client MQTT connessi contemporaneamente. Il broker è responsabile della ricezione di tutti i messaggi, del loro filtraggio, della determinazione di chi è iscritto al topic di ciascun messaggio e dell'invio del messaggio a questi client iscritti. Il broker conserva anche i dati di sessione di tutti i client che hanno sessioni persistenti, che comprendono anche le sottoscrizioni e i messaggi persi. Un'altra responsabilità del broker è l'autenticazione e l'autorizzazione dei client. Di solito, il broker è estensibile, il che facilita le autenticazioni, le autorizzazioni e le integrazioni personalizzate nei sistemi di backend. L'integrazione è particolarmente importante perché il broker è spesso il componente direttamente esposto su Internet che gestisce molti client e deve passare i messaggi ai sistemi di analisi ed elaborazione a valle. In breve, il broker è l'hub centrale attraverso il quale deve passare ogni messaggio. Pertanto, è importante che esso sia altamente scalabile, integrabile in sistemi di backend, facile da monitorare e, ovviamente, resistente ai guasti.

2.1.2 Messaggi fondamentali

La connessione MQTT avviene sempre tra un client e il broker. Per avviare una connessione, il client invia un messaggio *CONNECT* al broker. Quest'ultimo risponde con un messaggio *CONNACK* e un codice che identifica lo stato della connessione. Una volta stabilita la connessione, il broker la mantiene aperta finché il client non invia un comando di disconnessione o fino a quando la connessione non si interrompe.

Un client MQTT può pubblicare messaggi non appena si connette a un broker. Ogni messaggio deve contenere un topic che il broker può utilizzare per inoltrare il messaggio stesso ai client interessati. Esso è una semplice stringa strutturata gerarchicamente con il carattere `"/"` come delimitatore tra le varie componenti che la formano. Tipicamente, ogni messaggio ha un payload, che contiene il contenuto effettivo del messaggio da trasmettere. Il protocollo è indipendente dal formato dei dati e solo il caso d'uso del client determina come è strutturato il payload. Il client publisher decide se inviare dati binari, dati di testo o persino XML o JSON. Quando un client invia un messaggio a un broker MQTT per la pubblicazione, il broker legge il messaggio e lo elabora. L'elaborazione da parte del broker include la determinazione dei client che sono sottoscritti al topic e l'invio del messaggio. Il client che inizialmente pubblica il messaggio si preoccupa solo di consegnare il messaggio *PUBLISH* al broker. Una volta che quest'ultimo riceve tale messaggio, è sua responsabilità consegnarlo a tutti i sottoscrittori. Il client di pubblicazione non riceve alcun feedback sul fatto che qualcuno sia interessato al messaggio pubblicato, o sul numero di client che hanno ricevuto il messaggio dal broker. Il formato tipico del messaggio *PUBLISH* è mostrato in Figura 2.2.

MQTT-Packet:

PUBLISH

contains:	Example
packetId (always 0 for qos 0)	4314
topicName	"topic/1"
qos	1
retainFlag	false
payload	"temperature:32.5"
dupFlag	false

Figura 2.2. Attributi di un messaggio *PUBLISH*

Pubblicare un messaggio non ha senso se nessuno lo riceve mai, ovvero se non ci sono client da sottoscrivere ai topic dei messaggi. Per ricevere messaggi sul topic di interesse, il client invia un messaggio *SUBSCRIBE* al broker MQTT. Questo messaggio di iscrizione è molto semplice; esso contiene un identificatore di pacchetto univoco e un elenco di sottoscrizioni. Per confermare ogni sottoscrizione, il broker invia un messaggio di riconoscimento *SUBACK* al client. Tale messaggio contiene l'identificatore del pacchetto del messaggio di sottoscrizione originale (per identificare chiaramente il messaggio) e un elenco di codici di ritorno che identificano se la sottoscrizione è avvenuta con successo o meno. Dopo che un client invia con successo il messaggio *SUBSCRIBE* e riceve il messaggio *SUBACK*, esso riceve tutti i messaggi pubblicati nei topic contenuti nelle sottoscrizioni del messaggio *SUBSCRIBE*.

La controparte del messaggio *SUBSCRIBE* è il messaggio *UNSUBSCRIBE*. Questo elimina le sottoscrizioni esistenti di un client sul broker. Il messaggio *UNSUBSCRIBE* è simile al messaggio *SUBSCRIBE*, essendo formato da un identificatore di pacchetto e un elenco di topic. Per confermare la cancellazione, il broker invia un messaggio di conferma *UNSUBACK* al client. Tale messaggio contiene solo l'identificatore del pacchetto del messaggio *UNSUBSCRIBE* originale (per identificare chiaramente il messaggio).

2.2 Sviluppo del Gateway

In questo progetto, il broker MQTT è implementato all'interno della piattaforma aziendale ALGIZ.STRUCT, attraverso l'utilizzo della libreria `Mosquitto`. La stessa piattaforma ALGIZ.STRUCT svolge anche la funzione di client all'interno del protocollo MQTT. Infatti, essa effettua pubblicazioni sui topic di riferimento, con le informazioni riguardanti i template e i device configurati. Infatti, per ciascun progetto, quindi per ciascuna struttura monitorata, nella piattaforma vengono prima inseriti tutti i dispositivi che verranno utilizzati nell'acquisizione, partendo dal Gateway e definendo poi gli End-Node, con i relativi ID. Vengono, poi, configurati i template relativi alle possibili acquisizioni che si effettueranno. Per ciascun progetto potranno essere definiti più template, come mostrato in Figura 2.3.

Per ogni template le informazioni che vengono inserite riguardano ogni aspetto della configurazione di un'acquisizione. Esse comprendono:

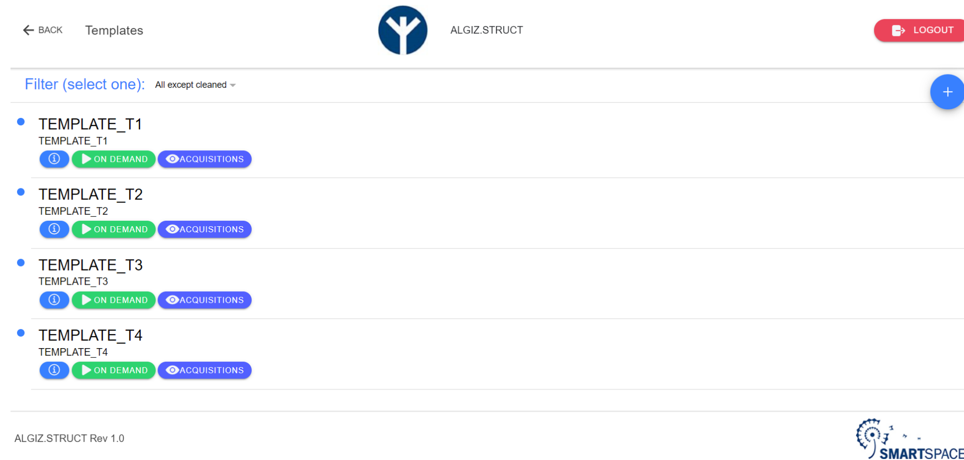


Figura 2.3. Lista dei possibili template per un progetto

- Il codice progetto.
- La possibilità di effettuare un'acquisizione in Broadcast o Multicast. Nel primo caso vengono utilizzati tutti gli End-Node, mentre nel secondo caso, essi possono essere scelti o meno. In quest'ultimo caso è necessario specificare la lista degli End-Node che devono essere utilizzati. Questi devono essere stati prima inseriti nella lista dei device.
- La frequenza di campionamento in Hertz.
- La durata del campionamento in secondi.
- La Cron Expression, che indica quando deve essere eseguita ciascuna acquisizione schedulata. Ad esempio, essa potrebbe essere eseguita tutti i giorni del mese a mezzanotte.
- La data di inizio e di fine della schedulazione imposta nella Cron Expression.
- La possibilità di eseguire una schedulazione on-demand, ovvero l'acquisizione delle accelerazioni viene eseguita appena viene avviato il template dalla piattaforma AGIL.STRUCT.
- Il flag di sincronizzazione con il Gateway.

Un esempio di Template viene visualizzato in Figura 2.4.

L'altro client MQTT presente è, appunto, il Gateway, che, sottoscritto agli stessi topic della piattaforma, riceve le informazioni riguardanti i device (sia Gateway che End-Node) e i template (con le configurazioni delle acquisizioni) da utilizzare per pianificare le acquisizioni accelerometriche. Come già detto, il Gateway è costituito da un'unica Raspberry Pi 3B+. L'intera applicazione si sviluppa sui seguenti quattro file:

- `main.py`;
- `Template_Model.py`;
- `mqtt.py`;
- `send_synch.py`.

Figura 2.4. Possibile configurazione di un template

Inoltre vi è la presenza di due file di testo, ovvero `logfile.txt` e `output.txt`, nei quali viene direzionato l'intero output del file `send_synch.py`, piuttosto che utilizzare l'output di default su schermo.

Il primo file da descrivere è, ovviamente, `main.py`, che contiene il loop principale del programma. Esso si occupa della creazione del database nel quale verranno salvati i dati ricevuti tramite MQTT (*init_database()*), e della sottoscrizione ai giusti topic del broker MQTT al primo avvio dell'app (*mqtt.connect_subscribe()*); altrimenti, se il database è già presente, carica da esso tutte le informazioni ad ogni riavvio della scheda (*mqtt.initialize()*). Il codice presente in tale file è visibile nel Listato 2.1.

```
import sqlite3
from peewee import *
from Template_Model import *
import os.path
from mqtt import mqtt
import time
```

```
if not os.path.exists('./templates.db'):
    init_database()
    mqtt=mqtt()
    mqtt.connect_subscribe()
else:
    mqtt=mqtt()
    mqtt.initialize()

if __name__ == '__main__':
    while True:
        print("\n . \n")
        time.sleep(10)
```

Listato 2.1. Codice presente nel file `main.py` del Gateway

Il secondo file, ovvero `Template_Model.py`, contiene la definizione delle tre tabelle che costituiscono il database, indicando tutti i campi e i tipi di dato presenti. Inoltre, viene definita la funzione per l'inizializzazione dell'SQLite database, che non fa altro che creare le tre tabelle all'interno di quest'ultimo. Le tre tabelle vengono denominate:

- *Template*, che viene utilizzata per raccogliere tutti i dati relativi al template da eseguire;
- *Device*, che viene utilizzata per memorizzare tutti i dispositivi presenti; questi possono essere o dei Gateway o degli End-Node;
- *TemplateDevice*, che viene utilizzata per determinare a quali device è rivolto ciascun template.

Il codice presente in tale file è visibile nel Listato 2.2.

```
from peewee import *
import sqlite3

db = SqliteDatabase('/home/pi/templates.db')

class Template(Model):

        code = CharField()
        uuid = CharField()
        descr = CharField()
        proj_uuid = CharField()
        proj_code = CharField()
        broadcast = BooleanField()
        samples = IntegerField()
        period = CharField()
        interval = IntegerField()
        start_ts = CharField()
        end_ts = CharField()
        topic_ack = CharField()
        topic_ev = CharField()
        enabled = BooleanField()
        on_demand = BooleanField()
        last_ts_on_demand = CharField(null = True)
        devices = CharField()
        active = BooleanField()
        socket_id = CharField()
        last_ts_on_sched = CharField()

        class Meta:
            database = db

class Device(Model):

        gateway = BooleanField()
        code = CharField()
        uuid = CharField()
        descr = CharField()
        proj_uuid = CharField()
        proj_code = CharField()
        eui = CharField()
        topic_ack = CharField()
        socket_id = CharField()
        associated_ll_id = CharField()
        class Meta:
            database = db
```

```
class TemplateDevice(Model):

        template = ForeignKeyField(Template, on_delete='CASCADE')
        device = ForeignKeyField(Device, on_delete='CASCADE')

        class Meta:
            database = db

def init_database():
    db.connect()
    db.create_tables([Template, Device, TemplateDevice], safe=True)
    db.close()
```

Listato 2.2. Codice presente nel file `Template_Model.py` del Gateway

Il prossimo file da analizzare è `mqtt.py`, che contiene la definizione della classe *mqtt*, la quale costituisce il core dell'applicazione. Infatti, di fondamentale importanza sono i suoi metodi di seguito descritti:

- *initialize()*: utilizzato per il caricamento delle informazioni dal database e, in particolare, per caricare tutte quelle presenti nelle tabelle *Template* e *Device*. Inoltre, viene effettuato il reload della crontab al riavvio del sistema, rimuovendo i vecchi *job* e inserendo i nuovi. I *job* presenti sono le acquisizioni schedulate.
- *on_disconnect()*: viene richiamata alla disconnessione dall'MQTT per comunicare tale evento all'utente.
- *on_connect()*: è la funzione invocata alla connessione all'MQTT. Essa viene utilizzata per la sottoscrizione ai giusti topic, che comprende, anche, la gestione del caso di una riconnessione a seguito di una disconnessione.
- *on_message()*: è la funzione principale della classe, ovvero, quella che viene richiamata ogni volta che viene ricevuto un messaggio sull'MQTT. Essa permette di gestire:
  - Gli eventi di creazione, update e delete di un Gataway sulla piattaforma ALGIZ.STRUCT. Tale operazione viene comunicata sull'MQTT e vengono così create, modificate o cancellate le opportune istanze di *Device*.
  - Gli eventi di creazione, update e delete di un End-Node sulla piattaforma ALGIZ.STRUCT. Tale operazione viene comunicata sull'MQTT e vengono così create, modificate o cancellate le opportune istanze di *Device*.
  - Gli eventi di creazione, update e delete di un template sulla piattaforma ALGIZ.STRUCT. Tale operazione viene comunicata sull'MQTT e vengono così create, modifcate o cancellate le opportune istanze di *Template* e di *TemplateDevice*. Inoltre, vengono aggiunti alla crontab i *job* relativi al template aggiunto o modificato, che permetteranno di mandare in esecuzione lo script `send_synch.py` negli istanti programmati, facendo così avviare l'acquisizione.
  - La schedulazione di un'acquisizione on-demand, con l'avvio diretto dello script `send_synch.py` per iniziare l'acquisizione.
- *on_lorawan_msg()*: è la funzione che gestisce gli acknowledgement e i messaggi uplink (ovvero i messaggi dagli End-Node al Gateway) degli End-Node ricevuti tramite protocollo LoRaWAN. Essi saranno segnalati attraverso l'MQTT alla piattaforma ALGIZ.STRUCT.
- *json2dict()*: è la funzione utilizzata per convertire i messaggi in arrivo sui topic in dizionari Python.

- *connect_subscribe()*: è la funzione richiamata nel `main.py` per la connessione e sottoscrizione ai giusti topic del broker MQTT al primo avvio dell'app.
- *topics_subscribe()* e *topics_unsubscribe()*: sono le funzioni usate per sottoscriversi ad un topic e per cancellare la sottoscrizione.
- *send_ID_proj_code()*: è la funzione utilizzata per mandare un messaggio downlink (ovvero dal Gateway agli End-Node) tramite protocollo LoRaWAN, all'atto di creazione o update di un nuovo End-Node, con codice del progetto, ID del device e low level ID. Tale messaggio sarà inviato sulla porta 5.
- *send_dev_delete()*: utilizzata per mandare un messaggio di delete all'End-Node interessato, che verrà disattivato; ciò è effettuato tramite protocollo LoRaWAN e utilizzando la porta 6.
- *check_device_in_chirpstack()*: utilizzata per controllare se un determinato EUI (Extended Unique Identifier) è registrato o meno su Chirpstack.
- *get_gateway_eui()*: è la funzione che ritorna l'EUI del Gateway registrato in Chirpstack.

Il codice presente all'interno del file `mqtt.py` non verrà mostrato, poiché esso è molto esteso e, per la comprensione del progetto, è sufficiente l'analisi dei metodi principali appena svolta.

L'ultimo file presente è `send_synch.py`. Esso viene richiamato attraverso la crontab, ogni volta che viene raggiunto lo specifico timestamp di un'acquisizione schedulata. In alternativa, esso viene attivato direttamente all'interno della funzione *on_message()* nel caso di un'acquisizione on-demand. Lo script si occuperà di effettuare inizialmente un check su eventuali template già in corso e stabilire se vi sono dei conflitti, come, ad esempio, template concorrenti che condividono dispositivi in comune per l'acquisizione. Se questa verifica è superata, viene inviato sull'MQTT un messaggio di conferma.

Si prosegue, poi, inviando un messaggio con durata del template, codice del template, frequenza di acquisizione e lista dei dispositivi associati al template a tutti gli End-Node attraverso il protocollo LoRaWAN, nella porta 10. Successivamente, viene inviato il segnale di TRIGGER per avviare il salvataggio dell'acquisizione, mentre ogni 20 secondi vengono inviati i segnali di BEACON per la sincronizzazione, per poi terminare con il segnale di END. Tutti i 3 messaggi contengono anche la lista dei dispositivi a cui sono rivolti per permettere un filtraggio dei messaggi da parte dell'End-Node, che decide, così, se scartare o meno ciascuno di essi.

Il codice presente all'interno dello script `send_synch.py` è mostrato nel Listato 2.3.

```
from datetime import date, datetime
import requests
import json
import time
import sys
from peewee import *
from Template_Model import *
import base64
import paho.mqtt.publish as publish
from paho.mqtt.client import Client
from datetime import date, datetime
from dateutil import tz
import ast
import csv

def client_connect():
    client = Client(transport='websockets')
    client.ws_set_options(path="/mqtt", headers=None)
    client.username_pw_set("smartspace", "sm4rtsp4c3")
    client.tls_set()
```

```
    client.connect("struct.smartspace.it", 443, 20)
    return client

def common_devices(list1, list2):
    result = False
    for x in list1:
        for y in list2:
            if x == y:
                result = True
                return result
    return result

from_zone = tz.gettz('UTC')
to_zone = tz.gettz('Italy/Rome')
counter = 1
separator = ','

url = 'http://localhost:8080/api/multicast-groups/d423d703-a35a-4118-bc76-ed66401b5f8c/queue'

MyHeaders = {"Grpc-Metadata-Authorization": "Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
       eyJhcGlfa2V5X2lkIjoiZWYyODBhNzAtMDNmMi00ZGZkLWFmZDctMjUwMTVlYmQxNzZmIiwiYXVkIjoiYXMiLCJpc3MiOi
       JhcyIsIm5iZiI6MTYxNzAyMjI0Mywic3ViIjoiYXBpX2tleSJ9.4WwGd4UMMZGNWk4ECxNGUgpex0y4X0zzarmNq8Z8_gw",
                      "Content-Type": "application/json",
                      "Accept":"application/json"}

ack = {
                     "op":None,
                     "success" : 0,
                     "descr": "String",
                     "on_sched":True,
                     "ts" : 0,
                     "socket_id": "String"
                                       }
ev = {
                     "op":None,
                     "on_demand":None,
                     "success" : 0,
                     "descr": "String",
                     "ts": "String"
                                       }
ev_success = {
                     "op":None,
                     "on_demand":None,
                     "success" : 0,
                     "descr": "String"
                                       }

def send_request(url, data, headers):
   requests.post(url, data=json.dumps(data), headers=headers)

#####FOR TEST######
template = str(sys.argv[1])

try:
    f = open("/home/pi/logfile.txt", "a")
    db.connect()
    template_to_execute = Template.select().where(Template.code==template).get()
    topic_ack = template_to_execute.topic_ack
    topic_ev = template_to_execute.topic_ev
    #costruisco le liste con i dispositivi del template che vuole partire
    if (template_to_execute.broadcast == False):
        current_template_devices = ast.literal_eval(template_to_execute.devices)
        ids = []
        for item in current_template_devices:
            try:
                device = Device.select().where(Device.code==item).get()
                ids.append(device.associated_ll_id)
            except:
                print("Device %s deleted" %item)
                pass
        devices = separator.join(ids)
    #if broadcast, all devices are copied to the variable ids
    else:
        ids=[]
        for device in Device.select():
            if (device.gateway==False):
                ids.append(device.associated_ll_id)
        devices = separator.join(ids)
    print("Lista devices template:")
    print(ids)
    #######
    utc_start_time = datetime.strptime(template_to_execute.start_ts, '%Y-%m-%dT%H:%M:%SZ')
    utc_end_time=datetime.strptime(template_to_execute.end_ts, '%Y-%m-%dT%H:%M:%SZ')
    now = datetime.utcnow()
    if now < utc_end_time and now > utc_start_time:
        #####NEW
        conflict = False
        cnt = 0
        for template in Template.select().where(Template.active == True):
            if (template_to_execute.broadcast == True or template.broadcast == True):
                conflict = True
                if template_to_execute.on_demand == True:
                    if template.on_demand == False:
                        ack['op'] = 3
```

```
                ack['success'] = False
                ack['on_sched'] = True
                ack['ts'] = template.last_ts_on_sched
                ack['descr']= "Discovered conflicting acquisitions"
                ack['socket_id'] = template.socket_id
            else:
                ack['op'] = 3
                ack['success'] = False
                ack['on_sched'] = False
                ack['ts'] = template.last_ts_on_demand
                ack['descr']= "Discovered conflicting acquisitions"
                ack['socket_id'] = template.socket_id

    else:
        already_active_devices = ast.literal_eval(template.devices)
        ids_already_active = []
        for item in already_active_devices:
            try:
                device = Device.select().where(Device.code==item).get()
                ids_already_active.append(device.associated_ll_id)
            except:
                printf("Device %s deleted" %item)
                pass
        conflict = common_devices(ids_already_active, ids)
        if conflict == True:
            if template.on_demand == False:
                ack['op'] = 3
                ack['success'] = False
                ack['on_sched'] = True
                ack['ts'] = template.last_ts_on_sched
                ack['descr']= "Discovered conflicting acquisitions"
                ack['socket_id'] = template.socket_id
            else:
                ack['op'] = 3
                ack['success'] = False
                ack['on_sched'] = False
                ack['ts'] = template.last_ts_on_demand
                ack['descr']= "Discovered conflicting acquisitions"
                ack['socket_id'] = template.socket_id
            cnt = cnt + 1

if (cnt>0):
    conflict = True
print("Discovered conflicting templates:")
print(conflict)

if (template_to_execute.active == False and conflict == False):
    #send on mqtt success start message, on ack if on demand acq
    if (template_to_execute.on_demand==True):
        on_demand_time = datetime.utcnow()
        template_to_execute.last_ts_on_demand = on_demand_time.strftime("%Y-%m-%dT%H:%M:%S")
        template_to_execute.save()
        client = client_connect()
        ack['op']=3
        ack['success']=True
        ack['descr']='Acquisition on demand started'
        ack['on_sched']= False
        ack['ts'] = template_to_execute.last_ts_on_demand
        ack['socket_id'] = template_to_execute.socket_id
        client.publish(topic_ack, json.dumps(ack))
        client.disconnect()
    #send on mqtt success start message, on ev if scheduled acq
    else:
        scheduled_time = datetime.utcnow()
        template_to_execute.last_ts_on_sched = scheduled_time.strftime("%Y-%m-%dT%H:%M:%S")
        template_to_execute.save()
        client = client_connect()
        ev['op']=0
        ev['on_demand'] = False
        ev['success'] = True
        ev['ts'] = template_to_execute.last_ts_on_sched
        ev['descr']='Acquisition scheduled started'
        client.publish(topic_ev, json.dumps(ev))
        client.disconnect()
    ####### if template not expired not already running and not conflicts with other, update db "active"
         field and publish on mqtt#####
    print("Running acquisition!\n")
    template_to_execute.active = True
    template_to_execute.save()
    ########TEMPLATE CODE, SAMPLE RATE################### The proj code is sent when the device is craated
    synch_csv = open(template_to_execute.code+'.csv', 'w', newline='')
    writer = csv.writer(synch_csv)
    writer.writerow(["Counter", "Timestamp"])
    proj_code = template_to_execute.proj_code
    template_config = str(template_to_execute.interval)+'/'+str(template_to_execute.code)+'/'+str(
          template_to_execute.samples)+'/'+devices+'@'
    #THE DEVICES ARE ALWAYS SENT ALSO IN BROADCAST FOR THOSE ELIMINATED
    template_config_bytes = bytes(template_config, 'utf-8')
    template_config_2b64message = base64.b64encode(template_config_bytes)
    payload = template_config_2b64message.decode(encoding='UTF-8')
    template_config_req = {
                "multicastQueueItem" : {"confirmed": False,
                                      "data":payload,
                                      "fCnt":11,
```

```
                                        "fPort":10,
                                        "multicastGroupID":"d423d703-a35a-4118-bc76-ed66401b5f8c"}
                    }

        send_request(url, template_config_req, MyHeaders)
        #########PREPARE TRIGGER, BEACON AND END JSON######################################
        trigger = devices+'@'+'1'
        beacon = devices+'@'+'2'
        end = devices+'@'+'3'
        trigger_bytes = bytes(trigger, 'utf-8')
        beacon_bytes = bytes(beacon, 'utf-8')
        end_bytes = bytes(end, 'utf-8')
        trigger_2b64message = base64.b64encode(trigger_bytes)
        beacon_2b64message = base64.b64encode(beacon_bytes)
        end_2b64message = base64.b64encode(end_bytes)
        trigger_payload = trigger_2b64message.decode(encoding='UTF-8')
        beacon_payload = beacon_2b64message.decode(encoding='UTF-8')
        end_payload = end_2b64message.decode(encoding='UTF-8')
        trigger_json = {
                    "multicastQueueItem" : {"confirmed": False,
                                        "data":trigger_payload,
                                        "fCnt":11,
                                        "fPort":4,
                                        "multicastGroupID":"d423d703-a35a-4118-bc76-ed66401b5f8c"}
                    }
        beacon_json = {
                    "multicastQueueItem" : {"confirmed": False,
                                        "data":beacon_payload,
                                        "fCnt":11,
                                        "fPort":4,
                                        "multicastGroupID":"d423d703-a35a-4118-bc76-ed66401b5f8c"}
                    }
        end_json = {
                    "multicastQueueItem" : {"confirmed": False,
                                        "data":end_payload,
                                        "fCnt":11,
                                        "fPort":4,
                                        "multicastGroupID":"d423d703-a35a-4118-bc76-ed66401b5f8c"}
                    }
        print("Send samples")
        time.sleep(5)
        #########SEND TRIGGER AND BEACONS#########################################################
        elapsed = 0
        start = time.time()
        print("Send trigger")
        send_request(url, trigger_json, MyHeaders)
        writer.writerow([counter, datetime.utcnow().strftime("%Y-%m-%dT%H:%M:%S")])
        while (elapsed < template_to_execute.interval):
            time.sleep(15)
            send_request(url, beacon_json, MyHeaders)
            counter += 1
            writer.writerow([counter, datetime.utcnow().strftime("%Y-%m-%dT%H:%M:%S")])
            print("Send beacon")
            elapsed = (time.time()-start)
        print("Send end")
        time.sleep(1.5)
        send_request(url, end_json, MyHeaders)

        if (template_to_execute.on_demand == False):
            ev_success['op'] = 1
            ev_success['on_demand'] = False
            ev_success['descr'] = "Acquisition scheduled completed"
            ev_success['success']=True
            topic = template_to_execute.topic_ev
            client = client_connect()
            client.publish(topic, json.dumps(ev_success))
            client.disconnect()
            debug = str(datetime.now())+'\n'+template+":\nTemplate periodic acquisition executed\n"
        else:
            ev_success['op'] = 1
            ev_success['on_demand'] = True
            ev_success['descr'] = "Acquisition on demand completed"
            ev_success['success']=True
            topic = template_to_execute.topic_ev
            client = client_connect()
            client.publish(topic, json.dumps(ev_success))
            client.disconnect()
            debug = str(datetime.now())+'\n'+template+":\nTemplate on demand acquisition executed\n"
        template_to_execute.active = False
        template_to_execute.on_demand = False
        template_to_execute.save()
        synch_csv.close()
        ##############write log file#################################
        f.write(debug)
        f.close()
    else:
        if (conflict == True):
            if template_to_execute.on_demand == True:
                client = client_connect()
                client.publish(topic_ack, json.dumps(ack))
                client.disconnect()
            f.write(str(datetime.now())+":\n"+"Conflicting templates already running!\n")
        else:
            if (template_to_execute.on_demand == False):
```

```
                f.write(str(datetime.now())+":\n"+str(template_to_execute.code)+" periodic acquisition "+"
                        already running!\n")
            else:
                f.write(str(datetime.now())+":\n"+str(template_to_execute.code)+" on demand acquisition "+"
                        already running!\n")
            f.close()
    else:
        f.write("\n"+str(template_to_execute.code)+" expired!\n")
        f.close()


except Exception as e:
    f.write(str(datetime.now())+":"+"\nError:")
    f.write(str(e)+"\n")
    f.close()

db.close()
```

Listato 2.3. Codice presente nel file `send_synch.py` del Gateway

2.3 Configurazione in servizi e librerie da installare nel Gateway

L'applicazione del Gateway è stata configurata in servizi; in particolare, essi sono:

- Il *gateway*, che avvia l'applicazione in Python appena descritta; la sua configurazione è mostrata nel Listato 2.4.

```
sudo nano /etc/systemd/system/gateway.service

[Unit]
Description=Gateway
After=network.target

[Service]
ExecStart=/usr/bin/python3 -u main.py
WorkingDirectory=/home/pi
StandardOutput=journal+console
StandardError=inherit
Restart=always
User=root
Group=root

[Install]
WantedBy=multi-user.target
```

Listato 2.4. Configurazione del servizio *gateway*

- Il *packet_forwarder*, che avvia il packet forwarder responsabile della gestione del Gateway LoraWAN Picocell SX1308; quest'ultimo consente la comunicazione con gli End-Node; la sua configurazione è mostrata nel Listato 2.5.

```
sudo nano /etc/systemd/system/packet_forwarder.service

[Unit]
Description=My service

[Service]
ExecStart=/home/pi/lora-net/picoGW_packet_forwarder/lora_pkt_fwd/lora_pkt_fwd
WorkingDirectory=/home/pi/lora-net/picoGW_packet_forwarder/lora_pkt_fwd
Restart=on-failure
RestartSec=5
User=pi

[Install]
WantedBy=multi-user.target
```

Listato 2.5. Configurazione del servizio *packet_forwarder*

- Il *wvdial*, utilizzato per la gestione della connessione alla rete attraverso l'utilizzo del Modem Multitech 4G; la sua configurazione è mostrata nel Listato 2.6.

```
sudo nano /etc/systemd/system/wvdial.service

[Unit]
Description=Modem wvdial service

[Service]
ExecStart=/usr/bin/wvdial
ExecReload=/bin/kill -HUP $MAINPID
KillMode=process
Restart=always
RestartSec=30

[Install]
WantedBy=multi-user.target
Alias=wvdial.service
```

Listato 2.6. Configurazione del servizio *wvdial*

- Infine, vi sono i servizi creati con l'installazione dell'infrastruttura Chirpstack; questi verranno visti nel Capitolo 3.

Per avviare ciascun servizio, dopo aver incollato la sua configurazione nel file corrispondente, è necessario eseguire i comandi presenti nel Listato 2.7.

```
daemon-reload
sudo systemctl enable
#con nome servizio èidentificato il nome di ciascun servizio
nomeservizio.service
sudo systemctl start nomerservizio.service
```

Listato 2.7. Avvio dei servizi

Infine, per permettere l'avvio dell'applicazione, bisogna prima installare le librerie `sqlite3`, `peewe`, `paho-mqtt` e `crontab`, utilizzando il comando *pip3 install*.

3

Rete di comunicazione LoRaWAN

In questo capitolo verranno analizzati brevemente il protocollo di comunicazione LoRaWAN e la modulazione LoRa. Verranno, inoltre, descritti gli elementi principali dell'architettura LoRaWAN, nonché, le differenze tra le classi di comunicazione A, B e C. Si spiegherà, anche, perché è stata scelta proprio la Classe C. Infine, verrà mostrata l'intera configurazione da effettuare in Chirpstack per permettere la comunicazione tra il Gateway e gli End-Node, utilizzando anche la modalità Multicast.

3.1 Protocollo di comunicazione LoRAWAN

3.1.1 LoRa e LoRaWAN

LoRa è una tecnica di modulazione wireless derivata dalla tecnologia Chirp Spread Spectrum (CSS) [11].

LoRa è ideale per le applicazioni che trasmettono piccoli blocchi di dati con basse velocità di trasmissione. I dati possono essere trasmessi a una distanza maggiore rispetto a tecnologie come WiFi, Bluetooth o ZigBee. Queste caratteristiche rendono LoRa particolarmente adatto per sensori e attuatori che operano in modalità a basso consumo. LoRa può essere utilizzato sulle bande sub-gigahertz senza licenza, ad esempio 915 MHz, 868 MHz e 433 MHz. Può anche funzionare a 2.4 GHz per ottenere velocità di trasmissione dati più elevate rispetto alle bande sub-gigahertz, a scapito della portata [15].

LoRa è un'implementazione di livello puramente fisico (PHY), o "bit", come definito dal modello di rete a sette livelli OSI.

LoRaWAN è un protocollo di livello Media Access Control (MAC) basato sulla modulazione LoRa. È un livello software che definisce come i dispositivi utilizzano l'hardware LoRa, ad esempio quando trasmettono, e il corrispettivo formato dei messaggi. Il protocollo LoRaWAN è sviluppato e mantenuto dalla LoRa Alliance. Esso è adatto per la trasmissione di payload di piccole dimensioni (come i dati dei sensori) su lunghe distanze. La modulazione LoRa fornisce un raggio di comunicazione significativamente maggiore, con larghezze di banda ridotte, rispetto ad altre tecnologie di trasmissione di dati wireless concorrenti. La Figura 3.1 mostra alcune

tecnologie che possono essere utilizzate per la trasmissione di dati wireless e i loro intervalli di trasmissione previsti rispetto alla larghezza di banda.

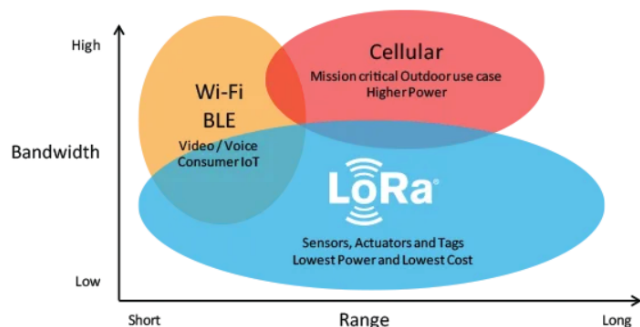


Figura 3.1. Tecnologie utilizzate per la trasmissione di dati wireless

Le principali caratteristiche del protocollo LoRaWAN sono di seguito elencate [15]:

- *Si ha una bassissima potenza*; infatti gli End-Node LoRaWAN sono ottimizzati per funzionare in modalità a basso consumo e possono durare fino a 10 anni con una singola batteria.
- *Lavora su lungo raggio*; infatti, i Gateway LoRaWAN possono trasmettere e ricevere segnali su una distanza di oltre 15 chilometri nelle aree rurali e fino a 5 chilometri nelle aree urbane dense.
- *Può fornire una copertura interna profonda* e coprire facilmente edifici a più piani.
- Per implementare una rete LoRaWAN *non è necessario pagare costose tariffe* per la licenza dello spettro di frequenze.
- *Una rete LoRaWAN può determinare la posizione degli End-Node* utilizzando la triangolazione senza la necessità del GPS. Un End-Node LoRa può essere localizzato se almeno tre Gateway ricevono il suo segnale.
- *Si ha un'alta capacità*, poiché i Network Server LoRaWAN possono gestire milioni di messaggi da migliaia di Gateway.
- *Viene garantita una comunicazione sicura* tra l'End-Node e l'Application Server, utilizzando la crittografia AES-128.
- *È possibile aggiornare da remoto il firmware* (applicazioni e stack LoRaWAN) per uno o più End-Node.
- *Si ha un basso costo di implementazione*, poiché sono presenti un'infrastruttura minima, degli End-Node a basso costo e un software open source.

3.1.2 Architettura del protocollo LoRaWAN

LoRaWAN è un protocollo MAC (Media Access Control) progettato per consentire ai dispositivi a bassa potenza di comunicare con le applicazioni connesse a Internet

su connessioni wireless a lungo raggio. LoRaWAN può essere mappato al secondo e terzo livello del modello OSI.

Nella Figura 1.4 erano stati già mostrati gli elementi principali della rete LoRaWAN. Essi sono di seguito elencati [11][15]:

- Gli End-Device, o End-Node, sono connessi in modalità wireless alla rete LoRaWAN, tramite dei Gateway che utilizzano la modulazione RF LoRa. Nella maggior parte delle applicazioni, un End-Node è un sensore autonomo, spesso alimentato a batteria, che digitalizza le condizioni fisiche e gli eventi ambientali. Invece, i casi d'uso tipici per un attuatore includono l'illuminazione stradale, le serrature wireless e la chiusura di valvole. Quando vengono prodotti, ad essi vengono assegnati diversi identificatori univoci. Questi vengono utilizzati per attivare e amministrare in modo sicuro il dispositivo, per garantire il trasporto sicuro dei pacchetti su una rete privata o pubblica e per fornire dati crittografati al Cloud.
- Un Gateway LoRaWAN riceve messaggi RF modulati LoRa da qualsiasi End-Node a distanza udibile e inoltra tali messaggi al Network Server LoRaWAN (LNS). Non esiste un'associazione fissa tra un End-Node e un Gateway specifico. Lo stesso End-Node, invece, può essere servito da più Gateway nell'area. Con il protocollo LoRaWAN, ogni pacchetto di uplink inviato da un End-Node sarà ricevuto da tutti i Gateway a cui il segnale arriva. Questa disposizione riduce significativamente il tasso di errore dei pacchetti, poiché le probabilità che almeno un Gateway riceva il messaggio sono molto alte; essa, inoltre, riduce significativamente il sovraccarico della batteria per i sensori mobili e consente la geolocalizzazione a basso costo (supponendo che i Gateway in questione siano in grado di geolocalizzare). Il traffico IP da un Gateway al Network Server può essere "backhauled" (ovvero fluisce verso la rete centrale grazie ad una porzione di rete, che può essere realizzata con diverse tecnologie) tramite Wi-Fi, Ethernet cablata o una connessione cellulare. I Gateway LoRaWAN operano interamente a livello fisico e, in sostanza, non sono altro che inoltri di messaggi radio LoRa. Essi controllano solo l'integrità dei dati di ogni messaggio LoRa RF in arrivo. Se l'integrità non è intatta, ovvero se il CRC (Cyclic Redundancy Check) non è corretto, il messaggio verrà eliminato. Se, invece, è corretto, il Gateway lo inoltrerà all'LNS, insieme ad alcuni metadati che includono il livello RSSI di ricezione del messaggio (che indica il livello di potenza ricevuto dopo ogni possibile perdita a livello di antenna e cavo) e un timestamp opzionale. Per i downlink LoRaWAN, un Gateway esegue le richieste di trasmissione provenienti dagli LNS senza alcuna interpretazione del payload. Poiché più Gateway possono ricevere lo stesso messaggio RF LoRa da un singolo End-Node, l'LNS esegue la deduplicazione dei dati ed elimina tutte le copie. I Gateway LoRaWAN possono essere classificati in Gateway interni (picocell, come il Gateway utilizzato in questo progetto) ed esterni (macrocell). I Gateway per interni forniscono copertura in luoghi interni difficili da raggiungere e sono, quindi, adatti per l'uso in abitazioni, aziende ed edifici.
- Il Network Server gestisce l'intera rete LoRaWAN e riceve il traffico IP dai Gateway. Esso è responsabile delle funzioni di gestione della rete come:
  - Gestione del processo di Over-The-Air Activation (che verrà visto nella Sezione 3.1.4) per gli End-Node da aggiungere alla rete. Esso contiene le in-

formazioni richieste per elaborare i frame di richiesta di join di uplink e generare i frame di accettazione di join di downlink, utilizzati per collegarsi alla rete. Inoltre, esegue le derivazioni della chiave crittografica della sessione di rete e dell'applicazione (Network e Application Session Key); successivamente, mantiene la Network Session Key (NwkSKey) prodotta e distribuisce Application Session Key (AppSKey) all'Application Server.
  - Deduplicazione dei messaggi, ovvero eliminazione dei messaggi duplicati ricevuti da più Gateway.
  - Routing dei messaggi, ovvero inoltro dei payload di uplink all'Application Server appropriato e accodamento dei payload di downlink provenienti da qualsiasi Application Server a qualsiasi dispositivo connesso alla rete.
  - Riconoscimento dei messaggi di dati confermati e di alcuni comandi MAC.
  - Adattamento della velocità di trasmissione dati utilizzando il protocollo ADR.
- L'Application Server elabora i messaggi di dati specifici dell'applicazione ricevuti dagli End-Node. Esso, inoltre, genera tutti i payload di downlink a livello di applicazione e li invia agli End-Node collegati tramite il Network Server. Una rete LoRaWAN può avere più di un Application Server. I dati raccolti possono essere, poi, interpretati e analizzati.

3.1.3 Tipi di messaggio

I messaggi LoRa possono essere suddivisi in messaggi uplink e downlink in base alla direzione in cui viaggiano [15].

I messaggi uplink vengono inviati dagli End-Node al Network Server, inoltrati da uno o più Gateway. Se il messaggio di uplink deve essere inviato all'Application Server, il Network Server lo inoltra al destinatario corretto.

Invece, ogni messaggio di downlink viene inviato dal Network Server a uno o più End-Node e ritrasmesso da un singolo Gateway. Ciò include, anche, alcuni messaggi avviati dall'Application Server.

Queste due tipologie di messaggi sono usate per trasmettere comandi MAC o dati dell'applicazione. I comandi MAC sono messaggi usati per inviare richieste e conferme all'interno della rete, come, ad esempio, i comandi di Join-request e Join-accept. Solitamente, un messaggio può contenere qualsiasi sequenza di comandi MAC, ma può anche trasportare contemporaneamente sia i comandi MAC che i dati dell'applicazione in campi separati.

3.1.4 Messa in servizio di un dispositivo

Per motivi di sicurezza, qualità del servizio, fatturazione e altre esigenze, i dispositivi devono essere messi in servizio e attivati sulla rete all'inizio del funzionamento. Il processo di commissioning allinea in modo sicuro ogni dispositivo e la rete rispetto ai parametri di provisioning essenziali (come identificatori, chiavi di crittografia e posizioni dei server).

LoRaWAN consente due tipi di attivazione: Over-the-Air Activation (OTAA) e Activation By Personalization (ABP) [15]. In questo progetto, come si vedrà nel Capitolo 4 è stata usata la seconda tipologia di attivazione.

OTAA è il metodo di attivazione più sicuro e consigliato per gli End-Node. I dispositivi eseguono una procedura di collegamento con il Network Server, durante la quale viene assegnato un indirizzo del dispositivo dinamico, e le chiavi di sicurezza vengono stabilite insieme all'End-Node e al Network Server.

Invece, ABP richiede, nell'End-Node, l'hardcoding dell'indirizzo del dispositivo e delle chiavi di sicurezza. ABP è meno sicuro di OTAA e ha anche lo svantaggio che i dispositivi non possono cambiare provider di rete senza cambiare manualmente le chiavi nel dispositivo.

Per quanto riguarda l'OTAA, la procedura di join richiede lo scambio di due messaggi MAC tra l'End-Node e il Network Server. Il primo è la richiesta di join (Join-request) che va dall'End-Node al Network Server; il secondo è il messaggio di Join-accept che va dal Network Server all'End-Node. Prima dell'attivazione, AppEUI, DevEUI e AppKey devono essere archiviati nell'End-Node. L'AppKey è una chiave segreta, crittografata con l'AES-128, nota come chiave root. La stessa AppKey deve essere fornita sulla rete in cui l'End-Node si registrerà. Il Network Server elabora il messaggio di richiesta di join. Esso genererà due chiavi di sessione (NwkSKey e AppSKey) e il messaggio Join-accept, se l'End-Node è autorizzato a unirsi alla rete; inoltre, mantiene la NwkSKey e distribuisce la AppSKey all'Application Server. L'End-Node decrittografa il messaggio di accettazione del join utilizzando l'operazione di crittografia AES. Esso, utilizzando AppKey, deriverà le due chiavi di sessione, ovvero NwkSKey e AppSKey. L'AppEUI e il DevEUI non sono segreti e visibili a tutti, mentre l'AppKey non viene mai inviata attraverso la rete. In Figura 3.2 vengono descritti i passaggi per tale procedura.

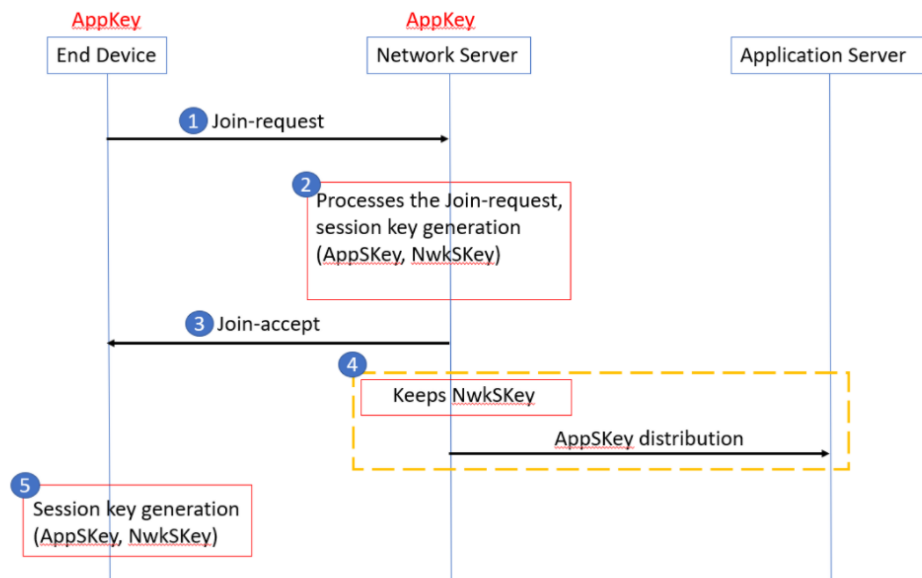


Figura 3.2. Flusso di messaggi per l'Over-The-Air Activation (OTAA)

Per quanto riguarda l'ABP, essa collega direttamente un End-Node a una rete

preselezionata, ignorando la procedura OTAA. Un End-Node attivato utilizzando il metodo ABP può funzionare solo con una singola rete e mantiene la stessa sessione di sicurezza per tutta la sua durata. Il DevAddr e le due chiavi di sessione NwkSKey e AppSKey vengono archiviati direttamente nell'End-Node, al posto della DevEUI, dell'AppEUI e dell'AppKey. Ogni End-Node dovrebbe avere un set univoco di NwkSKey e AppSkey. Gli stessi DevAddr e NwkSKey devono essere archiviati nel Network Server e l'AppSKey deve essere archiviata nell'Application Server. In Figura 3.3 è riassunto quanto detto.

Figura 3.3. Pre-condivisione di DevAddr e delle chiavi di sessione per l'Activation By Personalization (ABP)

3.1.5 Classi A, B e C

Gli End-Node basati su LoRa possono funzionare in Classe A, B o C. Tutti questi dispositivi devono supportare il funzionamento di Classe A. I dispositivi di Classe B devono supportare entrambe le modalità di Classe A e di Classe B e i dispositivi di Classe C devono supportare tutte e tre le modalità di funzionamento. Le Classi A, B e C riguardano il modo in cui i dispositivi comunicano con la rete.

Nel caso della Classe A, l'End-Node trascorre la maggior parte del tempo in uno stato di inattività (ovvero in modalità di sospensione). Quando c'è un cambiamento nell'ambiente, relativo a qualunque cosa il dispositivo è programmato per monitorare, il dispositivo si riattiva e avvia un uplink, trasmettendo i dati, relativi al cambiamento di stato, alla rete (Tx), attendendo, poi, da quest'ultima una risposta. Se non riceve un downlink durante questa finestra di ricezione (Rx_1), torna brevemente a dormire, svegliandosi di nuovo qualche istante dopo, in ascolto, ancora una volta, di una risposta (Rx_2). Se non viene ricevuta alcuna risposta durante questa seconda finestra, il dispositivo torna in modalità di sospensione fino alla prossima volta che ha dati da segnalare, rientrando in modalità di trasmissione Tx. Una limitazione molto importante, che non rende adatta la Classe A per gli attuatori, è che non c'è modo dall'applicazione di svegliare un dispositivo in Classe A [7].

Un miglioramento della Classe A è la modalità LoRaWAN Classe B, che offre agli End-Node la possibilità di avere finestre temporali fisse e regolarmente schedulate, per ricevere downlink dalla rete, oltre a quelle che si aprono ogni volta che un uplink

di Classe A viene inviato al Network Server. Tutto ciò rende gli End-Node di Classe B adatti sia per il monitoraggio dei sensori che per gli attuatori. Tutti gli End-Node basati su LoRa si avviano in modalità Classe A; tuttavia, i dispositivi programmati, in produzione, con uno stack di Classe B possono essere passati alla modalità di Classe B dal livello dell'applicazione. Affinché la modalità di comunicazione di Classe B funzioni, è necessario un processo chiamato beaconing . Durante tale processo di beaconing, un beacon sincronizzato nel tempo deve essere trasmesso periodicamente dalla rete, tramite i Gateway. L'End-Node deve ricevere periodicamente uno di questi beacon di rete in modo da allineare il suo riferimento temporale interno con la rete [8].

Infine, vi è la Classe C, che è sempre "accesa", trascurando la carica della batteria. Questi dispositivi sono sempre in ascolto di messaggi di downlink, a meno che non trasmettano un uplink. Di conseguenza, offrono la latenza più bassa per la comunicazione dal server a un End-Node. Gli End-Node di Classe C implementano le stesse due finestre di ricezione dei dispositivi di Classe A, ma non chiudono la finestra Rx_2 fino a quando non inviano la trasmissione successiva al server. Pertanto, possono ricevere un downlink nella finestra Rx_2, praticamente in qualsiasi momento [9].

Poiché in questo progetto si ha la necessità di poter ricevere sempre dei messaggi di downlink dal Gateway per avviare e sincronizzare l'acquisizione, è necessario implementare una comunicazione in Classe C sulla STM32 presente nell'End-Node.

3.2 Configurazione di Chirpstack

Come già detto nel Capitolo 1, per la realizzazione del Network Server e dell'Application Server, e per la gestione della comunicazione tra il Network Server e il Gateway LoRaWAN Picocell SX1308, è stato utilizzato Chirpstack, che fornisce componenti open source per le reti LoRaWAN. Come prima cosa bisogna procedere all'installazione del *ChirpStack Gateway Bridge*, del *ChirpStack Network Server* e del *ChirpStack Application Server*, con la creazione dei servizi *chirpstack-network-server* e *chirpstack-application-server*. La guida per l'installazione e la configurazione dei servizi non verrà riportata, poiché essa è già disponibile, in maniera molto dettagliata, nel sito ufficiale di Chripstack `https://www.chirpstack.io/project/guides/debian-ubuntu/`.

Invece, è importante analizzare la configurazione di Chripstack effettuata. Come prima cosa, bisogna accedere alla web interface dell'Application Server, selezionare *Network-servers* e premere *ADD*. Occorre, quindi, compilare la scheda *GENERAL*, nella quale bisogna inserire il nome del nuovo server, l'indirizzo `127.0.0.1` (localhost) e la porta 8000 del Network-Server; questi sono derivati dal file *chirpstack-network-server.toml* creato durante l'installazione. Infine, è necessario salvare premendo *ADD NETWORK-SERVER*. Ciò è visibile in Figura 3.4.

Successivamente, bisogna aggiungere il Gateway. Prima di farlo bisogna, però, controllare il file `global_conf.json`, prodotto durante l'istallazione del packet forwarder, responsabile, insieme al *ChirpStack Gateway Bridge*, della gestione del Gateway LoraWAN Picocell SX1308. In questo file deve essere presente il campo *server_address*, definito pari all'hostname del Gateway, che, in questo caso, è

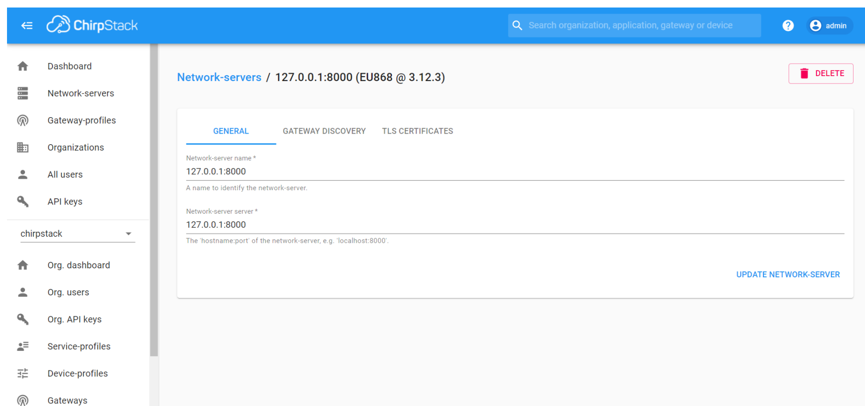


Figura 3.4. Configurazione del Network Server in Chirpstack

localhost; inoltre, i parametri *serv_port_up* e *serv_port_down* devono fare riferimento alla porta 1700. Se tutto ciò è verificato, è sufficiente compilare la scheda *GENERAL*, inserendo un nome, una descrizione e l'ID univoco del Gateway, anche questo definito nel file `global_conf.json`. È necessario, poi, aggiungere un nuovo *Service-profile*, inserendo il valore 7 alla voce *Maximum allowed data-rate*. Infine, è necessario salvare la configurazione. Ciò è mostrato nella Figura 3.5.

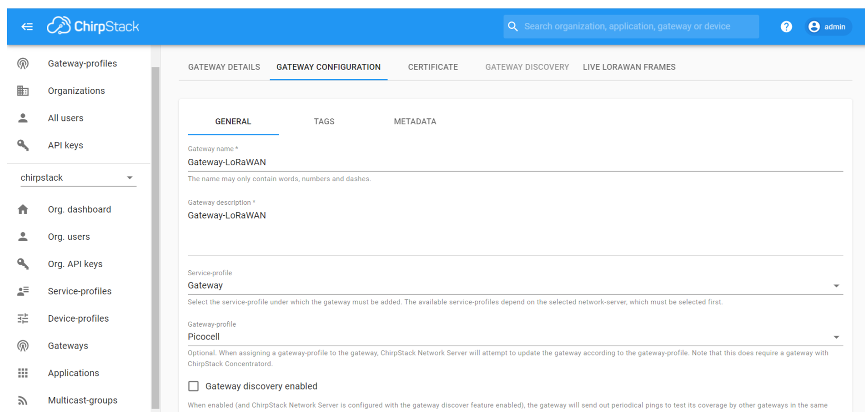


Figura 3.5. Configurazione del Gateway in Chirpstack

A questo punto, si passa ad aggiungere un nuovo *Device-profile*. Nella scheda *GENERAL* bisogna scegliere un nome, selezionare la versione LoRaWAN 1.0.3 supportata attualmente dall'STM32-B-L072Z-LRWAN1, con *Regional Parameters revision* pari a B, come mostrato in Figura 3.6.

Invece, nella scheda *JOIN*, bisogna deselezionare il parametro *Device supports OTAA*, poiché la modalità di attivazione scelta è la ABP, dal momento che tale modalità è quella prevista per la configurazione del Multicast. È necessario, quindi, inserire 1 come *RX1 delay*, e inserire 869525000 come *RX2 channel frequency*. Inol-

Figura 3.6. Configurazione di un nuovo *Device-profile* in Chirpstack

tre, nella textbox *Factory-preset-frequencies*, si deve inserire la lista delle frequenze 868100000, 868300000, 868500000, 867100000, 867300000, 867500000, 867700000, 867900000. Ciò è mostrato in Figura 3.7.

Figura 3.7. Configurazione della procedura di join per gli End-Node in Chirpstack

Infine, nella scheda *CLASSE-C*, è necessario spuntare la casella *Device supports Class-C*, poiché, come già detto, questa sarà la classe utilizzata negli End-Node per soddisfare le esigenze richieste.

Successivamente, è necessario creare una nuova *Application*, inserendo il nome (in questo caso *Accelerometri*) e la descrizione. Dalla pagina *DEVICES*, bisogna creare i device associati; innanzitutto bisogna associare al device un nome (in questo caso *STMB-L072Z*) e una descrizione; poi, nella scheda *ACTIVATION*, è necessario associare al dispositivo un address (*Device address*), una *Network session key* e una *Application session key*, che devono essere identici a quelli impostati nel file `Commissioning.h`, presente nel firmware dell'STM32 dell'End-Node, come era stato già spiegato nella Sezione 3.1.4. Tutto ciò è mostrato in Figura 3.8.

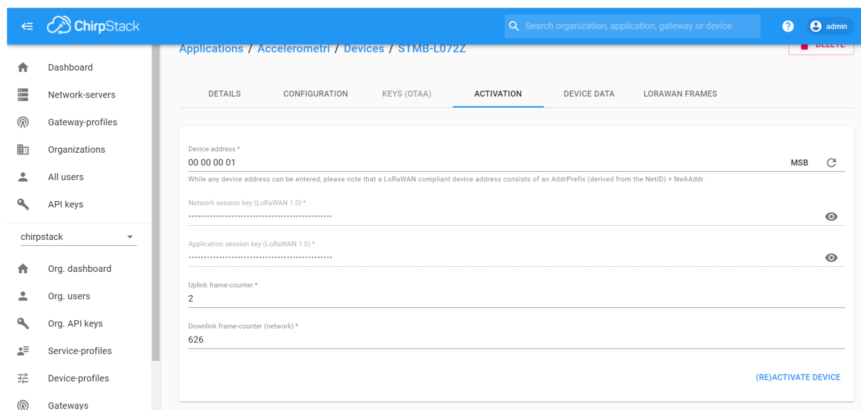


Figura 3.8. Definizione del *Device address*, della *Network session key* e della *Application session key* in Chirpstack

Come ultima operazione, è necessario creare un *MULTICAST GROUP*, inserendo l'indirizzo del Multicast, la *Multicast network session key*, la *Multicast application session key* e la frequenza, tutti uguali ai parametri impostati all'interno del file `Commissioning.h`. Infine, è necessario associare nel *Multicast Group* i dispositivi aggiunti precedentemente nella pagina *Application*. Ciò è mostrato in Figura 3.9.

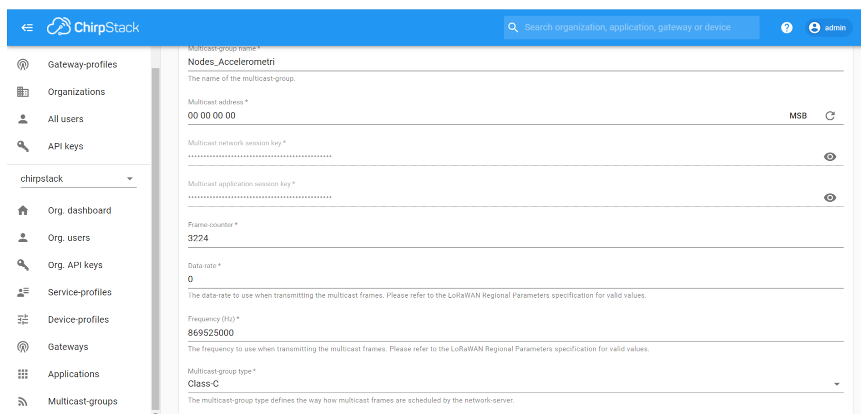


Figura 3.9. Definizione dei parametri del Multicast in Chirpstack

3.3 Doppia funzionalità del Gateway

In questo progetto il Gateway, quindi quello che comprende la Raspberry Pi 3B+ e il Gateway LoRaWAN Picocell SX1308, svolge una doppia funzione, che è stata già spiegata ma mai puntualizzata. In particolare, esso svolge la funzione di client

MQTT, come visto nel Capitolo 2, e proprio grazie ad esso vengono ricevute tutte le informazioni necessarie per la configurazione delle acquisizioni. Esso, quindi, elabora tali informazioni per schedulare le varie attività che deve compiere, che consistono nell'invio dei vari messaggi di configurazione e di sincronizzazione agli End-Node.

La seconda funzione che svolge è, appunto, quella da Gateway LoRaWAN, grazie all'utilizzo del Gateway Picocell SX1308. Come spiegato in questo capitolo, il Gateway LoRaWAN, sostanzialmente, è un dispositivo che si occupa di effettuare l'inoltro dei messaggi che vengono ricevuti dal Network Server. Però, nella sezione 2.2 veniva impropriamente detto che il Gateway inviava i messaggi tramite protocollo LoRaWAN all'End-Node, poiché ancora non era stato spiegato come funziona nel dettaglio il protocollo LoRaWAN. Quello che, invece, avviene effettivamente è che il Gateway invia il messaggio, che deve ricevere l'End-Node, sul Network Server, utilizzando la libreria `requests.py` e il metodo *post()*. Tale metodo invia una richiesta di POST all'url specificato e viene utilizzato quando si desidera inviare alcuni dati al server.

Sarà, quindi, il Network Server LoRaWAN che farà partire il messaggio di downlink verso l'End-Node, che verrà inoltrato dal Gateway LoRaWAN, gestito dal servizio *packet_forwarder*, e dal *ChirpStack Gateway Bridge*, che permette la comunicazione tra il Network Server e il Gateway LoraWAN.

4

Analisi End-Node

In questo capitolo verrà svolta un'analisi dettagliata della struttura del codice che è stato implementato all'interno degli End-Node, che, come già ripetuto più volte, è formato da due microcontrollori diversi. Inoltre, verranno descritte le caratteristiche del convertitore A-D che è stato usato, specificando, appunto, perché la scelta sia ricaduta proprio su questo componente. Infine, verranno analizzate le caratteristiche dell'accelerometro scelto.

4.1 STM32 BL072Z

La board STM32 ha sostanzialmente due compiti principali, ovvero gestire la comunicazione con il Gateway tramite protocollo LoRaWAN e inoltrare i messaggi, relativi al Template da eseguire e quelli necessari per la sincronizzazione, alla Raspberry contenuta nell'End-Node, tramite protocollo UART.

4.1.1 Ricezione messaggi LoRaWAN

Come già accennato nel Capitolo 1, ST fornisce un pacchetto di espansione software chiamato I-CUBE-LRWAN, in grado di supportare trasmissioni in Classe A, B e C per comunicazioni tramite protocollo LoRaWAN. Nel progetto preso in esame, come già detto nel Capitolo 3, la comunicazione tra Gateway ed End-Node avverrà in Classe C, poiché è quella che rispetta meglio le esigenze richieste.

Per prima cosa, è importante capire come è strutturato il software I-CUBE-LRWAN fornito da ST. La sua architettura è mostrata in Figura 4.1.

Il pacchetto di espansione I-CUBE-LRWAN si basa sui driver HAL (Hardware Abstraction Layer) di STM32Cube. Esso rappresenta un'iniziativa originale di ST-Microelectronics, nato per migliorare la progettazione dei software, riducendo lo sforzo, i tempi e i costi di sviluppo, e coprendo l'intera gamma di schede STM32 [2]. Esso include:

- Un insieme di strumenti di sviluppo software user-friendly per coprire lo sviluppo del progetto dall'ideazione alla realizzazione, tra cui si trova anche STM32CubeIDE, un tool di sviluppo all-in-one con funzionalità di configurazione delle periferiche, generazione di codice, compilazione del codice e debug.

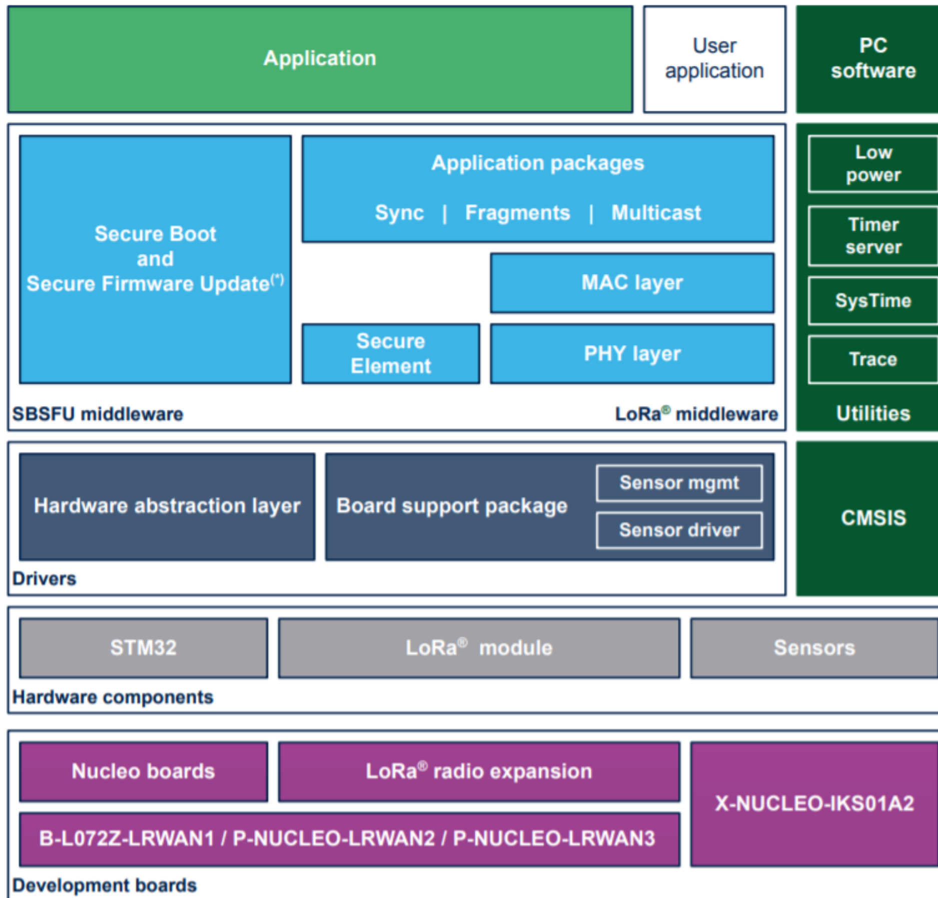


Figura 4.1. Architettura del pacchetto di espansione I-CUBE-LRWAN

- STM32Cube MCU and MPU Packages, pacchetti software integrati specifici per ogni serie di microcontrollori (MCU) e microprocessori (MPU). Questi includono:
  - un layer di astrazione hardware (HAL) STM32Cube, che garantisce la massima portabilità nella gamma di schede STM32;
  - delle API STM32Cube a basso livello, che garantiscono migliori prestazioni e un footprint con un alto grado di controllo dell'utente sull'hardware;
  - un insieme coerente di componenti del middleware come il file system FAT, e librerie RTOS, USB e Touch;
  - delle software utilities integrate con set completi di esempi applicativi.
- Pacchetti di espansione STM32Cube che contengono componenti software embedded che completano le funzionalità dei pacchetti STM32Cube MCU e MPU con estensioni del middleware e dei livelli applicativi, ed esempi in esecuzione su alcune schede di sviluppo STMicroelectronics.

Proprio tra questi pacchetti di espansione c'è, appunto, I-CUBE-LRWAN. Esso

offre uno stack middleware per microcontrollori della serie STM32 [3]. Il pacchetto completo include:

- Lo stack middleware LoRa, costituito da:
  - LoRaMac layer module;
  - LoRa utilities module;
  - LoRa crypto module;
  - LoRa core module.
- Il Board Support Package, costituito da:
  - driver per la Radio Semtech;
  - driver per i sensori ST (di pressione, umidità e temperatura).
- I driver per i STM32 HAL.
- Diverse Utilities, tra cui:
  - il Tool sequencer, che fornisce servizi per gestire le attività;
  - il Timer server, che fornisce il servizio timer all'applicazione;
  - la gestione Low-Power (a basso consumo), che fornisce un servizio di gestione dell'alimentazione all'applicazione.
- Degli esempi di applicazioni LoRa, tra cui si trova l'applicazione End-Node, che è quella dalla quale si è partiti per lo sviluppo del software finale.

L'HAL utilizza le API STM32Cube per guidare l'hardware del microcontrollore secondo quanto richiesto dall'applicazione.

L'RTC (Real-Time-Clock) fornisce un'unità di tempo centralizzata che continua a funzionare anche in modalità a basso consumo.

Il driver radio utilizza l'SPI e i GPIO (General Purpose Input Output) per controllare la radio. Esso fornisce, anche, una serie di API che possono essere utilizzate da software di livello superiore. La radio LoRa è fornita da Semtech, sebbene le API siano state leggermente modificate per interfacciarsi con STM32Cube HAL.

Il MAC layer si interfaccia con il driver del PHY layer e utilizza il Timer server per aggiungere o rimuovere Task temporizzati, e per occuparsi del *Tx time on air*, ovvero il tempo di trasmissione. Questa azione garantisce che venga eseguito l'algoritmo di crittografia/decrittografia AES, per crittografare l'intestazione MAC e il payload, ovvero l'informazione trasmessa.

È possibile osservare la struttura dei file che costituiscono il progetto in Figura 4.2.

L'applicazione End-Node, ovvero quella dalla quale si è partiti per lo sviluppo del programma finale, implementa un dispositivo End-Node della rete LoRaWAN. Nella sua forma originale, essa legge la temperatura, l'umidità e la pressione atmosferica dai sensori attraverso l'I2C, e misura la tensione d'alimentazione che viene fornita ad essa per calcolare il livello della batteria. Questi quattro dati (temperatura, umidità, pressione atmosferica e livello della batteria) vengono inviati periodicamente alla rete LoRa utilizzando la radio LoRa in Classe A a 868 MHz. Per avviare il progetto LoRa End_Node, l'utente deve andare su `/Projects/B-L072Z-LRWAN1/Applications/LoRa/End_Node` e scegliere la cartella toolchain `LoRaWAN/App`; infine, deve selezionare il progetto LoRa dalla target board appropriata.

Esistono due modi per attivare un dispositivo sulla rete LoRaWAN, ovvero tramite OTAA o tramite ABP; ciò viene impostato all'interno del file `/Projects/B-`

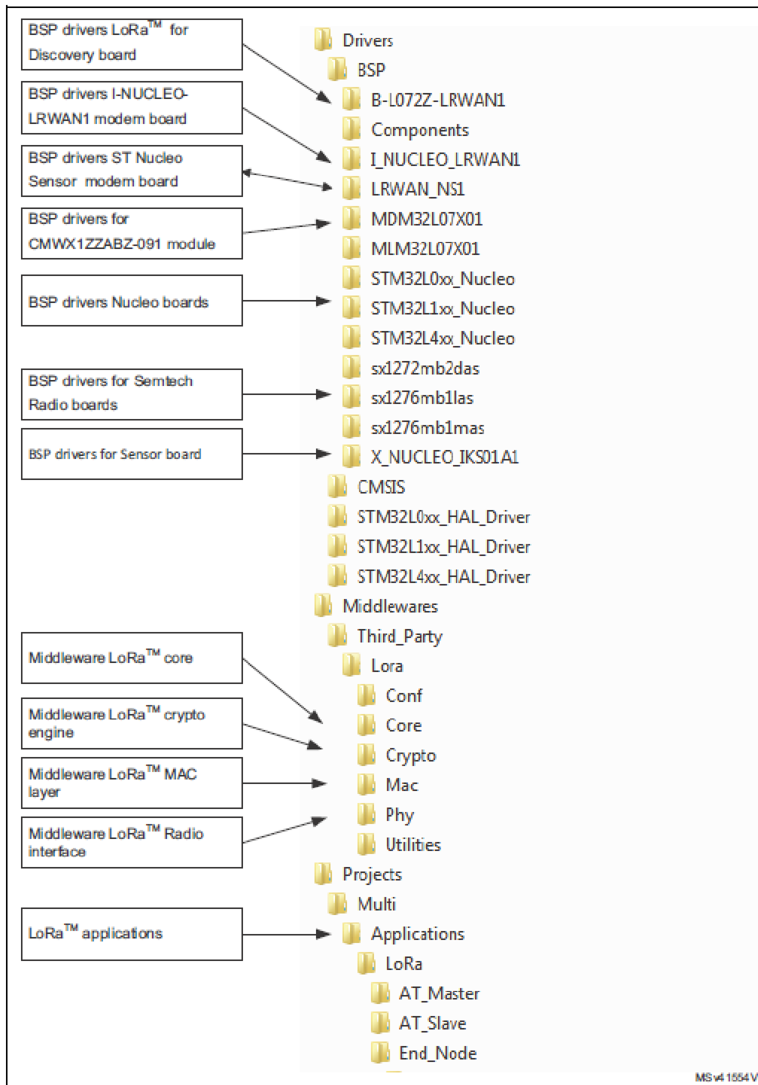


Figura 4.2. Struttura dei file di progetto

`L072Z-LRWAN1/Applications/LoRa/End_Node/LoRaWAN/App/inc/Commissioning.h`, che raccoglie, appunto, tutti i dati relativi all'attivazione del dispositivo. Il metodo scelto, insieme ai dati di messa in servizio, sono stampati sulla porta virtuale e visibili su terminale.

L'OTAA (Over-The-Air Activation) è una procedura di adesione per l'End-Node LoRa che permette di partecipare a una rete LoRaWAN. Sia l'End-Node LoRa che l'Application Server condividono la stessa chiave segreta nota come AppKey. Durante una procedura di Join, l'End-Node LoRa e l'Application Server si scambiano dati per generare due chiavi di sessione, ovvero:

- una chiave di sessione di rete (NwkSKey) per la crittografia dei comandi MAC;
- una chiave di sessione dell'applicazione (AppSKey) per la crittografia dei dati dell'applicazione.

Nel caso di ABP (Activation By Personalization), NwkSkey e AppSkey sono già memorizzati nell'End-Node LoRa, che invia i dati direttamente alla rete LoRa.

Per ciò che concerne il codice fornito da I-CUBE-LRWAN, sono state apportate delle modifiche soltanto sui seguenti tre file:

- `Commissioning.h`, che contiene le definizioni dei parametri LoRaWAN;
- `main.c`, che contiene il loop principale del sistema, che si occupa della ricezione dei messaggi LoRaWAN nonché dell'invio, tramite UART, dei comandi alla Raspberry;
- `lora.c`, che contiene l'inizializzazione del protocollo LoRaWAN.

In particolare, le modifiche effettuate sui file `Commissioning.h` e `lora.c` sono state quelle necessarie per implementare una comunicazione LoRaWAN di Classe C che permettesse l'utilizzo del Multicast.

Nel file `Commissioning.h` è stata attivata la ABP (una delle due modalità di attivazione del dispositivo sulla rete), ponendo a 0 il flag *OVER-THE-AIR-ACTIVATION*. È stata scelta questa come modalità poiché è l'unica che permette di utilizzare il Multicast. Inoltre, sono stati impostati l'EUI (Extended Unique Identifier) e l'Address del dispositivo, così come configurati precedentemente in Chirpstack, attraverso la definizione dei due parametri *LORAWAN_DEVICE_EUI* e *LORAWAN_DEVICE_ADDRESS*, e ponendo ad 1 i parametri *STATIC_DEVICE_EUI* e *STATIC_DEVICE_ADDRESS*.

Infine, sono stati definiti i campi *MULTICAST_CHANNEL_0_NWKSKEY* e *MULTICAST_CHANNEL_0_APPSKEY*, per definire i valori delle chiavi di sessione del server e dell'applicazione nel caso Multicast; tali valori sono stati riportati in Chirpstack, nella sezione Multicast Groups.

Nel file `lora.c`, per prima cosa, è stato abilitato il Multicast, definendo il flag *MULTICAST*. Poi, all'interno della funzione di inizializzazione *Lora_Init()* è stato aggiunto lo snippet riportato nel Listato 4.1. Esso consente di abilitare e di configurare il Multicast all'interno dei singoli nodi, attraverso gli opportuni parametri del protocollo LoRaWAN. I parametri elencati in tale snippet sono stati aggiunti anche in Chirpstack.

```
#if defined MULTICAST
  // MC channel 0 App session key setup
  mibReq.Type = MIB_MC_APP_S_KEY_0;
  mibReq.Param.McAppSKey0 = ( uint8_t* )Multicast0AppSKey;
  LoRaMacMibSetRequestConfirm( &mibReq );

  // MC channel 0 Nwk session key setup
  mibReq.Type = MIB_MC_NWK_S_KEY_0;
  mibReq.Param.McNwkSKey0 = ( uint8_t* )Multicast0NwkSKey;
  LoRaMacMibSetRequestConfirm( &mibReq );

  mibReq.Type = MIB_MC_KE_KEY;
  mibReq.Param.McNwkSKey0 = ( uint8_t* )MulticastEKey;
  LoRaMacMibSetRequestConfirm( &mibReq );

  // Setup Channel 0
  channel.Address = Multicast_Address;
  channel.IsEnabled = true;
  channel.Class = CLASS_C;
  channel.GroupID = MULTICAST_0_ADDR;
  channel.FCountMin = 0;
  channel.FCountMax = 1000;
  RxParams.ClassC.Datarate = 0;
  RxParams.ClassC.Frequency = 869525000;
```

```
  channel.RxParams = RxParams;
  //channel.McKeyE = 0x00;
  LoRaMacMcChannelSetup( &channel );
#endif
```

Listato 4.1. Codice inserito nella funzione *Lora_Init()* per abilitazione e configurazione del Multicast

4.1.2 Invio dei messaggi tramite UART

La gestione della comunicazione dell'UART, per l'invio dei messaggi alla Raspberry, tramite questo protocollo, è stata effettuata interamente all'interno del `main.c`.

Innanzitutto, sono state definite due nuove funzioni, *gpioUART_Init()* ed *UART_Init()*, necessarie per l'inizializzazione dei GPIO utilizzati per la comunicazione e per l'inizializzazione dei parametri che definiscono la comunicazione, come ad esempio quale UART viene utilizzato (poiché ve ne sono due, e viene scelta la UART1) e il BaudRate. Nel Listato 4.2 si può osservare ciò che è stato appena descritto.

```
/*GPIO UART INIT*/
void gpioUART_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStruct;
    __GPIOA_CLK_ENABLE();
    __USART1_CLK_ENABLE();

    /*PA9->USART1_TX  PA10->USART1RX */
    GPIO_InitStruct.Pin = GPIO_PIN_9 | GPIO_PIN_10;
    GPIO_InitStruct.Mod = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_LOW;
    GPIO_InitStruct.Alternate = GPIO_AF4_USART1;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
}

/*UART INIT*/
void UART_Init(void)
{
    huart1.Instance = USART1;
    huart1.Init.BaudRate = 115200;
    huart1.Init.WordLength = UART_WORDLENGTH_8B;
    huart1.Init.StopBits = UART_STOPBITS_1;
    huart1.Init.Parity = UART_PARITY_NONE;
    huart1.Init.Mode = UART_MODE_TX_RX;
    huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;
    huart1.Init.OverSampling = UART_OVERSAMPLING_16;
    huart1.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
    huart1.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
    if (HAL_UART_Init(&huart1) != HAL_OK)
    {
    Error_Handler();
    }
}
```

Listato 4.2. Definizione delle funzioni utilizzate per l'inizializzazione dell'UART

Queste due funzioni vengono, poi, richiamate all'interno del *main()* principale.

Le ultime modifiche fatte riguardano la funzione *Lora_RxData()*, che è quella che viene richiamata per elaborare i frame in arrivo tramite LoRaWAN; è proprio all'interno di questa funzione che ST indica che l'utente è libero di sviluppare il proprio codice per la gestione dei messaggi in entrata.

Nel Listato 4.3 è possibile vedere come tale funzione sia stata implementata.

```
static void LORA_RxData(lora_AppData_t *AppData)
{
  /* USER CODE BEGIN 4 */
  PRINTF("PACKET RECEIVED ON PORT %d\n\r", AppData->Port);

  switch (AppData->Port)
```

```
    {
      case 3:
        /*this port switches the class*/
        if (AppData->BuffSize == 1)
        {
          switch (AppData->Buff[0])
          {
            case 0:
            {
              LORA_RequestClass(CLASS_A);
              break;
            }
            case 1:
            {
              LORA_RequestClass(CLASS_B);
              break;
            }
            case 2:
            {
              LORA_RequestClass(CLASS_C);
              break;
            }
            default:
              break;
          }
        }
        break;
      case LORAWAN_APP_PORT:
        if (AppData->BuffSize == 1)
        {
          AppLedStateOn = AppData->Buff[0] & 0x01;
          if (AppLedStateOn == RESET)
          {
            PRINTF("LED OFF\n\r");
            LED_Off(LED_BLUE) ;
          }
          else
          {
            PRINTF("LED ON\n\r");
            LED_On(LED_BLUE) ;
          }
        }
        break;
      case LPP_APP_PORT:
      {
        AppLedStateOn = (AppData->Buff[2] == 100) ? 0x01 : 0x00;
        if (AppLedStateOn == RESET)
        {
          PRINTF("LED OFF\n\r");
          LED_Off(LED_BLUE) ;

        }
        else
        {
          PRINTF("LED ON\n\r");
          LED_On(LED_BLUE);
        }
        break;
      }
      case PING_DEVICE_PORT:
      //Transmit Proj_code and UID to RaspBerry and low level id
      {
          device_active = true;
          LORA_send(&AppData, LORAWAN_DEFAULT_CONFIRM_MSG_STATE);
          int i = 0;
          while (AppData->Buff[i]!='\n')
          {
                  proj_code[i] = AppData->Buff[i];
                  i++;
          }
          proj_code[i] = '/';
          proj_code_len = i+1;
          int j = 0;
          i++;

          while (AppData->Buff[i]!='\n')
          {
                  dev_id[j] = AppData->Buff[i];
                  i++;
                  j++;
          }
          dev_id[j]= '/';
          dev_id_len = j+1;
          i++;
          int k = 0;
          while (i<AppData->BuffSize)
          {
              ll_id[k] = AppData->Buff[i];
              low_level_id[k] = AppData->Buff[i];
              i++;
              k++;
          }
          ll_id[k]= '/';
          ll_id[k+1]='?';
```

```
        ll_id_len = k+2;
        //
        HAL_UART_Transmit(&huart1, proj_code, proj_code_len, 300);
        HAL_UART_Transmit(&huart1, dev_id, dev_id_len, 300);
        HAL_UART_Transmit(&huart1, ll_id, ll_id_len, 300);
        //
        //Send response
        LED_On(LED_BLUE) ;
        HAL_Delay(1000);
        LED_Off(LED_BLUE);
        break;
    }
    case DELETE_DEVICE_PORT:
    //set device_active flag
    {
        device_active = false;
        LORA_send(&AppData, LORAWAN_DEFAULT_CONFIRM_MSG_STATE);
        PRINTF("Device Deleted!\n\r");
        LED_On(LED_BLUE) ;
        HAL_Delay(1000);
        LED_Off(LED_BLUE);
        break;
    }
    case SYNCH_PORT:
    {
        int i = 0;
        while (AppData->Buff[i]!='@')
        {
                device_list[i] = AppData->Buff[i];
                i++;
        }
        device_list[i] = '@';
        device_list_len = i+1;
        match_found = check_uid(low_level_id, device_list, device_list_len);
        type = AppData->Buff[i+1];

                if ( type == TRIGGER && match_found==true )
                {
                        acquisition_ended = false;
                        acquisition_started = true;
                        HAL_UART_Transmit(&huart1, &trigger, 1, 200);
                        //start timer
                        StartTimeoutTimer(interval_to_load);
                }
                if (type == BEACON && match_found==true)
                {
                        HAL_UART_Transmit(&huart1, &beacon, 1, 200);
                }
                if (type == END && match_found==true)
                {
                        HAL_UART_Transmit(&huart1, &end, 1, 200);
                        acquisition_ended = true;
                        acquisition_started = false;
                }
                break;

    }
    default:
    //Mando template_code, sample rate, device list alla RaspBerry
    //Memorizzo il tempo della acquisizione per far caricare il timer in caso di perdita del messaggio end
    //TODO: se il dispositivo non appartiene alla lista dispositivi non mando nemmeno le configurazioni
    {
      if (AppData->BuffSize > 2 && device_active == true){
          uint8_t payload1[AppData->BuffSize];
          for (int i=0; i < AppData->BuffSize; i++)
          {
                  payload1[i] = AppData->Buff[i];
          }
          //RECUPERO L'INTERVALLO DELLA SCANSIONE(1)
          int i = 0;
          while (payload1[i] != '/')
          {
                interval_string[i] = payload1[i];
                i++;
          }
          //Durata della acquisizione in secondi
          interval = atoi(interval_string);
          //Converto in ms
          interval_ms = (interval*1000)+10000;
          //////////////////RECUPERO LISTA DISPOSITIVI TEMPLATE DA AVVIARE///////////////

          int z = 0;

          while (payload1[z] != '@')
          {
                  z++;
          }
          z--;
          int kk = 0;
          while(payload1[z]!='/')
          {
                  device_list[kk] = payload1[z];
                  z--;
                  kk++;
```

```
            }
            device_list[kk] = '@';
            device_list_len = kk+1;
            match_found = check_uid(low_level_id, device_list, device_list_len);
            if (match_found == true)
            {
            //Alla Raspberry non interessa la durata quindi mando solo quello che c'è dopo
                    HAL_UART_Transmit(&huart1, &payload1[i+1], AppData->BuffSize-(i+1),300 );
                    interval_to_load = interval_ms;
            }
        }
      }
      break;
    }
  /* USER CODE END 4 */
}
```

Listato 4.3. Implementazione della funzione *Lora_RxData()* utilizzata per elaborare i frame in arrivo dal Gateway

Il codice è stato sviluppato in modo da gestire in maniera diversa i frame che arrivano, in base alla porta di ricezione del messaggio. In particolare i messaggi che arrivano sulle porte 3, 4 (*SYNCH_PORT*), 5 (*PING_DEVICE_PORT*) e 6 (*DELETE_DEVICE_PORT*) sono stati separati dagli altri, che ricadono, invece, nel caso di default.

La porta 3 è stata riservata per l'arrivo del messaggio che contiene la Classe di trasmissione da impostare. In particolare esso può avere un valore pari a 0, 1, 2, che indicano, rispettivamente, le classi A, B e C.

La porta 5 è stata riservata per ricevere i messaggi relativi al template che viene eseguito, che comprende la ricezione dell'ID del device, del low level ID e del codice del progetto, che vengono, poi, trasmessi anche alla Raspberry. L'ID del device verrà utilizzato per caricare il file `.csv` nella giusta cartella dell'FTP server, mentre il low level ID identifica l'ID dell'End-Node nel protocollo LoRaWAN. Nel caso di arrivo di un messaggio su questa porta, innanzitutto, si manda un messaggio di conferma al Gateway di avvenuta ricezione e, solo successivamente, si passa alla sua valutazione.

La board STM32 riceve un unico messaggio tramite protocollo LoRaWAN, che contiene le tre informazioni sopra citate. Esse sono separate dal carattere "\n". Vengono, così, creati tre vettori, uno per ogni informazione da trasmettere; questi vengono, poi, inviati tramite la funzione *HAL_UART_Transmit()*, sul canale UART, alla Raspberry. Come ultimo elemento dei vettori che identificano il codice progetto e l'ID del device, viene messo il carattere "/", mentre in fondo al vettore che identifica il low level ID vengono messi i caratteri "/" e "?".

La porta 4 è stata dedicata all'arrivo dei messaggi di sincronizzazione, ovvero TRIGGER, BEACON ed END. Quando arriva un messaggio, esso contiene anche la lista dei device a cui è riferito, quindi, per prima cosa, la board STM32 verifica se tale messaggio è rivolto anche ad essa, altrimenti lo scarta. Tale operazione viene fatta invocando la funzione *check_uid()* (Listato 4.4), che verifica se il proprio ID è contenuto all'interno della lista degli ID ricevuti.

```
bool check_uid(uint8_t *ll_id, uint8_t *device_list, uint8_t device_list_len)
{
        const char delim = ',';
        //uint8_t *pointer = device_list;
        int k = 0;
        char device[3];
        for(int i = 0; i<device_list_len; i++)
        {
                if (device_list[i] != ',' && device_list[i] !='@')
                {
                        device[k] = device_list[i];
                        k++;
```

```
                }
                else
                {
                        device[k] = '\0';
                        k = 0;
                        if (strcmp(device, ll_id) == 0)
                        {
                                return true;
                        }
                }
        }
        return false;
}
```

Listato 4.4. Definizione della funzione per la verifica della presenza dell'ID della scheda all'interno della lista ricevuta

Se quest'ultima funzione ha esito positivo, si passa alla valutazione del tipo di segnale di sincronizzazione ricevuto. In tutti e tre i casi il messaggio ricevuto tramite protocollo LoRaWAN viene inoltrato alla Raspberry tramite UART, utilizzando la funzione *HAL_UART_Transmit()*.

Nel caso del TRIGGER, oltre a porre il flag *acquisition_started* al valore *True*, viene attivata la funzione *StartTimeoutTimer()*, che attiva un timer con durata pari al tempo di acquisizione (sempre ricevuto via LoRa) più venti secondi, in modo che, se non giungerà nessun messaggio di END, la board STM32 fermerà automaticamente l'acquisizione, inviando un messaggio di END sull'UART.

Nella porta 6, invece, è atteso il messaggio che disattiva l'End-Node; anche in questo caso, come avveniva sulla porta 5, la board STM32 invia al Gateway un messaggio di avvenuta ricezione.

Infine, nel caso di default, sono stati inseriti la ricezione e l'inoltro della configurazione dei template da eseguire. Nello specifico, la ST riceve, in una porta che non è nessuna di quelle precedentemente elencate, un messaggio unico che è formato dalle seguenti informazioni:

- la durata dell'intervallo di scansione, seguita dal carattere "/";
- il codice del template, seguito dal carattere "/";
- la frequenza di campionamento, seguita dal carattere "/";
- la lista degli ID dei device a cui era rivolto il messaggio, che ha come separatore tra gli indirizzi il carattere "," e termina con il carattere "@".

La board STM32 riceve il messaggio e salva, nella variabile *interval*, la durata della scansione, che verrà usata per inviare il segnale di END alla Raspberry, nel caso in cui la STM32 non lo riceve dal Gateway entro 20 secondi dalla fine del tempo d'acquisizione. Successivamente, essa estrapola dal messaggio la lista degli ID dei device al quale esso è riferito, e utilizza la funzione *check_uid()* per verificare che il suo ID sia contenuto all'interno di tale lista; se non lo è scarta il messaggio, altrimenti lo inoltra alla Raspberry tramite la funzione *HAL_UART_Trasmit()*, escludendo la prima parte del messaggio che contiene la durata dell'acquisizione, poiché la Raspberry non la utilizzerà.

4.2 Collegamento fisico tra STM32 e Raspberry all'interno dell'End-Node

Come già detto più volte, la modalità di comunicazione scelta per far dialogare STM32 e Raspberry all'interno dell'End-Node è l'UART (Universal Asynchronous Receiver-Transmitter). Esso è un dispositivo hardware, di uso generale o dedicato, che converte flussi di bit di dati da un formato parallelo a un formato seriale asincrono, o viceversa. Questo tipo di trasmissione è solitamente usato per far comunicare dispositivi con piccole distanze tra loro, in maniera seriale.

La comunicazione avviene attraverso l'uso di 3 GPIO per entrambe le schede; essi sono i pin di Trasmissione (Tx) e Ricezione (Rx) per lo scambio dei dati inviati in maniera seriale, e le masse (GND), che devono essere collegate tra loro per far lavorare i due dispositivi con un riferimento comune. In particolare il pin $D2$ (UART1 Rx) della scheda STM32 deve essere collegato con il $GPIO14$ (UART Tx) della Raspberry, il pin $D8$ (UART1 Tx) della scheda STM32 deve essere collegato con il $GPIO15$ (UART Rx) della Raspberry; infine, uno dei pin GND della STM32 deve essere collegato ad uno dei pin GND della Raspberry.

4.3 Scelta del convertitore MCC172

Uno dei componenti fondamentali dell'End-Node è il convertitore Analogico-Digitale MCC172. Esso è un DAQ HAT (Data Acquisition System, Hardware Attached on Top) a 24 bit che effettua misurazioni di suoni e vibrazioni da sensori analogici [10]. L'MCC172 è mostrato in Figura 4.3, collegato a una Raspberry Pi.

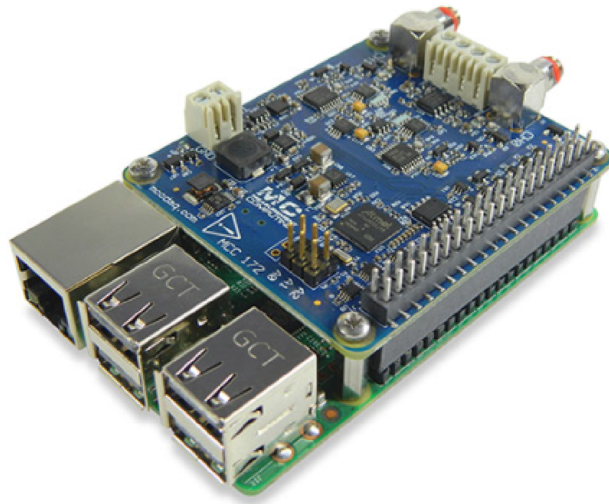


Figura 4.3. Convertitore MCC172 collegato alla Raspberry Pi

La scelta del convertitore dipende dall'ampiezza delle accelerazioni misurate, dalla loro banda di frequenza, dal rumore che si può accettare sulla misura, dalla sensibilità e dalla precisione del campionamento in termini temporali. Le esigenze richieste dal reparto strutturale, che hanno indicato come accettabile un valore di rumore molto ridotto, dell'ordine di qualche μV (microvolt), una sensibilità della misura dell'ordine di decine di μV, e una precisione nel campionamento, in termini temporale, che non abbia un errore superiore a qualche mS (millisecondo), hanno portato alla scelta di questa scheda ad altissima efficienza. Questi parametri sono stati scelti in accordo con il sensore utilizzato, di cui si parlerà nella Sezione 4.5.

Per scheda HAT si intende una board che può essere montata nel lato superiore di un'altra scheda, che, in questo caso, è, appunto, la Raspberry Pi 3B+, attraverso un meccanismo di incastro dei connettori. In particolare, l'header dell'MCC172 si collega a tutti i modelli Raspberry Pi con un connettore a 40 pin.

Su una sola Raspberry possono essere montate fino a 8 HAT, impilate una sopra l'altra, in modo tale da gestire simultaneamente l'acquisizione di tutte le 8 board. In Figura 4.4 è possibile vedere una configurazione formata da 4 HAT.

Figura 4.4. Stack di 4 HAT MCC172

I parametri di configurazione dell'HAT sono memorizzati in una EEPROM integrata a bordo della scheda MCC172, che consente alla Raspberry Pi di configurare automaticamente i pin GPIO quando l'HAT è collegato.

Ogni DAQ MCC172 fornisce due canali di acquisizione analogica differenziali; in altre parole, per ogni canale, vi suono due ingressi per l'acquisizione della tensione analogica, uno per il potenziale positivo e uno per quello negativo, escludendo, quindi, la necessità di avere una tensione riferita a massa. I due canali di ingresso analogico differenziali a 24 bit acquisiscono simultaneamente dati a velocità fino a 51,2 kS/s (con l'unità di misura S si intendono i campioni acquisiti, ovvero Samples). Ogni canale ha un convertitore A/D dedicato. Entrambi gli ADC condividono lo stesso clock, e sono sincronizzati per avviare le conversioni contemporaneamente, ottenendo, così, dati sincroni. Inoltre, è possibile sincronizzare più HAT MCC172 su un singolo clock di campionamento. In questo caso, il clock è programmabile per frequenze di campionamento comprese tra 51,2 kS/s e 200 S/s. È chiaro, quin-

di, che il throughput massimo di acquisizione, per una singola Raspberry, dipende dal numero di HAT che vi sono montate, ma dipende, anche, da limiti fisici della Raspberry e della comunicazione SPI; in particolare, essa sarà pari a:

- 102.4 kS/s (51.2 kS per due canali) nel caso di una singola scheda;
- 307.2 kS/s nel caso di più schede.

È possibile notare che il caso massimo è quello che si raggiungerebbe con solo 3 HAT, seppure se ne possano montare fino ad 8. Ciò è dovuto a limiti computazionali, che dipendono dal carico del processore della Raspberry e dall'interfaccia SPI, usata per la comunicazione tra Raspberry e MCC172. Ciò comporta, quindi, che se si aumentasse il numero di board che formano lo stack sopra la Raspberry ad un valore maggiore di 3, si diminuirebbe la frequenza massima di acquisizione per ogni canale della scheda.

La MCC172 è alimentata a $5V$ dalla Raspberry, attraverso il connettore GPIO sulla quale è montata. I segnali analogici differenziali in input, ovvero quelli da acquisire, possono essere portati alla MCC172 attraverso l'uso di connettori coassiali 10-32, o attraverso l'uso dei terminali a vite.

La scheda MCC172 possiede al proprio interno un microcontrollore in grado di dialogare con la Raspberry grazie al firmware installato al suo interno. È possibile osservare uno schema a blocchi che descrive la scheda nella Figura 4.5.

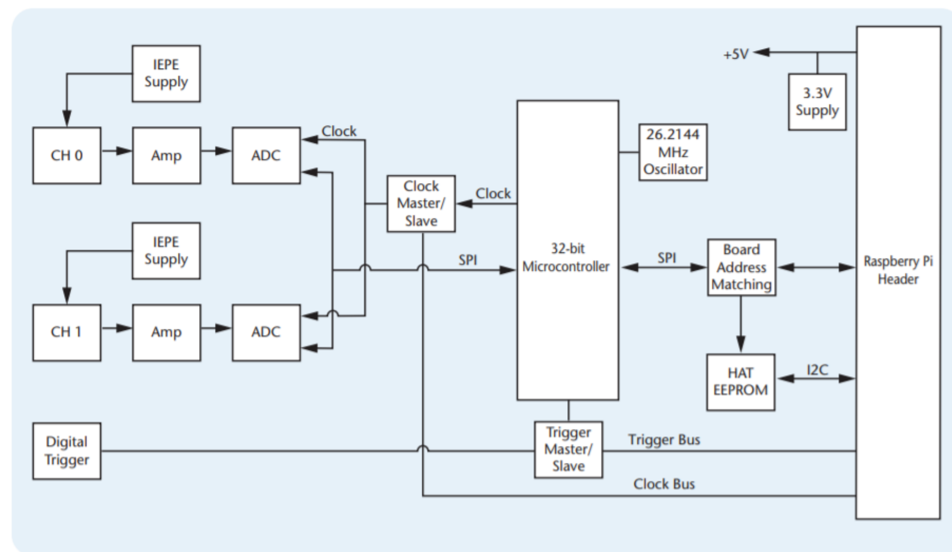


Figura 4.5. Diagramma a blocchi che descrive la HAT MCC172

Come si può vedere dallo schema, vi sono due ADC, uno per canale, che ricevono il clock da una sorgente interna, o attraverso la condivisione, tramite il connettore della Raspberry, del clock di un'altra scheda MCC172 montata nello stack, secondo il rispettivo funzionamento, da Master o Slave. Infatti, i connettori A0, A1 e A2, su cui è possibile montare dei Jumper, sono usati per identificare ogni HAT quando

ci sono più schede montate. La prima HAT, ovvero quella che va direttamente collegata alla Raspberry, deve avere indirizzo 0, e quindi non deve avere Jumper sui connettori. Per tutte le altre board che si montanto sopra, è necessario inserire i Jumper, definendo l'indirizzo di ciascuna scheda. La definizione dell'address avviene interpretando la presenza o meno dei Jumper, come di seguito specificato:

- se è presente si considera un 1 logico su quel connettore, se non è presente si considera, invece, uno 0 logico;
- considerando come bit meno significativo A0, e come bit più significativo A2, si possono determinare 8 indirizzi, che corrispondono ai numeri binari nel range $000 - 111$.

Nel funzionamento con più HAT, solo una può comportarsi da Master, e quindi condividere il proprio clock, mentre tutte le altre devono riceverlo da quest'ultima, e comportarsi quindi da Slave, oppure generarlo internamente senza condividerlo. Via software si può impostare tale caratteristica del clock selezionando le varie schede attraverso l'indirizzo che si imposta con i connettori A0, A1, e A2.

Inoltre, nella MCC172, vi è un microcontrollore a 32 bit, che gestisce la comunicazione tramite SPI con la Raspberry, e si occupa di interpretare i comandi che da essa riceve, e a rispondere con le corrette informazioni.

Vi sono, poi, elementi aggiuntivi che non sono stati utilizzati all'interno della tesi, ovvero:

- il canale dedicato al Trigger, utilizzato per avviare l'acquisizione quando su tale ingresso si verifica una particolare condizione, come, ad esempio, un fronte di salita o discesa di un segnale CMOS o $5V$ TTL;
- la possibilità di attivare sul canale d'acquisizione un'alimentazione IEPE (Integrated Electronics Piezo-Electric).

Prima di iniziare la descrizione delle librerie fornite, è importante definire le specifiche principali della scheda; alcune di esse sono le seguenti:

- ha 2 canali di input differenziali;
- la risoluzione degli ADC è a 24 bit;
- la tipologia di convertitore Analogico-Digitale è la Delta sigma;
- la modalità di campionamento nei vari canali è simultanea;
- il clock può essere generato internamente, o condiviso da un'altra MCC172 che fa parte dello stack di board montate;
- ha frequenze di campionamento per canale che vanno da 200 S/s a 51.2 kS/s;
- la AC cutoff frequency è pari 0.78 Hz, che indica, appunto, che segnali continui in input vengono filtrati e non misurati;
- il range di tensione in input deve essere compreso tra $\pm$ 5 V;
- l'impedenza di input è 202 $k\Omega$ sui canali differenziali;
- i GPIO pin utilizzati della Raspberry sono:
  - $GPIO8$, $GPIO9$, $GPIO10$, GPIO11 per l'interfaccia SPI;
  - ID_SD, ID_SC per l'ID EEPROM;
  - $GPIO12$, $GPIO13$, $GPIO26$ per l'indirizzo della board MCC172;
  - $GPIO5$, $GPIO6$, $GPIO19$, $GPIO16$, $GPIO20$ per la condivisione del clock e del trigger, per il Reset e per le Interrupt Request (IRQ);
- la frequenza di lavoro dell'SPI è quella massima, ovvero 18 MHz,

- le condizioni ambientali di lavoro hanno un range da $-40°C$ a 85 $°C$ per la temperatura, e da 0% a 90% per l'umidità;
- la precisione di misurazione del segnale AC in input ha un errore massimo del guadagno del segnale pari allo 0.43% e un errore di offset massimo di 5.1 mV;
- ha un valore quadratico medio del rumore massimo pari a 33 $\mu Vrms$ ad una frequenza di campionamento di 51.2 kS/s.

Measurament Computing Corporation (MCC), venditore della scheda MCC172, mette a disposizione degli utenti un set di librerie, sviluppate sia in C che in Python, che permette di realizzare applicazioni sulla Raspberry. Quest'ultima contiene il sistema operativo Raspbian, nella sua versione Lite, che è un sistema operativo basato su Linux. Nella versione Lite si hanno alcuni vantaggi prestazionali, poiché vengono eliminati tutti i servizi che si occupano di controllare il desktop.

Nel pacchetto fornito, oltre alle librerie, vi sono anche applicazioni ed esempi per l'acquisizione in varie modalità (ad esempio, continua o con l'utilizzo del trigger), che comporta l'utilizzo di una o più HAT.

Il progetto, all'interno dell'End-Node, è stato sviluppato interamente in C; per questo, nella prossima sezione (Sezione 4.4) verranno illustrate solo le librerie in questo linguaggio.

4.4 Raspberry Pi 3B+ e MCC172

Per comprendere il codice che si occupa dell'acquisizione dei segnali attraverso l'uso della MCC172, è indispensabile analizzare dapprima la libreria fornita da MCC. La libreria *daqhats* supporta diverse versioni di board MCC DAQ HAT, tra cui si trovano le seguenti:

- MCC118;
- MCC128;
- MCC134;
- MCC152;
- MCC172;

Il file principale della libreria, per l'utilizzo della scheda MCC172 si trova all'interno della cartella `/daqhats/lib`, e si chiama `mcc172.c`.

Al suo interno sono definite tutte le funzioni che permettono l'utilizzo della board; queste vanno dall'inizializzazione delle strutture dati usate per raccogliere le informazioni dell'acquisizione fino al cleanup della scheda.

Nella Figura 4.6 sono riassunte le funzioni principali che si trovano all'interno del file prima citato.

Tra quelle elencate, ve ne sono alcune di particolare interesse, e fondamentali da capire per lo sviluppo del codice:

- *mcc172_open()*: è la prima funzione che viene richiamata; essa apre la connessione con l'opportuna scheda; questa viene indicata attraverso il parametro *adrress* passato. La funzione alloca in maniera dinamica la struttura dati nella quale verranno memorizzati tutti i parametri dell'acquisizione per ciascuna board.

Function	Description
`mcc172_open()`	Open an MCC 172 for use.
`mcc172_is_open()`	Check if an MCC 172 is open.
`mcc172_close()`	Close an MCC 172.
`mcc172_info()`	Return information about this device type.
`mcc172_blink_led()`	Blink the MCC 172 LED.
`mcc172_firmware_version()`	Get the firmware version.
`mcc172_serial()`	Read the serial number.
`mcc172_calibration_date()`	Read the calibration date.
`mcc172_calibration_coefficient_read()`	Read the calibration coefficients for a channel.
`mcc172_calibration_coefficient_write()`	Write the calibration coefficients for a channel.
`mcc172_iepe_config_read()`	Read the IEPE configuration for a channel.
`mcc172_iepe_config_write()`	Write the IEPE configuration for a channel.
`mcc172_a_in_sensitivity_read()`	Read the sensitivity scaling for a channel.
`mcc172_a_in_sensitivity_write()`	Write the sensitivity scaling for a channel.
`mcc172_a_in_clock_config_read()`	Read the sampling clock configuration.
`mcc172_a_in_clock_config_write()`	Write the sampling clock configuration.
`mcc172_trigger_config()`	Configure the external trigger input.
`mcc172_a_in_scan_start()`	Start a hardware-paced analog input scan.
`mcc172_a_in_scan_buffer_size()`	Read the size of the internal scan data buffer.
`mcc172_a_in_scan_status()`	Read the scan status.
`mcc172_a_in_scan_read()`	Read scan data and status.
`mcc172_a_in_scan_channel_count()`	Get the number of channels in the current scan.
`mcc172_a_in_scan_stop()`	Stop the scan.
`mcc172_a_in_scan_cleanup()`	Free scan resources.

Figura 4.6. Elenco delle funzioni principali della libreria mcc172.c

- *mcc172_iepe_config_write()*: è la funzione che si occupa di scrivere la configurazione IEPE per un singolo canale; essa, quindi, deve essere richiamata per tutte le schede e per ogni canale, con il parametro *config* pari a 0 per disabilitarlo e ad 1 per attivarlo (come già detto, è stata disattivata questa opzione per ogni canale).
- *mcc172_a_in_clock_config_write()*: è la funzione che si occupa di scrivere sulla MCC172 la configurazione del clock di campionamento. In particolare, essa imposta i due parametri fondamentali del clock che pilota gli ADC, ovvero la sorgente del clock e la sua frequenza. Il primo parametro, definito *clock_source* può assumere uno dei seguenti valori:
  - *SOURCE_LOCAL*, che indica che il clock è generato su questa MCC172 e non è condiviso con altre MCC172;
  - *SOURCE_MASTER*, che indica che il clock è generato su questa MCC172 ed è condiviso come master clock per le altre MCC172; tutte le altre MCC172 devono essere configurati come *SOURCE_LOCAL* o *SOURCE_SLAVE*;
  - *SOURCE_SLAVE*, che indica che nessun clock viene generato su questa MCC172; in questo caso il clock viene ricevuto dal Master.

  Gli ADC verranno sincronizzati in modo da campionare gli ingressi contemporaneamente. Ciò richiede 128 cicli di clock prima che il primo campione sia disponibile. Inoltre, quando si utilizza una configurazione di clock Master - Slave con più schede ci sono ulteriori considerazioni da fare. In particolare:
  - È necessario configurare prima il clock sui dispositivi Slave, e successivamente nel Master. La sincronizzazione avverrà quando il clock del Master è configurato, facendo sì che gli ADC su tutti i dispositivi siano sincronizzati.
  - Se si modifica la configurazione del clock su un dispositivo dopo aver configurato il Master, i dati non saranno più sincronizzati. I dispositivi non possono rilevarlo e segnaleranno, comunque, che sono sincronizzati. Bisogna scrivere sempre la configurazione del clock su tutti i dispositivi quando si modifica la configurazione.
  - I dispositivi Slave devono avere una sorgente di clock Master, altrimenti le scansioni non verranno mai eseguite.

  La MCC172 può generare un clock di campionamento ADC pari a 51,2 kHz diviso per un numero intero compreso tra 1 e 256. La frequenza per ciascun canale sarà convertita internamente alla frequenza valida più vicina al valore che viene imposto.
- *mcc172_a_in_clock_config_read()*: è la funzione che misura la velocità effettiva del clock su una MCC172. Se utilizzato per una scheda Slave, il dispositivo misurerà la frequenza del clock del Master in ingresso al termine del periodo di sincronizzazione.
- *mcc172_a_in_scan_start()*: è la funzione che si occupa di far avviare l'acquisizione sui canali indicati. Vi sono diverse opzioni disponibili con le quali avviare la scansione; esse sono:
  - *OPTS_NOSCALEDATA*, che restituisce il codice ADC (un valore compreso tra *AI_MIN_CODE* e *AI_MAX_CODE*) anziché la tensione;
  - *OPTS_NOCALIBRATEDATA*, che restituisce i dati senza i fattori di calibrazione applicati;

  - *OPTS_EXTTRIGGER*, che sospende la scansione (dopo aver chiamato *mcc172_a_in_scan_start()*) fino a quando non viene soddisfatta la condizione di trigger;
  - *OPTS_CONTINUOUS*, che esegue la scansione in maniera continua fino a quando essa non viene interrotta dall'utente chiamando la funzione *mcc172_a_in_scan_stop()*; successivamente essa scrive i dati in un buffer circolare.
  - *OPTS_DEAFAULT*, che indica che sono dati scalati e calibrati, non viene usato nessun trigger e viene acquisito un numero di campioni prefissato.
- *mcc172_a_in_scan_stop()*: è la funzione che invia il comando, tramite SPI, alla MCC172, per fermare l'acquisizione dei campioni in ingresso.
- *mcc172_close()*: è la funzione che si occupa di chiudere la connessione con la HAT MCC172 e di liberare tutte le risorse di memoria allocate.

Ora, si può passare alla descrizione del codice sviluppato sulla Raspberry presente negli End-Node. Il progetto si sviluppa su più file, che si trovano all'interno della directory `home/pi/progetto-finale/C_Project-End_Node/obj`; tali file sono:

- `daqhats_utils.c` e `daqhats_utils.h`, che contengono alcune funzioni di interfacciamento, utilizzabili per tutta la serie MCC, e per l'inclusione delle varie librerie sviluppate su ciascuna board, tra cui la `mcc172.h`;
- `DEV_Config.c` e `DEV_Config.h`, che sono utilizzati per la lettura e scrittura del file `config.cfg`, che contiene tutte le informazioni per la configurazione del progetto da eseguire, come codice progetto e dispositivo;
- `Measurement.c` e `Measurement.h`, in cui vi sono tutte le procedure per l'inizializzazione della MCC172 e per l'acquisizione e il salvataggio dei dati;
- `UART_thread.c` e `UART_thread.h`, nei quali vi è sviluppata un'unica funzione per rimanere in ascolto sull'UART e, successivamente, elaborare il messaggio letto;
- `main.c`, che è il main principale del programma e si occupa di attivare due thread, uno per l'acquisizione dei dati e uno per la lettura dei messaggi sull'UART.

Come già detto il file principale del programma è `main.c` . La funzione *main()* è sviluppata come mostrato nel Listato 4.5.

```
int main(void)
{
    load_configs(progetto, ids.rasp_id, ids.ll_id);
    pthread_t thread_id;

    /*Start threads*/
    pthread_create(&thread_id, NULL, Measurement, NULL);
    pthread_create(&thread_id, NULL, UART_thread, NULL);

    //pthread_create(&thread_id, NULL, Persistor, NULL);
    pthread_join(thread_id, NULL);

    return 0;
}
```

Listato 4.5. Implementazione del *main()* in esecuzione nella Raspberry presente nell'End-Node

Inizialmente vengono caricati i dati di default di configurazione del progetto, attraverso la funzione *load_conf()*; successivamente vengono creati due thread, ov-

vero *Measurement* e *UART_thread*, che si occupano, rispettivamente, di avviare l'acquisizione dei dati accelerometrici e di rimanere in ascolto, sul canale UART, dei messaggi che arrivano dalla STM32. I due thread vengono implementati all'interno dei file `Measurement.c` e `UART_thread.c`.

L'implementazione del thread *Measurement* è visibile nel Listato 4.6.

```
void *Measurement(void *vargp)
{
    while(1)
    {

        // initialization of MCC172
        Init(scan_rate_init);

        // create mutex for save beacon
        init_beacon_mutex();

        beacon_count = 0;

        // start the acquisition
        Check_Acquisition();

        // destroy mutex for save beacon
        destroy_beacon_mutex();

    }
}
```

Listato 4.6. Implementazione del thread *Measurement* in esecuzione nella Raspberry presente nell'End-Node

All'interno del thread viene inizialmente attivata la procedura di inizializzazione della MCC172, che richiama le funzioni *mcc172_open()*, *mcc172_a_in_clock_config-_write()*, *mcc172_a_in_clock_config_write()* e *mcc172_a_in_clock_config_read()*, come si può osservare dal Listato 4.7.

```
void Init(double scan_rate)
{
    int max_channel_array_length = mcc172_info()->NUM_AI_CHANNELS;
    int chans[DEVICE_COUNT][max_channel_array_length];

     // Determine the addresses of the devices to be used
    if (get_hat_addresses(address) != 0)
    {
        exit(0);
    }

    for (device = 0; device < DEVICE_COUNT; device++)
    {
        // Open a connection to each device
        result = mcc172_open(address[device]);
        error(result);

        // channel_mask is used by the library function mcc172_a_in_scan_start.
        // The functions below parse the mask for display purposes.
        chan_count[device] = convert_chan_mask_to_array(chan_mask[device],
            chans[device]);
        convert_chan_mask_to_string(chan_mask[device], chan_display[device]);

        for (i = 0; i < chan_count[device]; i++)
        {
            result = mcc172_iepe_config_write(address[device], chans[device][i],0);
            error(result);
        }

        // Configure the clock (slaves only)
        if (device != MASTER)
        {
            result = mcc172_a_in_clock_config_write(address[device],
                SOURCE_SLAVE, scan_rate);
            error(result);
        }
    }

    // Configure the master clock last so the clocks are synchronized
    result = mcc172_a_in_clock_config_write(address[MASTER],
        SOURCE_MASTER, scan_rate);
    error(result);
```

```
    // Wait for the ADCs to synchronize.
    do
    {
        result = mcc172_a_in_clock_config_read(address[MASTER], &clock_source,
            &actual_scan_rate, &synced);
        error(result);
        usleep(5000);
    } while (synced == 0);

    printf("   Requested scan rate: %-10.2f\n", scan_rate);
    printf("   Actual scan rate: %-10.2f\n", actual_scan_rate);
    for (device = 0; device < DEVICE_COUNT; device++)
    {
        printf("   MCC 172 %d:\n", device);
        printf("     Address: %d\n", address[device]);
        printf("     Channels: %s\n", chan_display[device]);
    }
}
```

Listato 4.7. Implementazione della funzione *Init()* presente nel file `Measurement.c`

Il parametro che viene fornito alla funzione, ovvero *scan_rate*, è il valore che viene utilizzato per definire la frequenza di campionamento degli ADC. Sempre all'interno della funzione *Init*, viene disattivata l'alimentazione IEPE e viene assegnata, come sorgente del clock, l'opzione *SOURCE_MASTER* alla HAT montata direttamente sulla Raspberry (address 0), e l'opzione *SOURCE_SLAVE* all'altra board (address 1). Come già fatto osservare precedentemente, è necessario prima impostare le configurazioni di tutti gli Slave e, solo successivamente, quella del Master. Inoltre, non si esce dalla fase di inizializzazione finché la funzione *mcc172_a_in_clock_config_read()* non restituisce il parametro *synced* uguale a 1, che indica, appunto, che gli ADC sono sincronizzati e si può iniziare l'acquisizione.

Una volta finita la fase di inizializzazione, nel thread *Measurement* del Listato 4.6, viene attivata la funzione *Check_Acquisition()*, che è quella che si occupa dell'acquisizione dei campioni. Essa è descritta nel Listato 4.8.

```
void Check_Acquisition(void)
{
    //MEASURE UNTIL THE ARRIVAL OF THE  TRIGGER
    printf("Misurando in attesa del trigger...\n");
    // Start the scans
    for (device = 0; device < DEVICE_COUNT; device++)
    {
        result = mcc172_a_in_scan_start(address[device], chan_mask[device],
            samples_per_channel1, options1);
        error(result);
    }
    //_scan_thread is kept active until the trigger signal arrives
    while (trigger_event_flag == 0);
    for (device = 0; device < DEVICE_COUNT; device++)
    {
        print_error(mcc172_a_in_scan_stop(address[device]));
        print_error(mcc172_a_in_scan_cleanup(address[device]));
    }
    //save data PRE-TRIGGER
    print_error(mcc172_save_data_pre_t(address[0],address[1]));
    for (device = 0; device < DEVICE_COUNT; device++)
    {
        print_error(mcc172_close(address[device]));
    }

    //ON THE ARRIVAL OF THE TRIGGER THE MEASUREMENT CONTINUES UNTIL THE ARRIVAL OF THE END
    if (trigger_event_flag == 1)
    {
        //if acquisition is scheduled by user, change samplerate to the one received
        if (acquisition_type == 0)
        {
            Init(sr);
        }
        else
        {
            Init(scan_rate_init);
        }

        printf("Trigger arrivato!\n");
```

```
            trigger_event_flag = 0;
            beacon_event_flag = 1;
            // Start the scans
            for (device = 0; device < DEVICE_COUNT; device++)
            {
                   result = mcc172_a_in_scan_start(address[device], chan_mask[device],
                          samples_per_channel2, options2);
                   error(result);
            }

            //_scan_thread is kept active until the end signal arrives
            while (end_event_flag == 0);


            end_event_flag = 0;
            trigger_event_flag = 0;
            printf("End arrivato!\n");
            for (device = 0; device < DEVICE_COUNT; device++)
            {
                   print_error(mcc172_a_in_scan_stop(address[device]));
                   print_error(mcc172_a_in_scan_cleanup(address[device]));
            }

            //save data POST TRIGGER
            print_error(mcc172_save_data_finite(address[0],address[1]));

            for (device = 0; device < DEVICE_COUNT; device++)
            {
                   print_error(mcc172_close(address[device]));
            }
        }
    }
```

Listato 4.8. Implementazione della funzione *Check_Acquisition()* presente nel file `Measurement.c`

Nella funzione *Check_Acquisition()* viene avviata l'acquisizione dei campioni nelle due board, in particolare in entrambi i canali della HAT 0 e solo nel canale 0 della HAT 1. La funzione *mcc172_a_in_scan_start()* viene attivata due volte, una per ogni HAT. Inoltre, tale funzione avvia il thread *_scan_thread()*, definito all'interno della libreria `mcc172.c`, che memorizza i campioni acquisiti all'interno di un buffer circolare di 20000 elementi, andando, così, a memorizzare costantemente gli ultimi 20 secondi di dati acquisiti, poiché la frequenza di campionamento di default è di 1000 Hz. La funzione *_scan_thread()* è stata notevolmente modificata dalla sua versione originale, ma ciò verrà discusso nel Capitolo 5, dove verranno descritte le varie fasi di progettazione. L'acquisizione dei campioni continuerà fino all'arrivo del segnale di TRIGGER sul canale UART. A questo punto verrà fermata l'acquisizione; successivamente verranno salvati gli ultimi 20 secondi di dati acquisiti all'interno del vettore *dati_salvati*, grazie alla funzione *mcc172_save_data_pre_t()*, anch'essa aggiunta alla libreria `mcc172.c` in fase di progettazione.

Terminato il salvataggio, verrà avviata una nuova scansione. Poiché, per ogni scansione effettuata è necessario chiudere il collegamento con le schede MCC172 invocando la funzione *mcc172_close()*, è necessario avviare una nuova fase di inizializzazione con la funzione *Init()*. Questa potrà avvenire in due modi diversi, ovvero con la frequenza di campionamento di default di 1000 Hz, o con una frequenza impostata dall'utente, fornita via UART dalla STM32. Viene, così, avviata una nuova scansione finché non arriva il segnale di END; per ogni BEACON ricevuto, viene memorizzato il valore del conteggio dei BEACON ricevuti sul vettore *time*, all'interno del *_scan_thread()*, invocato dalla funzione *mcc172_a_in_scan_start()*. Anche in questo caso, una volta conclusa l'acquisizione, verranno salvati i dati, richiamando però la funzione *mcc172_save_data_post_t()*. Essa si occupa della creazione del file `.csv` che dovrà essere caricato sull'FTP server. Il file conterrà, al proprio interno, sia i dati relativi ai 20 secondi di acquisizione precedenti all'arrivo del TRIGGER,

salvati temporaneamente nel vettore *dati_salvati*, sia quelli relativi all'acquisizione appena terminata. Per permettere il caricamento nella giusta cartella del server e avere a disposizione l'esatto istante dell'inizio dell'acquisizione, il nome del file `.csv` sarà strutturato nel seguente modo:

- carattere "_", seguito dal codice del progetto e dal carattere "-";
- codice del template, seguito dal carattere "!";
- ID della Raspberry, seguito dal carattere "&";
- timestamp nel formato *Anno-Mese-Giorno*-"T"*Ora-Minuti-Secondi*"Z".

Infine verrà di nuovo chiusa la comunicazione con le board MCC172, per poi avviare, dal thread *Measurement*, una nuova scansione in attesa del TRIGGER.

Invece, l'implementazione del thread *UART_thread* è stata svolta all'interno del file `UART_thread.c`. L'intera gestione ed elaborazione dei messaggi, che vengono ricevuti sul canale UART, viene eseguita all'interno di questo thread; il suo codice è mostrato nel Listato 4.9.

```
void *UART_thread (void *vargp)
{
        //UART INIT//
        int serial_port ;
        char dat;
        int length;
        if ((serial_port = serialOpen ("/dev/ttyAMA0", 115200)) < 0) /* open serial port */
        {
            fprintf (stderr, "Unable to open serial device: %s\n", strerror (errno)) ;
            return NULL ;
        }

        if (wiringPiSetup () == -1) /* initializes wiringPi setup */
        {
                fprintf (stdout, "Unable to start wiringPi: %s\n", strerror (errno)) ;
                return NULL ;
        }
        else
        {
                printf("UART correctly initialized! \n");
        }
        //UART READ THREAD LOOP//
        while(1)
        {
                length = serialDataAvail(serial_port);
                if (length > -1)
                {
                        dat = serialGetchar(serial_port);
                        // the value 255 corresponds to the character -1, which indicates that 10 seconds have passed
                                with no message being read
                        if (dat != 255)
                        {
                                printf("%c", dat);
                                if (receive_code==1)
                                {
                                        device_active = true;
                                        code[i] = dat;
                                        if (code[i] == '@' || code[i] == '?')
                                        {
                                                //the message containing the template_code, sample_rate, device_list
                                                        ends with the "@" character
                                                if (code[i] == '@')
                                                {
                                                        code[i] = '\0';
                                                        printf("Informazioni template in esecuzione: %s\n", code);
                                                        sprintf(template_info, code);
                                                        // the 3 informations are divided by the character "/"
                                                        char *pointer = strtok(template_info, delim);
                                                        sprintf(template, pointer);
                                                        pointer = strtok(NULL, delim);
                                                        sprintf(sample_rate, pointer);
                                                        sr = atof(sample_rate);
                                                        pointer = strtok(NULL, delim1);
                                                        //if not NULL, the devices that must be activated are read
                                                        printf("Device attivi: ");
                                                        if (pointer == NULL)
                                                        {
                                                                printf("All \n");
                                                        }
                                                        while (pointer != NULL)
                                                        {
                                                                sprintf(devices[j], pointer);
                                                                printf("\n%s\n", devices[j]);
```

```
                                pointer = strtok(NULL, delim1);
                                j++;
                        }
                        printf("ID Device: %s\n", ids.ll_id);
                        //check that ll_id is in the device list
                        int match = 0;
                        if (j == 0)
                        {
                                match = 1;
                        }
                        else
                        {
                                for (int k = 0; k<j; k++)
                                {
                                        if (!strcmp(ids.ll_id, devices[k]))
                                        {
                                                match=1;
                                        }
                                }
                        }
                        if (match == 0)
                        {
                                device_active = false;
                                printf("\nDevice non attivo!\n");
                        }
                        else
                        {
                                device_active = true;
                                printf("\nDevice attivo!\n");
                        }
                        i = 0;
                        j = 0;
                        receive_code=0;
                        receive_synch=1;
                        acquisition_type = 0;
                }
                else
                {
                        //the message containing the project_code, rasp_id, ll_id ends
                                with the "?" character
                        code[i] = '\0';
                        printf("Salvo codice progetto e UID\n");
                        sprintf(progetto, code);
                        char *pointer = strtok(progetto, delim);
                        sprintf(proj_code, pointer);
                        //
                        pointer = strtok(NULL, delim);
                        sprintf(ids.rasp_id, pointer);
                        pointer = strtok(NULL, delim);
                        sprintf(ids.ll_id, pointer);
                        i = 0;
                        update_configs(proj_code, ids.rasp_id, ids.ll_id);
                        printf("%s\n", proj_code);
                        printf("%s\n", ids.rasp_id); //User chosen ID for ftp upload
                        printf("%s\n", ids.ll_id); //ID in lorawan message
                }
        }
        else
        {
                i++;
        }
        fflush (stdout);
}

//Once the initial message has been received, it waits for the TRIGGER, BEACON, END

if (receive_synch == 1 && dat != '@')
{
        printf("\nReceiveing Synch\n");
        printf("letto %s\n", &dat);
        if ((char)dat == trigger && device_active==true)
        {
                printf("Trigger arrivato! \n");
                trigger_event_flag=1;
                // save the start acquisition timestamp to name the csv file
                time_t now = time(NULL);
                tm = gmtime(&now);
        }
        if ((char)dat == beacon && device_active==true)
        {
                printf("Beacon arrivato! \n");
                beacon_event_flag=1;
        }
        if ((char)dat == end)
        {
                printf("End arrivato! \n");
                end_event_flag=1;
                if (device_active==false)
                {
                        end_event_flag=0;
                }
                receive_synch = 0;
                receive_code = 1;
        }
```

```
                                }

                        }
                        fflush (stdout);
                }
        }
}
```

Listato 4.9. Implementazione del thread *UART_thread()* presente nel file `UART_thread.c` .

Il codice è sviluppato in due grandi blocchi; nella prima parte si ricevono ed elaborano i messaggi che riguardano le informazioni relative alla configurazione del template (*receive_code*=1), mentre nella seconda parte vengono ricevuti i messaggi di sincronizzazione di TRIGGER, BEACON ed END (*receive_synch*=1).

Prima di avviare una scansione, la Raspberry riceve dalla board STM32 due blocchi di messaggi. Il primo è quello che viene inviato dalla STM32 nel caso di ricezione di un messaggio LoRa sulla porta 5. In particolare, il messaggio che viene ricevuto contiene il codice del progetto (*proj_code*), l'ID della Raspberry (*ids.rasp_id*) e il low level ID (*ids.ll_id*), separati tra di loro dal carattere "/" (la variabile *delim* nel Listato 4.9 è valorizzata con questo carattere), e terminante con i caratteri "/" e "?". I tre parametri letti vengono, poi, usati per aggiornare il file `config.cfg` attraverso la funzione *update_config()*.

Il secondo blocco di messaggi che riceve la Raspberry è quello che contiene il codice del template, la frequenza di campionamento e la lista dei device a cui è rivolto il messaggio con le informazioni dell'acquisizione. Anche in questo caso, le tre informazioni sono separate tra di loro con il carattere "/". Inoltre, ciascun elemento della lista degli ID dei device è separato dagli altri dal carattere "," (la variabile *delim1* nel Listato 4.9 è valorizzata con questo carattere), e termina con il carattere "@". Anche in questo caso le tre informazioni vengono estrapolate e memorizzate all'interno delle 3 variabili *template*, *sample_rate* e *devices[]*. Infine viene verificato che il low level ID ricevuto sia contenuto nella lista dei dispositivi e, in caso affermativo, viene settato il flag *device_active*, che permette l'elaborazione dei dati di sincronizzazione.

Una volta terminati i messaggi di configurazione, sul canale UART vengono inviati i tre messaggi di sincronizzazione. In particolare, il TRIGGER è identificato dal carattere "*A*", il BEACON da "*B*" e l'END da "*C*". All'arrivo del TRIGGER viene salvato anche il timestamp, che contiene data e ora, per rinominare il `.csv` con i dati dell'acquisizione. Inoltre, all'arrivo di ciascun segnale, viene settato il corrispettivo flag, che verrà utilizzato, nella funzione *Check_Acquisition()* prima descritta, per iniziare l'acquisizione, inserire il conteggio dei BEACON e fermare la scansione.

4.5 L'accelerometro ADXL354

L'accelerometro scelto per l'acquisizione delle vibrazioni è l'ADXL354, rappresentato in Figura 4.7.

Esso è un dispositivo basato sulla tecnologia MEMS (Micro Electro-Mechanical Systems), che restituisce in uscita tre segnali analogici che corrispondono alle accelerazioni sui tre assi del sensore [1], come mostato in Figura 4.8.

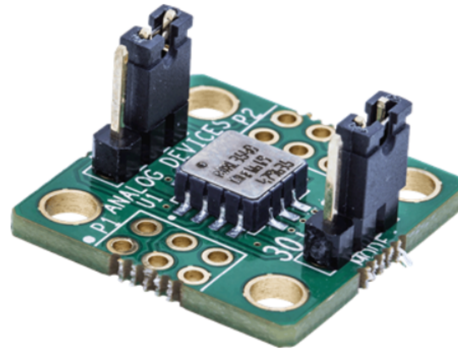


Figura 4.7. Accelerometro ADXL354

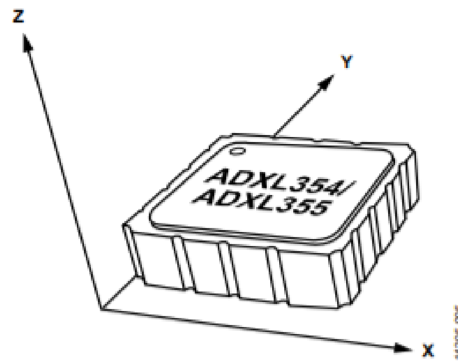


Figura 4.8. Verso degli assi di misurazione dell'accelerometro ADXL354

Su di esso sono montati due header, ovvero P1 e P2, e si hanno, anche, due connettori nei quali è possibile inserire dei Jumper, per impostare la modalità (measurement o standby), e il range di accelerazioni misurabili; quest'ultimo può essere $\pm 4g$ o $\pm 2g$, dove con g è indicata l'accelerazione di gravità. Per i primi due, è possibile vedere il loro collegamento nelle Figure 4.9 e 4.10 .

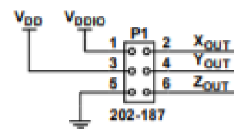


Figura 4.9. Diagramma funzionale dell'header P1

L'header P1 deve essere completamente collegato, connettendo i pin 1 e 3 alla tensione di alimentazione di 3.3 V, il pin 5 a massa, e i pin 2, 4 e 6 agli ingressi della board MCC172. Dell'header P2 è, invece, sufficiente collegare il pin 6 a massa, poiché gli altri pin forniscono funzionalità che non sono state utilizzate.

Le caratteristiche principali di questo componente sono di seguito elencate:

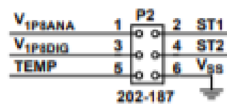


Figura 4.10. Diagramma funzionale dell'header P2

- la corrente di alimentazione assorbita in modalità di misurazione è di circa $150\mu A$, mentre in standby è di 21 μA;
- l'errore, in base alla temperatura, dell'accelerazione misurata è di 0.15 $mg/°C$;
- la densità del rumore su ogni asse è di 22.5 $\mu g/\sqrt{Hz}$;
- ha 3 uscite analogiche, con tensione proporzionale all'accelerazione misurata;
- ha un sensore di temperatura integrato;
- il range di temperatura di operatività va da -50°C a + 125°C;
- supporta accelerazioni nel range $\pm 4g$ o $\pm 2g$.

La scelta è ricaduta su questo sensore poiché l'ADXL354 è affetto da una quantità di rumore minima, tra le migliori sul mercato, e ha una deriva minima dell'offset sulla temperatura e una stabilità a lungo termine che consentono applicazioni di precisione con una calibrazione minima. Tutto ciò lo rende ideale per applicazioni dell'IoT per la misura di vibrazioni.

5

Progettazione della componente software dell'End-Node

In questo capitolo verranno discussi i passi svolti durante la progettazione del software finale dell'End-Node, ed in particolare quello all'interno della Raspberry. Infatti essa è stata la parte principale in cui il lavoro di tirocinio si è incentrato, ed è stato necessario apportare significative modifiche ai driver della scheda MCC172 forniti da MCC, poiché, con tali implementazioni delle funzioni utilizzate, non era possibile ottenere i risultati desiderati. Inoltre, prima della modifica dei driver, è stata effettuata l'installazione della patch Real Time del sistema operativo Raspbian Lite, in esecuzione all'interno della Raspberry. Infine, verranno mostrati i risultati di alcuni test svolti nei vari passi di progettazione, per capire appieno i miglioramenti fatti.

5.1 Analisi della libreria `mcc172.c`

Lo sviluppo del software è partito dall'analisi della libreria `mcc172.c`, di cui sono già state elencate e descritte le funzioni principali nella Sezione 4.4 . Ora, però, è interessante analizzare come avvengono la lettura e l'acquisizione dei campioni, tramite l'analisi dettagliata delle funzioni presenti nella libreria.

Tralasciando le funzioni di inizializzazione, che non sono state modificate dalla loro versione originale fornita da MCC, le funzioni più importanti che dovranno essere analizzate sono:

- *mcc172_a_in_scan_start()*;
- *_scan_thread()*;
- *mcc172_a_in_scan_read()*.

Queste rappresentano le funzioni che si occupano della lettura e della restituzione dei campioni all'utente.

In particolare, nella Sezione 4.4 non è stata descritta la funzione *mcc172_a_in_scan_read()*, poiché essa non è stata inserita nel codice finale. In realtà, negli esempi forniti da MCC e nella loro documentazione, questa funzione risulta indispensabile per la lettura dei campioni. Infatti, inizialmente, come verrà spiegato in

seguito, questa funzione è stata utilizzata; successivamente, per migliorare le prestazioni della Raspebrry, è stata tolta, perché non più necessaria dopo lo modifiche fatte alle altre due funzioni appena elencate.

La prima funzione da analizzare è, sicuramente, *mcc172_a_in_scan_start()*, responsabile dell'avvio dell'acquisizione. Essa è mostrata nel Listato 5.1.

```
int mcc172_a_in_scan_start(uint8_t address, uint8_t channel_mask,
    uint32_t samples_per_channel, uint32_t options)
{
    int result;
    uint8_t num_channels;
    uint8_t channel;
    double sample_rate_per_channel;
    struct mcc172Device* dev;
    struct mcc172ScanThreadInfo* info;
    uint8_t buffer[10];
    uint32_t scan_count;
    uint8_t clock_source;
    uint8_t synced;

    if (!_check_addr(address) ||
        (channel_mask == 0) ||
        (channel_mask >= (1 << NUM_CHANNELS)) ||
        (options & OPTS_EXTCLOCK) ||
        ((samples_per_channel == 0) && ((options & OPTS_CONTINUOUS) == 0)))
    {
        return RESULT_BAD_PARAMETER;
    }

    dev = _devices[address];

    if (dev->scan_info != NULL)
    {
        // scan already running?
        return RESULT_BUSY;
    }

    dev->scan_info = (struct mcc172ScanThreadInfo*)calloc(
        sizeof(struct mcc172ScanThreadInfo), 1);
    if (dev->scan_info == NULL)
    {
        return RESULT_RESOURCE_UNAVAIL;
    }

    info = dev->scan_info;
    info->options = (uint16_t)options;

    num_channels = 0;
    for (channel = 0; channel < NUM_CHANNELS; channel++)
    {
        if (channel_mask & (1 << channel))
        {
            // save the channel list and coefficients for calibrating the
            // incoming data
            info->channels[num_channels] = channel;
            info->slopes[num_channels] = dev->factory_data.slopes[channel];
            info->offsets[num_channels] = dev->factory_data.offsets[channel];

            // set the scaling factor using the LSB size and sensitivity
            info->scale_factors[num_channels] = LSB_SIZE /
                dev->sensitivities[channel];

            num_channels++;
        }
    }
    info->channel_count = num_channels;
    info->channel_index = 0;

    // Read the clock config, wait until in sync
    int count = 0;
    do
    {
        result = mcc172_a_in_clock_config_read(address, &clock_source,
            &sample_rate_per_channel, &synced);
        if (result != RESULT_SUCCESS)
        {
            free(info);
            dev->scan_info = NULL;
            return result;
        }

        if (synced == 0)
        {
            usleep(1000);
        }
        count++;
    } while (synced == 0);
```

```
// Calculate the buffer size
if (options & OPTS_CONTINUOUS)
{
    // Continuous scan - buffer size is set to the (samples_per_channel
    // * number of channels) unless that value is less than:
    //
    // Rate        Buffer size
    // ----        -----------
    // < 1024 S/s  1 kS per channel
    // < 10.24 kS/s 10 kS per channel
    // < 100 kS/s  100 kS per channel

    if (sample_rate_per_channel <= 1024.0)
    {
        info->buffer_size = 1000;
    }
    else if (sample_rate_per_channel <= 10240.0)
    {
        info->buffer_size = 10000;
    }
    else
    {
        info->buffer_size = 100000;
    }

    if (info->buffer_size < samples_per_channel)
    {
        info->buffer_size = samples_per_channel;
    }
}
else
{
    // Finite scan - buffer size is the number of channels *
    // samples_per_channel,
    info->buffer_size = samples_per_channel;
}

info->buffer_size *= num_channels;

// allocate the buffer
info->scan_buffer = (double*)calloc(1, info->buffer_size * sizeof(double));
if (info->scan_buffer == NULL)
{
    // can't allocate memory
    free(info);
    dev->scan_info = NULL;
    return RESULT_RESOURCE_UNAVAIL;
}

// Set the device read threshold based on the scan rate - read data
// every 100ms or faster.
info->read_threshold = (uint16_t)(sample_rate_per_channel / 10);
if (info->read_threshold > MAX_SAMPLES_READ)
{
    info->read_threshold = MAX_SAMPLES_READ;
};
info->read_threshold = COUNT_NORMALIZE(info->read_threshold,
    info->channel_count);
if (info->read_threshold == 0)
{
    info->read_threshold = info->channel_count;
}

pthread_attr_t attr;
if ((result = pthread_attr_init(&attr)) != 0)
{
    free(info->scan_buffer);
    free(info);
    dev->scan_info = NULL;
    return RESULT_RESOURCE_UNAVAIL;
}

pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

// Start the scan
if (options & OPTS_EXTTRIGGER)
{
    // enable the trigger
    channel_mask |= 0x04;
}

if (options & OPTS_CONTINUOUS)
{
    // set to 0 for continuous
    scan_count = 0;
}
else
{
    scan_count = samples_per_channel;
}


buffer[0] = (uint8_t)scan_count;
```

```
    buffer[1] = (uint8_t)(scan_count >> 8);
    buffer[2] = (uint8_t)(scan_count >> 16);
    buffer[3] = (uint8_t)(scan_count >> 24);
    buffer[4] = channel_mask;
    buffer[5] = 0;

    result = _spi_transfer(address, CMD_AINSCANSTART, buffer, 6, NULL, 0,
        20*MSEC, 10);

    if (result != RESULT_SUCCESS)
    {
        pthread_attr_destroy(&attr);
        free(info->scan_buffer);
        free(info);
        dev->scan_info = NULL;
        return result;
    }

    info->thread_started = false;

    // create the scan data thread
    uint8_t* temp_address = (uint8_t*)malloc(sizeof(uint8_t));
    *temp_address = address;
    if ((result = pthread_create(&info->handle, &attr, &_scan_thread,
        temp_address)) != 0)
    {
        free(temp_address);
        mcc172_a_in_scan_stop(address);
        pthread_attr_destroy(&attr);
        free(info->scan_buffer);
        free(info);
        dev->scan_info = NULL;
        return RESULT_RESOURCE_UNAVAIL;
    }

    pthread_attr_destroy(&attr);

    dev->scan_info->scan_running = true;

    // Wait for thread to start to avoid race conditions reading thread status
    bool running;
    do
    {
        usleep(1);
        pthread_mutex_lock(&_devices[address]->scan_mutex);
        running = info->thread_started;
        pthread_mutex_unlock(&_devices[address]->scan_mutex);
    } while (!running);

    return RESULT_SUCCESS;
}
```

Listato 5.1. Implementazione della funzione *mcc172_a_in_scan_start()* presente nella libreria `mcc172.c`

Innanzitutto, per fare chiarezza sul codice, è importante osservare la presenza di due strutture dati, ovvero:

- *mcc172Device*, che raccoglie i metadati associati a ciascun dispositivo, come, ad esempio, la versione del firmware, i suoi dati di fabbrica e altri parametri. Inoltre, tra i campi di questa struttura, vi è anche *scan_info*, che è un puntatore alla struttura *mcc172ScanThreadInfo*.
- *mcc172ScanThreadInfo*, che raccoglie tutti i dati relativi all'acquisizione su ciascun canale. Oltre ai vari flag di controllo, si possono trovare, al suo interno, il vettore *scan_buffer*, che è quello in cui vengono immagazzinati i valori letti, e la variabile *buffer_size*, che ne identifica la dimensione. Ci sono, poi, le variabili *write_index*, *read_index* e *buffer_depth*, che indicano, rispettivamente, l'indice di scrittura e lettura del buffer e il numero di campioni memorizzati al suo interno.

Come prima cosa, nel codice, viene instanziata in maniera dinamica la struttura *dev→scan_info* (successivamente copiata dentro *info*); al suo interno vengono salvati la lista dei canali e i coefficienti per la calibrazione dei dati acquisiti; infine, viene letta la configurazione del clock, rimanendo in attesa che le schede si sincronizzino.

Si passa, poi, al dimensionamento del buffer in cui verranno immagazzinati i valori acquisiti, attraverso la definizione della variabile *info→size_buffer*. In particolare, nel dimensionamento, vengono distinti due casi, ovvero quando si sta acquisendo in *CONTINUOS_MODE* e tutti gli altri casi.

Quando si acquisisce in modalità continua, la dimensione del buffer per ogni canale sarà pari a 1000, 10000 o 100000 elementi, a seconda della frequenza di campionamento scelta. Invece, negli altri casi, tale dimensione sarà pari al parametro *samples_per_channel* fornito alla funzione.

Il parametro appena calcolato verrà usato per l'allocazione dinamica del buffer *info→scan_buffer*.

Si passa, poi, alla valorizzazione della variabile *info→read_threshold*, di fondamentale importanza nell'esecuzione del codice. Infatti, essa indica il numero di elementi che devono essere disponibili sulla memoria della MCC172, prima di inviare il comando, tramite SPI, volto a restituire tali campioni alla Raspberry. Questa soglia viene imposta ad un valore pari ad un decimo della frequenza di campionamento, in maniera tale da effettuare un invio dei dati dalla MCC172 alla Raspberry ogni $100ms$, indipendentemente dalla frequenza alla quale si sta campionando.

Questo è il primo problema che si è presentato nell'utilizzo della scheda. Infatti, sembra evidente come essa sia stata progettata per restituire blocchi di campioni e non un singolo campione alla volta appena acquisito. Ovviamente, ciò non va bene per le specifiche di questo progetto, poiché si ha la necessità di avere ciascun elemento subito disponibile sulla Raspberry, per inserire, eventualmente, i segnali di sincronizzazione. Tale problema verrà affrontato in maniera dettagliata nella Sezione 5.2, dopo aver terminato l'analisi delle tre funzioni.

In seguito alla determinazione della soglia, tramite la funzione *__spi_transfer()*, viene inviato il comando *CMD_AINSCANSTART* per l'avvio dell'acquisizione; come messaggio trasmesso si avrà il vettore *buffer*, "valorizzato" in maniera diversa in base alla modalità di acquisizione scelta, che verrà, così, comunicata alla MCC172.

Infine viene creato lo *__scan_thread*, attivato tramite un parametro che è pari all'indirizzo della scheda sulla quale è partita l'acquisizione. Infatti la funzione *mcc172_a_in_scan_start()* verrà attivata per ciascuna board montata sopra la Raspberry; perciò verranno create tante istanze di *__scan_thread* per quante sono le HAT.

Lo *__scan_thread* si occuperà della ricezione dei valori dalla MCC172 e del riempimento dell' *info→scan_buffer*.

Si passa, ora, all'analisi dello *__scan_thread*. Esso è definito sempre all'interno della libreria `mcc172.c`; il suo codice è mostrato nel Listato 5.2.

```
static void* _scan_thread(void* arg)
{
    bool done;
    uint16_t available_samples;
    uint16_t max_read_now;
    uint16_t read_count;
    int error;
    uint32_t sleep_us;
    uint32_t status_count;
    uint8_t address = *(uint8_t*)arg;
    struct mcc172ScanThreadInfo* info = _devices[address]->scan_info;
    bool calibrated;
    bool scaled;
    bool stop_thread;
    uint8_t rx_buffer[5];
    bool scan_running;
    int result;
```

```
    free(arg);

    if (!_check_addr(address) ||
        (info == NULL))
    {
        return NULL;
    }

    pthread_mutex_lock(&_devices[address]->scan_mutex);
    info->thread_started = true;
    info->thread_running = true;
    info->hw_overrun = false;
    pthread_mutex_unlock(&_devices[address]->scan_mutex);

    status_count = 0;

    if (info->options & OPTS_NOSCALEDATA)
    {
        scaled = false;
    }
    else
    {
        scaled = true;
    }

    if (info->options & OPTS_NOCALIBRATEDATA)
    {
        calibrated = false;
    }
    else
    {
        calibrated = true;
    }

#define MIN_SLEEP_US  100
#define TRIG_SLEEP_US 1000

    done = false;
    sleep_us = MIN_SLEEP_US;
    do
    {
        // read the scan status
        if ((result = _spi_transfer(address, CMD_AINSCANSTATUS, NULL, 0,
            rx_buffer, 5, 1*MSEC, 20)) == RESULT_SUCCESS)
        {
            available_samples = ((uint16_t)rx_buffer[2] << 8) + rx_buffer[1];
            max_read_now = ((uint16_t)rx_buffer[4] << 8) + rx_buffer[3];
            scan_running = (rx_buffer[0] & 0x01) == 0x01;

            pthread_mutex_lock(&_devices[address]->scan_mutex);
            info->hw_overrun = (rx_buffer[0] & 0x02) == 0x02;
            info->triggered = (rx_buffer[0] & 0x04) == 0x04;
            pthread_mutex_unlock(&_devices[address]->scan_mutex);

            status_count++;

            if (info->hw_overrun)
            {
                done = true;
                pthread_mutex_lock(&_devices[address]->scan_mutex);
                info->scan_running = false;
                pthread_mutex_unlock(&_devices[address]->scan_mutex);
            }
            else if (info->triggered == 0)
            {
                // waiting for trigger, use a longer sleep time
                sleep_us = TRIG_SLEEP_US;
            }
            else
            {
                // determine how much data to read
                if (!scan_running ||
                    (available_samples >= info->read_threshold) ||
                    (available_samples > max_read_now))
                {
                    read_count = available_samples;
                    if (max_read_now < read_count)
                    {
                        read_count = max_read_now;
                    }
                    if (read_count > MAX_SAMPLES_READ)
                    {
                        read_count = MAX_SAMPLES_READ;
                    }
                }
                else
                {
                    read_count = 0;
                }

                if (read_count > 0)
                {
                    // handle wrap at end of buffer
```

```
                    if ((info->buffer_size - info->write_index) < read_count)
                    {
                        read_count = (info->buffer_size - info->write_index);
                    }

                    if ((error = _a_in_read_scan_data(address, read_count,
                        scaled, calibrated,
                        &info->scan_buffer[info->write_index])) ==
                        RESULT_SUCCESS)
                    {
                        info->write_index += read_count;
                        if (info->write_index >= info->buffer_size)
                        {
                            info->write_index = 0;
                        }

                        pthread_mutex_lock(&_devices[address]->scan_mutex);
                        info->buffer_depth += read_count;
                        pthread_mutex_unlock(&_devices[address]->scan_mutex);

                        if (info->buffer_depth > info->buffer_size)
                        {
                            pthread_mutex_lock(&_devices[address]->scan_mutex);
                            info->buffer_overrun = true;
                            info->scan_running = false;
                            pthread_mutex_unlock(
                                &_devices[address]->scan_mutex);
                            done = true;
                        }
                        info->samples_transferred += read_count;
                    }

                    // adaptive sleep time to minimize processor usage
                    if (status_count > 4)
                    {
                        sleep_us *= 2;
                    }
                    else if (status_count < 1)
                    {
                        sleep_us /= 2;
                        if (sleep_us < MIN_SLEEP_US)
                        {
                            sleep_us = MIN_SLEEP_US;
                        }
                    }

                    status_count = 0;
                }

                if (!scan_running && (available_samples == read_count))
                {
                    done = true;
                    pthread_mutex_lock(&_devices[address]->scan_mutex);
                    info->scan_running = false;
                    pthread_mutex_unlock(&_devices[address]->scan_mutex);
                }
            }
        }

        usleep(sleep_us);

        pthread_mutex_lock(&_devices[address]->scan_mutex);
        stop_thread = info->stop_thread;
        pthread_mutex_unlock(&_devices[address]->scan_mutex);

    } while (!stop_thread && !done);

    if (info->scan_running)
    {
        // if we are stopped while the device is still running a scan then
        // send the stop scan command
        mcc172_a_in_scan_stop(address);
    }

    pthread_mutex_lock(&_devices[address]->scan_mutex);
    info->thread_running = false;
    pthread_mutex_unlock(&_devices[address]->scan_mutex);
    return NULL;
}
```

Listato 5.2. Implementazione della funzione *__scan_thread()* presente nella libreria `mcc172.c`

Inizialmente si verifica se si vuole un'acquisizione con parametri calibrati e scalati (in tal caso vengono effettuate delle correzioni di offset sulla misura e viene calcolato il valore in Volt), per poi passare al loop principale che si occupa della lettura dei dati dalla MCC172.

All'interno del ciclo *while*, come prima operazione, viene inviato il comando *CMD_AINSCANSTATUS* tramite SPI alla HAT MCC172, che restituisce il vettore *rx_buffer*, contenente le informazioni sullo stato corrente di acquisizione. Nello specifico, le informazioni più importanti che vengono estratte sono:

- *available_samples*, che indica il numero di campioni acquisiti, già disponibili sulla scheda MCC172, ma che non sono stati ancora inviati alla Raspberry;
- *scan_running*, flag che indica se la HAT in questione sta ancora acquisendo dati o meno;
- *hw_overrun*, flag che indica se si è in una situazione di overrun o meno nella memoria della MCC172; in caso affermativo, viene fermata l'acquisizione.

A questo punto si valuta se il numero di campioni disponibili (*available_samples*) è maggiore della soglia prima impostata nella funzione *mcc172_a_in_scan_start()*; quando lo diventa, si inizia la procedura di lettura dei campioni. Si imposta, quindi, il valore *read_count* pari al valore dei campioni disponibili, verificando che sia minore di opportuni limiti dimensionali dovuti al trasferimento su SPI. Inoltre, si verifica che nel vettore *info→scan_buffer* ci sia ancora posto per una quantità di campioni pari a *read_count*, controllando, appunto, che la differenza tra la dimensione massima (*info→buffer_size*) e l'indice di scrittura (*info→write_index*) sia maggiore del numero dei campioni da allocare; in caso contrario, viene ridimensionato *read_count* con questa differenza.

Esso viene passato come argomento alla funzione *_a_in_read_scan_data()*, che si occupa di ricevere i campioni dalla MCC172 e di restituire i valori letti. In particolare, a questa funzione vengono passati i seguenti parametri:

- *address*, per identificare la HAT di riferimento sulla quale svolgere la lettura;
- *read_count*, che indica il numero di campioni che devono essere trasferiti dalla MCC172 alla Raspberry;
- *scaled* e *calibrated*, che indicano, appunto, se si desidera che i campioni restituiti siano calibrati e riportati in Volt, o siano, semplicemente, i valori grezzi restituiti dall'ADC;
- *info→scan_buffer[info→write_index]*, passato per riferimento, che è, come già detto, il vettore che verrà riempito con i valori campionati, a partire dall'indice *info→write_index*.

Una volta acquisiti i campioni, si verifica che l'indice di scrittura non sia arrivato alla fine del buffer; in caso contrario si ricomincia la scrittura dall'indice 0. Si verifica, poi, che il numero di elementi all'interno del buffer non abbia superato la sua dimensione massima, segnalando, nel caso, un errore di *buffer_overrun*.

L'acquisizione dei dati continuerà a ripetersi, svolgendo in maniera ciclica le operazioni sopra descritte, finché non avverrà una condizione di stop. Essa può arrivare dall'interno della HAT, che fermerà il campionamento se si è impostata un'acquisizione con un numero di campioni predeterminato o si è incorsi in una condizione di overrun sulla MCC172. In alternativa, il segnale di stop può essere determinato dall'utente, attraverso l'utilizzo della funzione *mcc172_a_in_scan_stop()*.

L'ultima funzione di cui risulta indispensabile l'analisi è *mcc172_a_in_scan_read*, mostrata nel Listato 5.3.

```
int mcc172_a_in_scan_read(uint8_t address, uint16_t* status,
    int32_t samples_per_channel, double timeout, double* buffer,
    uint32_t buffer_size_samples, uint32_t* samples_read_per_channel)
{
    uint32_t samples_to_read;
    uint32_t samples_read;
    uint32_t current_read;
    uint32_t max_read;
    bool no_timeout;
    bool timed_out;
    bool error;
    uint32_t timeout_us;
    struct mcc172ScanThreadInfo* info;
    struct timespec start_time;
    struct timespec current_time;
    uint16_t stat;
    uint32_t buffer_depth;
    bool hw_overrun;
    bool buffer_overrun;
    bool triggered;
    bool scan_running;
    bool thread_running;

    if (!_check_addr(address) ||
        (status == NULL) ||
        ((samples_per_channel > 0) &&
        ((buffer == NULL) || (buffer_size_samples == 0))))
    {
        return RESULT_BAD_PARAMETER;
    }

    stat = 0;
    samples_read = 0;
    error = false;
    timed_out = false;

    if (timeout < 0.0)
    {
        no_timeout = true;
        timeout_us = 0;
    }
    else
    {
        no_timeout = false;
        timeout_us = (uint32_t)(timeout * 1e6);
    }

    if ((info = _devices[address]->scan_info) == NULL)
    {
        // scan not running?
        *status = 0;
        if (samples_read_per_channel)
        {
            *samples_read_per_channel = 0;
        }
        return RESULT_RESOURCE_UNAVAIL;
    }

    // get thread values
    pthread_mutex_lock(&_devices[address]->scan_mutex);
    buffer_depth = info->buffer_depth;
    hw_overrun = info->hw_overrun;
    buffer_overrun = info->buffer_overrun;
    triggered = info->triggered;
    scan_running = info->scan_running;
    thread_running = info->thread_running;
    pthread_mutex_unlock(&_devices[address]->scan_mutex);

    // Determine how many samples to read
    if (samples_per_channel == -1)
    {
        // return all available, ignore timeout
        samples_to_read = COUNT_NORMALIZE(buffer_depth,
            info->channel_count);
    }
    else
    {
        // return the specified number of samples, depending on the timeout
        samples_to_read = samples_per_channel * info->channel_count;
    }

    if (buffer_size_samples < samples_to_read)
    {
        // buffer is not large enough, so read the amount of samples that will
        // fit
        samples_to_read = COUNT_NORMALIZE(buffer_size_samples,
            info->channel_count);
    }

    if (samples_to_read)
    {
        // Wait for the all of the data to be read or a timeout
```

```
        clock_gettime(CLOCK_MONOTONIC, &start_time);
        timed_out = false;
        do
        {
            // update thread values
            pthread_mutex_lock(&_devices[address]->scan_mutex);
            buffer_depth = info->buffer_depth;
            hw_overrun = info->hw_overrun;
            buffer_overrun = info->buffer_overrun;
            triggered = info->triggered;
            scan_running = info->scan_running;
            thread_running = info->thread_running;
            pthread_mutex_unlock(&_devices[address]->scan_mutex);

            if (buffer_depth >= info->channel_count)
            {
                // read in increments of the number of channels in the scan
                current_read = MIN(buffer_depth, samples_to_read);
                current_read = COUNT_NORMALIZE(current_read,
                    info->channel_count);

                // check for a wrap at the end of the scan buffer
                max_read = info->buffer_size - info->read_index;
                if (max_read < current_read)
                {
                    // when wrapping, perform two copies
                    memcpy(&buffer[samples_read],
                        &info->scan_buffer[info->read_index],
                        max_read*sizeof(double));
                    samples_read += max_read;
                    memcpy(&buffer[samples_read], &info->scan_buffer[0],
                        (current_read - max_read)*sizeof(double));
                    samples_read += (current_read - max_read);
                    info->read_index = (current_read - max_read);
                }
                else
                {
                    memcpy(&buffer[samples_read],
                        &info->scan_buffer[info->read_index],
                        current_read*sizeof(double));
                    samples_read += current_read;
                    info->read_index += current_read;
                    if (info->read_index >= info->buffer_size)
                    {
                        info->read_index = 0;
                    }
                }
                samples_to_read -= current_read;
                buffer_depth -= current_read;
                pthread_mutex_lock(&_devices[address]->scan_mutex);
                info->buffer_depth -= current_read;
                pthread_mutex_unlock(&_devices[address]->scan_mutex);
            }
            usleep(100);

            if (!no_timeout)
            {
                clock_gettime(CLOCK_MONOTONIC, &current_time);
                timed_out = (_difftime_us(&start_time, &current_time) >=
                    timeout_us);
            }

            if (hw_overrun)
            {
                stat |= STATUS_HW_OVERRUN;
                error = true;
            }
            if (buffer_overrun)
            {
                stat |= STATUS_BUFFER_OVERRUN;
                error = true;
            }
        } while ((samples_to_read > 0) && !error &&
            (thread_running == false ? buffer_depth > 0 : true) &&
            !timed_out);

        if (samples_read_per_channel)
        {
            *samples_read_per_channel = samples_read / info->channel_count;
        }
    }
    else
    {
        // just update status
        if (hw_overrun)
        {
            stat |= STATUS_HW_OVERRUN;
        }
        if (buffer_overrun)
        {
            stat |= STATUS_BUFFER_OVERRUN;
        }

        if (samples_read_per_channel)
```

```
            {
                *samples_read_per_channel = 0;
            }
        }

        if (triggered)
        {
            stat |= STATUS_TRIGGERED;
        }
        if (scan_running)
        {
            stat |= STATUS_RUNNING;
        }

        *status = stat;

        if (!no_timeout && (timeout > 0.0) && timed_out && (samples_to_read > 0))
        {
            return RESULT_TIMEOUT;
        }
        else
        {
            return RESULT_SUCCESS;
        }
    }
```

Listato 5.3. Implementazione della funzione *mcc172_a_in_scan_read()* presente nella libreria `mcc172.c`

Questa funzione, una volta invocata, legge uno specifico numero di elementi dal vettore *info→scan_buffer* e li restituisce all'utente attraverso il parametro *buffer* della funzione. Infatti, già da subito, è possibile osservare come, in realtà, i campioni presenti su *info→scan_buffer* abbiano una visibilità limitata all'interno della libreria e non siano accessibili agli utenti che la utilizzano. Per questo, MCC mette a disposizione tale funzione di interfacciamento sulle strutture dati interne alla libreria, dando accesso, così, ai campioni letti.

Inizialmente vengono acquisiti tutti i parametri memorizzati all'interno della struttura *info*. A tali parametri si accede utilizzando un mutex, che impedisce di avere modifiche in parti di codice che vengono eseguite in parallelo ad esso, come, ad esempio, lo *_scan_thread*.

Il numero di elementi per canale che deve essere restituito all'utente è determinato dal parametro *samples_per_channel* fornito alla funzione. In particolare, esso può essere uguale a -1, e ciò indica che bisogna sempre restituire tutti i campioni disponibili su *info→scan_buffer*, a prescindere dalla quantità. Invece, se si pone un numero maggiore di zero, di volta in volta verrà estratto un numero di elementi pari al valore scelto. Poiché tale parametro indica il numero di campioni per canale, bisognerà estrarre da *info→scan_buffer* una quantità di elementi che sarà pari a quella indicata per il numero di canali attivi su quella HAT. Infatti, all'interno del buffer, vengono prima memorizzati i campioni di tutti i canali relativi ad un certo instante temporale, per poi passare ai campioni dell'istante successivo.

Il numero dei campioni totali da leggere è memorizzato all'interno della variabile *samples_to_read*, sulla quale viene effettuato anche un controllo sul suo dimensionamento rispetto al buffer, evitando di leggere quantità superiori alla sua dimensione. Inoltre, viene anche verificato che *samples_to_read* sia minore di *info→buffer_depth*, ovvero del numero dei campioni presenti all'interno del buffer, valorizzando la variabile *current_read* al minimo tra i due.

Si passa, infine, alla copia dei valori da *info→scan_buffer* a *buffer*, passato alla funzione per riferimento. Ciò avviene con l'utilizzo della funzione *memcpy()*, che copia un determinato numero di valori, pari a *current_read*, dal vettore passato come secondo argomento al vettore passato come primo argomento. Possono, però,

incorrere due diverse situazioni nelle quali andare ad effettuare la copia, dovute al fatto che il buffer dal quale si estraggono i valori è stato implementato come coda circolare:

- Nel primo caso si ha che il numero di elementi da estrarre è maggiore della differenza tra la dimensione del buffer e l'indice di lettura. In questo caso, quindi, vengono presi prima gli elementi disponibili in fondo al buffer, a partire dall'indice di lettura, e poi quelli iniziali, partendo dall'indice 0, fino ad ottenere tutti i campioni necessari.
- Nel secondo caso, invece, si ha che la differenza tra la dimensione del buffer e l'indice di lettura è maggiore del numero di campioni da leggere, e perciò si effettua una sola copia dei dati, partendo dall'indice *info→read_index* e prendendo un numero di elementi pari a *current_read*.

Vengono, infine, effettuati dei check sulle variabili di controllo. È interessante effettuare sin da subito alcune considerazioni. Risulta evidente, infatti, che la board MCC172 e la sua libreria siano state create per restituire blocchi di campioni. Infatti, come già descritto, la soglia per l'invio dei campioni è posta ad un decimo della frequenza di campionamento. Questo perché, indipendentemente dai campioni letti al secondo, avere un trasferimento di dati ogni $100ms$ permette alla scheda di lavorare perfettamente. In questo progetto si ha, però, la necessità di avere subito a disposizione sulla Raspberry il dato che viene campionato, poiché, dopo, dovranno essere inseriti anche i segnali di sincronizzazione. È, perciò, evidente che non è possibile acquisire blocchi di campioni, ma la scheda MCC172 deve restituire immediatamente il dato che essa legge. Per fare ciò basta impostare la soglia di lettura prima citata uguale ad 1, con una frequenza di campionamento che, in questo caso, deve essere di $1000Hz$. In questo modo, però, si hanno 1000 invii di dati al secondo tra ADC e Raspberry; la velocità di comunicazione tra le due schede, però, non risulta abbastanza elevata per far ciò. In questo modo si crea un problema nella lettura dei dati, ovvero si avrà che la *_scan_thread()*, non riuscendo a comunicare abbastanza velocemente con la MCC172, non inserisce, sempre nel vettore *info→scan_buffer*, un campione per canale alla volta, ma avrà momenti casuali in cui ne inserirà un numero maggiore. Considerando che il parametro *samples_per_channel* della funzione *mcc172_a_in_scan_read* dovrebbe essere posto ad 1 per ottenere in uscita un campione per canale alla volta, si ottiene che, nel funzionamento ideale, si dovrebbe avere che un campione alla volta venga prima copiato in *info→scan_buffer*, per poi essere di nuovo copiato nel vettore di output *buffer*. Nella realtà succede, però, che vi è un accumulo dei campioni sul primo buffer, che, in certi instanti, riceve insieme molti campioni, e viene svuotato dalla funzione *mcc172_a_in_scan_read* soltanto di un campione alla volta. Verranno visti, nelle prossime sezioni, alcuni test a dimostrazione di quanto appena detto, e verranno viste le differenze di prestazione nel caso in cui vengano usate 1 o 2 HAT.

5.2 Valutazione dei risultati con un'unica HAT

Nella progettazione del software per la Raspberry presente nell'End-Node, si è partiti dallo sviluppo di un codice che fosse in grado di acquisire con un'unica

HAT in entrambi i canali. A tal proposito, nella libreria *daqhats*, nella directory `daqhats/examples/c/mcc172`, è presente un set di esempi per le varie modalità di acquisizione; in particolare, si trova l'esempio `continuous_scan`, che contiene il file `continuous_scan.c`.

Al suo interno è sviluppato un programma che si occupa dell'acquisizione, in modalità continua, su una sola HAT, e utilizzando un numero di canali (uno o due) che deve essere indicato.

Senza inserire l'intero codice, si possono osservare solo alcuni dettagli, sia implementativi che di risultati.

Il codice è sviluppato con una logica molto simile a quella vista nei Listati 4.7 e 4.8, che comprendono una fase di inizializzazione della scheda, nella quale vengono richiamate le funzioni *mcc172_open()*, *mcc172_iepe_config_write()*, *mcc172_a_in_clock_config_write()* e *mcc172_a_in_clock_config_read()*, ovviamente per una sola HAT, a differenza di quanto era stato visto, e, successivamente, viene attivata la scansione attraverso l'utilizzo della funzione *mcc172_a_in_scan_start()*.

La scansione viene attivata con una frequenza di campionamento di $1000Hz$. Come già detto, la frequenza effettiva di campionamento verrà impostata come la più vicina a quella fornita. In particolare, la MCC172 può generare un clock di campionamento dell'ADC pari a 51,2 kHz diviso per un numero intero compreso tra 1 e 256. Perciò, la frequenza più vicina disponibile sarà pari a $1003Hz$. Questa risulterà essere anche la frequenza effettiva di campionamento nel software finale implementato nell'End-Node.

Ciò significa che la soglia *info→read_threshold*, che viene impostata all'interno della funzione *mcc172_a_in_scan_start()*, è pari a 100, ovvero ogni 100 campioni, quindi circa ogni $100ms$, gli stessi vengono trasferiti dalla memoria interna della MCC172 alla Raspberry, tramite protocollo SPI. Infine, viene attivata la funzione *mcc172_a_in_scan_read()* all'interno di un loop, poiché, come spiegato nella sezione precedente, essa esegue una singola lettura di campioni, che possono essere uno o più.

Nell'esempio, a questa funzione, viene passato il parametro *read_request_size* (il suo parametro nominale è *samples_per_channel*) pari a -1, che indica, appunto, di restituire in uscita tutti i valori presenti in *info→scan_buffer*. L'utilizzo della funzione *mcc172_a_in_scan_read()* nell'esempio `continuous_scan.c` è mostrato nel Listato 5.4.

```
do
{
    // Since the read_request_size is set to -1 (READ_ALL_AVAILABLE), this
    // function returns immediately with whatever samples are available (up
    // to user_buffer_size) and the timeout parameter is ignored.
    result = mcc172_a_in_scan_read(address, &read_status, read_request_size,
        timeout, read_buf, user_buffer_size, &samples_read_per_channel);
    STOP_ON_ERROR(result);

    if (read_status & STATUS_HW_OVERRUN)
    {
        printf("\n\nHardware overrun\n");
        break;
    }
    else if (read_status & STATUS_BUFFER_OVERRUN)
    {
        printf("\n\nBuffer overrun\n");
        break;
    }

    total_samples_read += samples_read_per_channel;
```

```
        if (samples_read_per_channel > 0)
        {
            // Display the samples read and total samples
            printf("\r%12.0d  %10.0d ", samples_read_per_channel,
                total_samples_read);

            // Calculate and display RMS voltage of the input data
            for (i = 0; i < num_channels; i++)
            {
                printf("%10.4f",
                    calc_rms(read_buf, i, num_channels,
                        samples_read_per_channel));
            }
            fflush(stdout);
        }

        usleep(50);
    }
    while ((result == RESULT_SUCCESS) &&
        ((read_status & STATUS_RUNNING) == STATUS_RUNNING) &&
        !enter_press());
```

Listato 5.4. Utilizzo della funzione *mcc172_a_in_scan_read()* nell'esempio di acquisizione continua su una sola HAT `continuous_scan.c`

La funzione *mcc172_a_in_scan_read()* "valorizza" il vettore *read_buf* con i campioni acquisiti. Per ciascuna lettura in cui si acquisisce un blocco di campioni, viene calcolato il suo valore quadratico medio, ovvero il valore efficace del segnale, attraverso la funzione *calc_rms()*; tale parametro, una volta calcolato, viene visualizzato.

Chiaramente, questo non è il comportamento desiderato. Vengono, così, sin da subito fatte queste due modifiche:

- La prima riguarda direttamente la libreria `mcc172.c` e, in particolare, la funzione *mcc172_a_in_scan_start()*. Qui, viene impostata la soglia di lettura, ovvero *info→read_threshold*, direttamente uguale ad 1, in maniera che, ogni volta che si avrà almeno un campione disponibile per ciascuna HAT, avverrà l'invio dei dati sul canale SPI. Inoltre, nel caso di due canali attivi su una singola HAT, essa acquisisce entrambi i campioni nello stesso istante; quindi, i campioni disponibili sulla memoria della MCC172 passeranno da 0 a 2, determinando, allo stesso modo, il superamento della soglia posta uguale ad 1.
- La seconda modifica riguarda il parametro *read_request_size* fornito alla funzione *mcc172_a_in_scan_read()*. In particolare, esso viene posto uguale ad 1, facendo si che, per ogni chiamata della funzione, venga restituito solo un campione per canale. In questo modo, si dovrebbe ottenere un comportamento ideale, nel quale la MCC172 trasferisce immediatamente ciascun campione acquisito alla Raspberry; subito dopo tale campione viene restituito in uscita all'utente che sta usando l'applicazione.

Nella realtà tale comportamento non è, però, sempre verificato.

Per il test è stata utilizzata la funzione *mcc172_a_in_scan_status()*, che restituisce come parametro il numero di campioni presenti in *info→scan_buffer* per ciascun canale, ovvero, *sample_available*. Si può usare proprio questo valore per vedere quanti campioni rimangono in tale buffer dopo l'attivazione della funzione *mcc172_a_in_scan_read()*. Nel caso ideale, si dovrebbe avere sempre un valore pari a 0, che indicherebbe, appunto, il non accumulo dei campioni.

Viene, quindi, inserita questa funzione nel codice, subito dopo la lettura dei campioni, e viene salvato il valore *sample_available* in un vettore, in modo da poterlo, successivamente, analizzare e graficare.

Nel Listato 5.5 è mostrata l'aggiunta effettuata al codice precedente, all'interno del loop di lettura.

```
mcc172_a_in_scan_status(address, &read_status, &samples_available);
accumulatore[total_samples_read]=samples_available;
```

Listato 5.5. Inserimento della funzione *mcc172_a_in_scan_status()* nel loop di lettura, nell'esempio di acquisizione continua su una sola HAT `continuous_scan.c`

Il vettore in questione viene, poi, salvato su un file `.csv` e, attraverso la libreria *Pandas* di Python, viene visualizzato su uno *ScatterPlot*. Il risultato è mostrato in Figura 5.1.

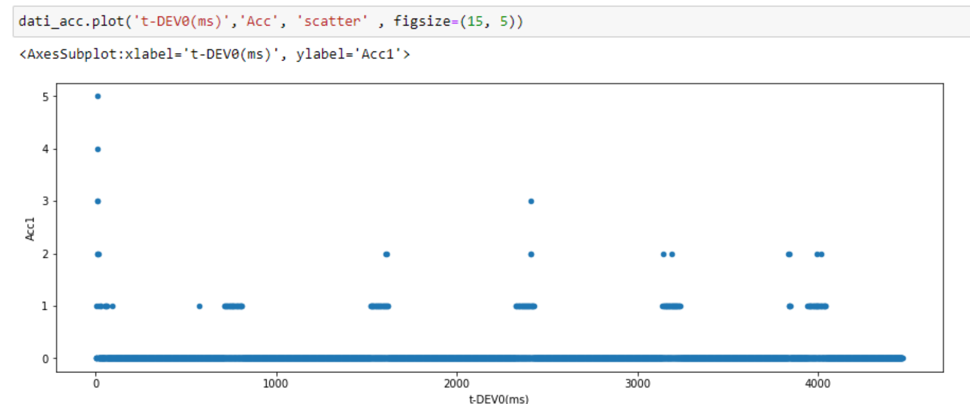


Figura 5.1. Campioni disponibili su *info→scan_buffer* dopo l'attivazione della funzione *mcc172_a_in_scan_read()*

Nel grafico è rappresentato, per ogni istante di campionamento, il numero di elementi rimasti su *info→scan_buffer*, una volta avvenuta la lettura. È evidente che il comportamento ideale non è sempre rispettato.

Infatti, ci sono molti momenti in cui nel buffer si accumulano campioni. Ciò è dovuto al fatto che la Raspberry non riesce a gestire la comunicazione con la MCC172, tramite protocollo SPI, con quelle velocità. Si creano, così, dei buchi temporali durante i quali la comunicazione è interrotta, e, ovviamente, non vengono inviati campioni alla Raspberry.

Ciò è stato verificato utilizzando un oscilloscopio digitale, collegato ai pin usati per la comunicazione SPI, e graficando i segnali trasmessi. Il risultato è visibile in Figura 5.2. In tale figura, sono visualizzati i 4 segnali di trasmissione usati nel protocollo SPI, ovvero *MISO* (Master Input, Slave Output), *MOSI* (Master Output, Slave Input), *Clock* ed *Enable*. Nel centro della figura è possibile notare un buco temporale di circa $2ms$, nel quale la trasmissione si è interrotta senza esplicita richiesta.

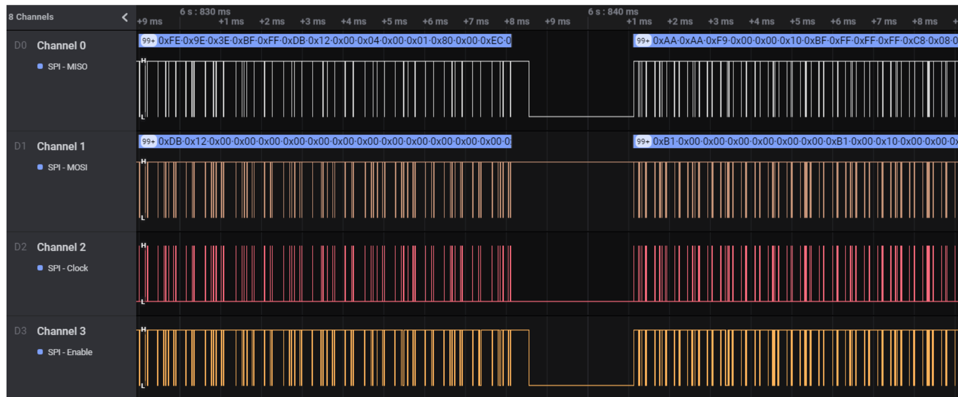


Figura 5.2. Visualizzazione, tramite oscilloscopio, di una delle interruzioni dell'SPI

È stato verificato su diversi forum online che questa problematica è comune, con l'utilizzo della Raspberry, quando si lavora con l'SPI a frequenze molto elevate; non esiste una soluzione a questo problema se non attraverso una riduzione del carico computazionale del processore della Raspberry. Infatti, seppure un'acquisizione effettuata a $1000Hz$ non sembri troppo elevata, non bisogna dimenticare che, per ogni campione acquisito, prima viene mandato sul canale SPI il comando *CMD_AINSCANSTATUS* e, successivamente, bisogna aspettare la risposta dalla MCC172, determinando un aumento significativo dei dati scambiati ogni secondo con ciascuna HAT.

Tornando alla Figura 5.1, negli istanti in cui si accumulano campioni, avvengono proprio queste interruzioni di trasmissione. Infatti, se la Raspberry smette di comunicare con la MCC172, nella memoria interna della HAT iniziano ad accumularsi campioni. Nel momento in cui la trasmissione riprende a funzionare, la Raspberry, attraverso l'utilizzo del *_scan_thread*, richiede tutti i campioni che ha a disposizione la MCC172. A questo punto, quest'ultima ne ritornerà un numero maggiore di 1; perciò, dopo che la funzione *mcc172_a_in_scan_read()* avrà tolto dal buffer un campione, in esso ne rimarrà un numero diverso da zero. Successivamente, nell'intervallo di qualche millisecondo, la *mcc172_a_in_scan_read()* restituirà, uno alla volta, tutti i campioni sul buffer, riportando un valore accumulato di campioni di nuovo pari a 0.

Sulla libreria originale, infatti, l'invio dei dati dati dalla MCC172 alla Rasberry avveniva soltanto 10 volte al secondo, e non 1000, come nella versione modificata. Infatti, fare invii con quelle temporizzazioni non comporta nessun tipo di carico computazionale eccessivo alla Raspberry, che riesce a svolgere tutte le funzioni richieste in maniera perfetta. Attraverso prove empiriche si è osservato che si ottiene un comportamento ottimale fino a circa 200 invii di dati al secondo.

È importante osservare un'ultima cosa, ovvero che la MCC172 acquisisce costantemente i campioni, con estrema precisione temporale e sulla misura, non perdendone nessuno. Essi, infatti, sono semplicemente inviati alla Raspberry in ritardo, in ordine rispetto all'istante di acquisizione, per i motivi appena descritti, ma la loro acquisizione avviene nell'istante esatto.

5.3 Valutazione dei risultati con 2 HAT

Nel caso d'utilizzo di più HAT, in particolare due, le prestazioni peggiorano notevolmente. Ciò è spiegato dal fatto che vengono effettuate due istanze di *_scan_thread*, con la conseguente creazione di due vettori *info→scan_buffer*, uno per ciascuna board montata. Quindi, la Raspberry dovrà gestire la comunicazione SPI con due schede, raddoppiando il numero di invii di dati al secondo che da esse partono.

Il codice implementato risulta essere molto simile a quello mostrato nel Listato 5.4. Ovviamente, in questo caso, tutte le funzioni di inizializzazione vengono attivate due volte, una per scheda; la stessa cosa vale per la funzione *mcc172_a_in_scan_start()*. Inoltre, dentro il loop principale del programma, ad ogni ciclo, la funzione *mcc172_a_in_scan_read()* viene attivata due volte, per copiare gli elementi da entrambi i buffer creati e, per restituirli in uscita all'utente.

L'implementazione della parte di inizializzazione e di avvio della scansione per il caso Multi-HAT è identica a quella mostrata nel thread *Measurement* nei Listati 4.7 e 4.8. La parte di restituzione dei campioni attraverso la funzione *mcc172_a_in_scan_read()*, come già spiegato, non è stata utilizzata nel programma finale a seguito delle modifiche effettuate sulla libreria, che verranno viste più avanti. Essa è, perciò, mostrata nel Listato 5.6, in cui è riportato il loop di acquisizione presente nel file `continuous_scan_multi_hat.c`.

```
do
{
    for (device = 0; device < DEVICE_COUNT; device++)
    {
        // Since the read_request_size is set to -1 (READ_ALL_AVAILABLE), this
        // function returns immediately with whatever samples are available (up
        // to user_buffer_size) and the timeout parameter is ignored.

        result = mcc172_a_in_scan_read(address[device],
            &scan_status[device], read_request_size, timeout, data[device],
            user_buffer_size, &samples_read_per_channel[device]);
        mcc172_a_in_scan_status(address[device], &scan_status[device],
                                &samples_available[device]);
        STOP_ON_ERROR(result);
        // Check for overrun on any one device
        scan_status_all |= scan_status[device];
        total_samples_read[device] += samples_read_per_channel[device];

    }


    if (scan_status_all & STATUS_HW_OVERRUN)
    {
        printf("\n\nHardware overrun\n");
        break;
    }
    else if (scan_status_all & STATUS_BUFFER_OVERRUN)
    {
        printf("\n\nBuffer overrun\n");
        break;
    }
    accumulatore[0][total_samples_read[MASTER]]=samples_available[0];
    accumulatore[1][total_samples_read[MASTER]]=samples_available[1];
}
while ((result == RESULT_SUCCESS) &&
      ((scan_status_all & STATUS_RUNNING) == STATUS_RUNNING) &&
      !enter_press());
```

Listato 5.6. Utilizzo della funzione *mcc172_a_in_scan_read()* nell'esempio di acquisizione continua con 2 HAT nel file `continuous_scan_multi_hat.c`

Come fatto per il caso di una singola HAT, è stata utilizzata la funzione *mcc172_a_in_scan_status()*, ottenendo, così, per ogni istante di campionamento e per ciascuna HAT, il numero di campioni che si accumulano sul vettore

info→scan_buffer dopo aver attivato la funzione di lettura *mcc172_a_in_scan_read()*. Anche in questo caso, il numero di campioni disponibili, per entrambe le schede, viene salvato sulla matrice *accumulatore*, che conterrà, nella prima riga, i dati relativi alla prima board, e nella seconda riga i dati relativi alla seconda scheda. Essi vengono salvati su un file `.csv` e graficati su Python, ottenendo i risultati riportati in Figura 5.3.

Figura 5.3. Campioni disponibili su *info→scan_buffer*, per entrambe le HAT, dopo l'attivazione della funzione *mcc172_a_in_scan_read()*

Come precedentemente affermato, le prestazioni peggiorano notevolmente. Infatti, si può notare che non vi sono mai intervalli temporali in cui si hanno 0 campioni accumulati sul buffer, ottenendo, invece, istanti nei quali si arriva fino ad un valore pari a 100.

Si può tranquillamente affermare che questo comportamento non è accettabile per lo sviluppo del progetto finale. Nelle prossime sezioni verranno illustrate le modifiche effettuate sul sistema operativo e ai driver della MCC172, per ottenere il comportamento desiderato.

5.4 Prova al passaggio del kernel in Real Time

La prima soluzione proposta è stata l'installazione della patch Real Time del sistema operativo Raspbian Lite; essa ha significativamente cambiato le prestazioni della Raspberry, non raggiungendo, però, un livello di prestazioni ottimale.

Le prestazioni in tempo reale si riferiscono al tempo di risposta di un'attività durante lo scheduling eseguito dal sistema operativo. Generalmente, questo tempo è molto variabile, secondo il carico computazionale del sistema preso in esame; esso dipende, anche, dal tipo di sistema operativo in esecuzione.

Alcune applicazioni in ambito industriale richiedono una maggiore precisione temporale in alcune delle attività svolte. In particolare, spesso non si può permettere di avere una variabilità troppo grande sulle schedulazioni di determinate attività che dovrebbero essere, invece, effettuate con molta più precisione.

Questo è proprio quello che accade nel progetto in considerazione. Infatti, la gestione della comunicazione SPI deve avvenire con estrema precisione, per permettere un campionamento, con la MCC172, a $1000Hz$. Molto probabilmente, senza una gestione in Real Time, il tempo concesso ad altre attività, che siano proprie della Raspberry, o ad altri processi mandati in esecuzione dell'utente, risulta essere maggiore di quello che si ha nella versione Real Time, togliendo, così, tempo ai processi di gestione dell'SPI e causando dei buchi temporali nella comunicazione, come visto precedentemente.

Ora, verranno descritti, in maniera abbastanza sintetica, i passaggi svolti per l'istallazione della patch. Quello che verrà fatto, in breve, sarà scaricare e compilare un nuovo kernel, con la patch del Real Time già applicata, configurare le sue impostazioni ed, infine, fare l'installazione del nuovo kernel. Alla nuova accensione della Raspberry, verrà usato proprio tale kernel per avviare il sistema.

I passaggi svolti sono i seguenti:

- Una volta installata l'ultima versione del sistema operativo Raspbian all'interno della Raspberry, bisogna, dapprima, installare alcune librerie, che permetteranno l'accesso al codice del nuovo kernel, presente su una cartella Ghithub, e la sua compilazione. Inoltre, si dovrà accedere al repository Git adeguato e selezionare il branch Real-Time del kernel 4.19. In tale branch, al kernel 4.19 è stata già applicata la patch Real-Time (rt). Il codice usato nel terminale della Raspberry è quello del Listato 5.7.

```
sudo apt install git bc bison flex libssl-dev make libncurses5-dev
git clone --depth=1 --branch rpi-4.19.y-rt https://github.com/raspberrypi/linux
```

Listato 5.7. Download del codice del kerner Real-Time

- Successivamente, bisogna effettuare la compilazione del kernel con le sue impostazioni di default, attraverso il codice del Listato 5.8, inserito nel terminale della Raspberry.

```
cd linux
KERNEL=kernel7
make bcm2709_defconfig
```

Listato 5.8. Compilazione del kerner Real-time con impostazioni di default

- Si cambiano, poi, alcune impostazioni di configurazione del kernel. Ciò avviene attraverso la *menuconfig interface*, che mette a disposizione un'interfaccia per la modifica delle impostazioni del kernel. In particolare, vengono effettuate due modifiche:
  - viene abilitato *CONFIG_PREEMPT_RT_FULL*, attraverso il percorso *Kernel Features → Preemption Model (Fully Preemptible Kernel (RT)) → Fully Preemptible Kernel (RT)*;
  - viene definito il paramatro *CONFIG_HZ* pari a $1000Hz$, per avere un tempo di latenza dei processi più basso, attraverso il percorso *Kernel Features → Timer frequency*.

  Per accedere al menu bisogna digitare il codice del Listato 5.9.

```
make menuconfig
```

Listato 5.9. Apertura del menù per la configurazione del kernel

Una volta effettuate le modifiche, bisogna uscire dal menù ed effettuare il salvataggio, che porterà alla modifica del file `.config`. Inoltre, per non sovrascrivere il vecchio kernel presente sulla Raspberry, bisogna cambiare il parametro *CONFIG_LOCALVERSION* per assicurarsi che il nuovo kernel non riceva la stessa stringa di versione del kernel in esecuzione. In questo modo, si potrà chiarire quale kernel si sta eseguendo una volta che l'installazione sarà finita, e assicurasi che i moduli esistenti in `/lib/modules` non vengano sovrascritti. Per fare ciò bisogna cambiare la riga mostrata nel Listato 5.10, nel file `.config`.

```
CONFIG_LOCALVERSION="-v7-rt+"
```

Listato 5.10. Modifica per visualizzare quale kernel è in esecuzione

- Infine, si devranno compilare e installare il kernel, i moduli e i Device Tree blobs; tale passaggio può richiedere molto tempo. Il codice utilizzato è quello mostrato nel Listato 5.11.

```
make -j4 zImage modules dtbs
sudo make modules_install
sudo cp arch/arm/boot/dts/*.dtb /boot/
sudo cp arch/arm/boot/dts/overlays/*.dtb* /boot/overlays/
sudo cp arch/arm/boot/dts/overlays/README /boot/overlays/
sudo cp arch/arm/boot/zImage /boot/$KERNEL.img
```

Listato 5.11. Installazione del kernel

Tutti i passaggi mostrati sono validi soltanto per il modello Raspberry Pi 3B+ a 32 bit. Per altre versioni, bisognerà eseguire una compilazione del kernel diversa.

A questo punto, una volta che si accede alla Raspberry, attraverso il comando *uname -r*, sarà possibile leggere la versione del kernel in esecuzione, che sarà, appunto, quella Real Time.

Come ultimo passo, sono stati svolti di nuovo i test con l'esempio di acquisizione con 2 HAT visto nella sezione precedente, salvando i valori dei campioni accumulati sul buffer *info→scan_buffer* e graficandoli attraverso l'utilizzo di Python. I risultati sono visibili in Figura 5.4.

Figura 5.4. Campioni disponibili su *info→scan_buffer*, per entrambe le HAT, dopo l'attivazione della funzione *mcc172_a_in_scan_read()*, utilizzando la versione Real Time del sistema operativo Raspbian Lite

È evidente come le prestazioni siano migliorate in maniera sostanziale. In particolare, sulla HAT0 si continua ad avere un comportamento non molto buono; infatti si accumulano costantemente 1 o 2 campioni per canale, e quasi mai si è nella situazione di funzionamento ideale con 0 campioni sul buffer. Invece, per la HAT1, il funzionamento è molto vicino a quello desiderato.

Nonostante questi miglioramenti, non si possono ancora considerare accettabili le prestazioni della scheda e, in particolare, la velocità con la quale vengono restituiti i campioni. Si è deciso, quindi, di agire direttamente sui driver che comandano la scheda e l'acquisizione e, in particolare, sulla libreria `mcc172.c`.

5.5 Modifica di `mcc172.c` per alleggerire il carico dell'acquisizione

L'idea della modifica della libreria è nata da un'ipotesi ben precisa, ovvero si è cercato di alleggerire al massimo il carico computazionale della Raspberry durante l'acquisizione dei campioni, in maniera tale da lasciare più tempo al processore per la gestione dell'SPI, ipotizzando, quindi, una migliore gestione della comunicazione.

Nella Figura 5.5 è riassunto il processo di acquisizione descritto nella Sezione 5.1.

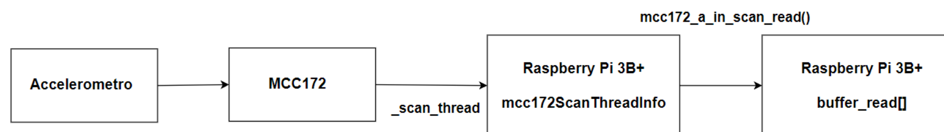


Figura 5.5. Schema a blocchi per riassumere il processo di acquisizione dei campioni

Come era già stato osservato, la funzione *mcc172_a_in_scan_read()* non fa altro che andare ad eseguire, per ciascun campione letto e per ciascuna HAT, una copia da una buffer ad un altro, entrambi all'interno della memoria della Raspberry. Infatti, essa serve per rendere disponibili i campioni all'utente, dato che la visibilità del vettore *info→scan_buffer* è limitata all'interno della libreria, e non vi è modo di accedere ad esso se non attraverso l'uso di funzioni di interfaccia.

Si è deciso, quindi, di rimuovere questa doppia copia di campioni. Ciò evita la creazione di un ciclo per la lettura nel processo d'acquisizione e, soprattutto, evita la copia di 3000 campioni (1000 per canale) al secondo da parte della Raspberry, il che fa notevolmente alleggerire il suo carico computazionale. Perciò, i campioni acquisiti rimangono all'interno di *info→scan_buffer*, che viene riempito fino alla fine dell'acquisizione; successivamente, da lì vengono trasferiti su un file `.csv`.

Ovviamente, per escludere l'utilizzo della funzione *mcc172_a_in_scan_read()*, sarà necessario apportare delle modifiche a *_scan_thread()*. Inoltre, nella libreria `mcc172.c`, sarà necessario aggiungere la funzione per il salvataggio dei dati acquisiti, proprio perché quei campioni saranno visibili solo all'interno della libreria. Infatti, non vi è necessità di restituire all'utente ciascun campione letto, poiché tali dati non verranno manipolati in nessun modo, se non con l'inserimento dei segnali di sincronizzazione, che sarà effettuato direttamente all'interno di *_scan_thread()*. L'inserimento dei BEACON verrà visto nella fase di implementazione del codice finale, che verrà affrontata nel Capitolo 6.

Le modifiche effettuate hanno riguardato le altre due funzioni descritte nella Sezione 5.1, ovvero *mcc172_a_in_scan_start()* e *_scan_thread()*, ma anche le funzioni usate dopo aver fermato l'acquisizione per chiudere il collegamento con la specifica HAT, ovvero *mcc172_a_in_scan_cleanup()* e *mcc172_close()*.

Inoltre, viene modificata la struttura *mcc172ScanThreadInfo*, con l'aggiunta del campo *samples_count*, che sarà un puntatore ad intero *uint16_t*. Esso viene aggiunto per verificare quanti campioni vengono trasferiti, durante ciascuna lettura, dalla MCC172 alla Raspberry. Infatti, a tale scopo, l'utilizzo della funzione *mcc172_a_in_scan_status()* non ha più senso, poiché i campioni non vengono mai tolti dal buffer, e quindi non si possono valutare eventuali accumuli di campioni.

Si parte, quindi, dalla funzione *mcc172_a_in_scan_start()*, che ha subito modifiche sulla valorizzazione del parametro *info→buffer_size*. Infatti, nella versione originale, tale parametro veniva "valorizzato" in maniera diversa in base alla frequenza di campionamento, poiché esso veniva continuamente riempito da *_scan_thread()* e svuotato da *mcc172_a_in_scan_read()*.

Ora, invece, il buffer non verrà mai svuotato fino alla fine dell'acquisizione; perciò è necessario definire la sua corretta dimensione. Il parametro *info→buffer_size* viene "valorizzato" a *samples_per_channel*, fornito come parametro alla funzione *mcc172_a_in_scan_start()*, per il numero di canali attivi, indipendentemente dalla frequenza di campionamento e dalla modalità di acquisizione.

Inoltre, viene aggiunta l'allocazione della memoria, in maniera dinamica, per il vettore *info→samples_count*, che dovrà avere la stessa dimensione di *info→scan_buffer*.

Le modifiche effettuate nella funzione *mcc172_a_in_scan_start()* sono visibili nel Listato 5.12, in cui è descritto il pezzo di codice aggiunto prima della creazione di *_scan_thread*.

```
info->buffer_size = samples_per_channel;
info->buffer_size *= num_channels;

// allocate the buffer
info->scan_buffer = (double*)calloc(1, info->buffer_size * sizeof(double));
if (info->scan_buffer == NULL)
{
    // can't allocate memory
    free(info);
    dev->scan_info = NULL;
    return RESULT_RESOURCE_UNAVAIL;
}

 // allocate the samples_count buffer
info->samples_count = (uint16_t*)calloc(1, info->buffer_size * sizeof(uint16_t));
if (info->samples_count == NULL)
{
    // can't allocate memory
    free(info);
    dev->scan_info = NULL;
    return RESULT_RESOURCE_UNAVAIL;
}

info->read_threshold = 1;
```

Listato 5.12. Modifica del codice della funzione *mcc172_a_in_scan_start()* all'interno della libreria `mcc172.c`

Successivamente, si è passati alla modifica della funzione *_scan_thread*, che è quella che ha subito le modifiche maggiori.

Innanzitutto, è stato necessario introdurre nuovi parametri di controllo all'interno della struttura *mcc172ScanThreadInfo* che permettessero di scrivere e leggere il vettore *info→scan_buffer* che, in questo, caso non viene mai svuotato, ma solo riempito fino alla fine dell'acquisizione. In ottica di un suo futuro uso all'interno del programma finale dell'End-Node, il buffer è stato concepito come una coda circolare. Infatti, ad esempio nella parte di acquisizione precedente al Trigger, si devono memorizzare soltanto gli ultimi 20 secondi di dati acquisiti prima di tale segnale. Perciò, si inseriscono continuamente i campioni all'interno di una coda circolare di 20000 elementi e, all'arrivo del TRIGGER, si avranno solo i campioni più recenti.

A tale proposito, nella struttura nella quale vengono memorizzati i dati relativi al thread di acquisizione, ovvero in *mcc172ScanThreadInfo*, vengono inseriti i seguenti campi:

- *volatile uint32_t front*, che indica la posizione dell'elemento più vecchio del vettore; sarà proprio da qui che dovrà partire la lettura dei campioni una volta che si salverà l'acquisizione fatta;
- *volatile uint32_t rear*, che indica la posizione nella quale si dovrà inserire il prossimo campione acquisito.

In particolare, all'interno di *__scan__thread*, questi campi vengono inizializzati entrambi al valore 0.

Nel Listato 5.13 è possibile osservare come il loop di acquisizione, presente nel thread, è stato modificato.

```
do
{
    // read the scan status
    if ((result = _spi_transfer(address, CMD_AINSCANSTATUS, NULL, 0,
        rx_buffer, 5, 1*MSEC, 20)) == RESULT_SUCCESS)
    {
        available_samples = ((uint16_t)rx_buffer[2] << 8) + rx_buffer[1];
        max_read_now = ((uint16_t)rx_buffer[4] << 8) + rx_buffer[3];
        scan_running = (rx_buffer[0] & 0x01) == 0x01;

        pthread_mutex_lock(&_devices[address]->scan_mutex);
        info->hw_overrun = (rx_buffer[0] & 0x02) == 0x02;
        pthread_mutex_unlock(&_devices[address]->scan_mutex);

        status_count++;

        if (info->hw_overrun)
        {
            done = true;
            pthread_mutex_lock(&_devices[address]->scan_mutex);
            info->scan_running = false;
            pthread_mutex_unlock(&_devices[address]->scan_mutex);
        }
        else
        {
            // determine how much data to read
            if (!scan_running ||
                (available_samples >= info->read_threshold) ||
                (available_samples > max_read_now))
            {
                read_count = available_samples;
                if (max_read_now < read_count)
                {
                    read_count = max_read_now;
                }
            }
            else
            {
                read_count = 0;
            }

            if (read_count > 0)
            {
            // handle wrap at end of buffer
            if ((info->buffer_size - info->rear) < read_count)
            {
                read_count = (info->buffer_size - info->rear);
            }
            if (info->buffer_size == info->buffer_depth)
            {
                info->front= (info->front + read_count) % info->buffer_size;
                info->buffer_depth= info->buffer_depth - read_count;
            }

            if ((error = _a_in_read_scan_data(address, read_count,
                    scaled, calibrated,
                    &info->scan_buffer[info->rear])) ==
                    RESULT_SUCCESS)
            {
                info->samples_count[info->rear]=read_count;
                info->rear = (info->rear + read_count);
                info->buffer_depth= info->buffer_depth + read_count;
                if (info->buffer_size<50000)
                {
                    while(read_count>1)
                    {
                        info->samples_count[info->rear-read_count+1]= 0;
                        read_count--;
                    }
                    info->rear = info->rear % info->buffer_size;
                }
            }

            // adaptive sleep time to minimize processor usage
            if (status_count > 4)
            {
                sleep_us *= 2;
            }
            else if (status_count < 1)
            {
                sleep_us /= 2;
                if (sleep_us < MIN_SLEEP_US)
                {
```

```
                    sleep_us = MIN_SLEEP_US;
                }
            }

            status_count = 0;
            }

            if (!scan_running && (available_samples == read_count))
            {
                done = true;
                pthread_mutex_lock(&_devices[address]->scan_mutex);
                info->scan_running = false;
                pthread_mutex_unlock(&_devices[address]->scan_mutex);
            }
        }
    }

    usleep(sleep_us);

    pthread_mutex_lock(&_devices[address]->scan_mutex);
    stop_thread = info->stop_thread;
    pthread_mutex_unlock(&_devices[address]->scan_mutex);

} while (!stop_thread && !done);
```

Listato 5.13. Modifica del codice della funzione *__scan__thread* all'interno della libreria `mcc172.c`

Come veniva fatto, anche, nella libreria originale, inizialmente, viene inviato il comando *CMD_AINSCANSTATUS*, che restituisce i parametri di controllo dell'acquisizione (da cui viene tolto *info→triggered*, poiché non utilizzato) e, soprattutto, il numero di campioni acquisiti disponibili sulla scheda.

Poiché la soglia *info→read_threshold* è stata impostata uguale ad 1, ogni volta che si avrà almeno un campione in memoria sulla MCC172 avverrà il trasferimento dei dati acquisiti.

Come prima cosa, si effettua un controllo sulla dimensione di *read_count*, che non deve eccedere la dimensione del buffer; in caso contrario lo si pone pari al numero massimo di elementi disponibili.

Successivamente, si analizza il numero di elementi presenti nel buffer. Infatti, come già spiegato nel capitolo precedente, *info→buffer_depth* indica il numero di elementi salvati nel buffer. Perciò, se essi sono pari alla dimensione massima del vettore, vuol dire che il buffer è stato completamente riempito e, per continuare l'acquisizione, è necessario rimuovere gli elementi più vecchi, implementando, così, una coda circolare. Infatti, l'indice *info→front*, che indica l'elemento più vecchio, viene fatto avanzare di una quantità pari a *read_count*, con eventuale azzeramento se si è arrivati all'ultimo elemento del buffer; *info→buffer_depth*, invece, viene decrementato della stessa quantità.

Viene, così, effettuato l'invio dei dati dalla MCC172; questi ultimi vengono memorizzati nel vettore *info→scan_buffer* partendo dalla posizione indicata in *info→rear*. Viene, inoltre, salvato il numero di campioni acquisiti in quella lettura, all'interno di *info→samples_count*, all'indice *info→rear*. Nel caso si legga più di un campione, è necessario porre uguale a 0 i restanti elementi del vettore *info→samples_count* fino a raggiungere la quantità *read_count*. Infatti, se in un'istante si sono letti, ad esempio, 2 campioni per un canale, memorizzati in *info→scan_buffer* nelle posizioni *info→rear* e *info→rear+1*, nel vettore *info→samples_count* si dovranno avere il valore 2 nella posizione *info→rear*, e il valore 0 nella posizione *info→rear+1*; ciò indica, appunto, che quest'ultimo campione deriva dalla lettura precedente.

Infine, viene incrementato *info→rear* della quantità *read_count*, con eventuale azzeramento se si è arrivati all'ultimo elemento del buffer.

In questo modo, finché non verrà riempito il buffer, si avrà che l'indice *info→front* punterà sempre al primo elemento, mentre *info→rear* avanzerà di volta in volta della quantità *read_count*. Una volta riempito il vettore, si avrà che anche *info→rear* ritornerà a puntare al primo elemento; perciò, si fa scorrere in avanti l'indice *info→front* di un numero di posti pari agli elementi da scrivere; successivamente a partire da *info→rear*, verranno sovrascritti i valori prima presenti.

Un'ultima osservazione da fare riguarda il fatto che la coda circolare verrà effettivamente utilizzata solo nella parte di acquisizione precedente all'arrivo del TRIGGER, con un buffer di 20000 elementi. Invece, nella parte successiva, si creerà un buffer di 1200000 elementi, che corrispondono ai 20 minuti di acquisizione desiderata. In questo secondo caso non si raggiungerà mai la dimensione massima del buffer; quindi non vi sarà mai la sovrascrittura di vecchi elementi per aggiungerne di nuovi.

Questa funzione subirà un'ulteriore modifica per l'inserimento dei segnali di sincronizzazione; ciò verrà visto nel Capitolo 6, quando verrà discussa la fase di implementazione all'interno del programma completo presente nella Raspberry dell'End-Node.

Viene, poi, aggiunta, alla libreria `mcc172.c`, la funzione per il salvataggio dei dati. Essa è descritta dal Listato 5.14.

```
int mcc172_save_data(uint8_t address0, uint8_t address1)
{
    double dati_salvati[3][20000];
    uint16_t accumulatore[2][20000];
    char logname[64];
    FILE* logfile;
    uint32_t i;
    uint32_t k;

    //Open the csv file
    sprintf(logname, "/home/pi/measurements/Analisi_temporale_2HAT.csv");
    logfile = fopen(logname, "wt");
    fprintf(logfile, "CH0-DEV0(V),CH1-DEV0(V),CH0-DEV1(V),t-DEV0(ms),Acc0,Acc1\n" );

    //HAT0
    if(_devices[address0]->scan_info->front < _devices[address0]->scan_info->rear)
    {
        k=0;
        for (i=0; i < _devices[address0]->scan_info->rear; i+=2)
        {
            dati_salvati[0][k]=_devices[address0]->scan_info->scan_buffer[i];
            dati_salvati[1][k]=_devices[address0]->scan_info->scan_buffer[i+1];
            accumulatore[0][k]= _devices[address0]->scan_info->samples_count[i];
            k++;
        }
    }
    else
    {
        k=0;
        for (i= _devices[address0]->scan_info->front; i < _devices[address0]->scan_info->buffer_size; i+=2)
        {
            dati_salvati[0][k]=_devices[address0]->scan_info->scan_buffer[i];
            dati_salvati[1][k]=_devices[address0]->scan_info->scan_buffer[i+1];
            accumulatore[0][k]= _devices[address0]->scan_info->samples_count[i];
            k++;
        }
        for (i=0; i < _devices[address0]->scan_info->rear; i+=2)
        {
            dati_salvati[0][k]=_devices[address0]->scan_info->scan_buffer[i];
            dati_salvati[1][k]=_devices[address0]->scan_info->scan_buffer[i+1];
            accumulatore[0][k]= _devices[address0]->scan_info->samples_count[i];
            k++;
        }
    }

    //HAT1
    if(_devices[address1]->scan_info->front < _devices[address1]->scan_info->rear)
    {
        for (i=0; i < _devices[address1]->scan_info->rear; i++)
        {
            dati_salvati[2][i]=_devices[address1]->scan_info->scan_buffer[i];
            accumulatore[1][i]= _devices[address1]->scan_info->samples_count[i];
```

```
            }
        }
        else
        {
            k=0;
            for (i= _devices[address1]->scan_info->front; i < _devices[address1]->scan_info->buffer_size; i++)
            {
            dati_salvati[2][k]=_devices[address1]->scan_info->scan_buffer[i];
            accumulatore[1][k]= _devices[address1]->scan_info->samples_count[i];
            k++;
            }
            for (i=0; i < _devices[address1]->scan_info->rear; i++)
            {
                dati_salvati[2][k]=_devices[address1]->scan_info->scan_buffer[i];
                accumulatore[1][k]= _devices[address1]->scan_info->samples_count[i];
                k++;
            }
        }

        for (i=0; i < _devices[address1]->scan_info->buffer_depth; i++)
        {
            fprintf(logfile, "%f,%f,%f,%d,%d,%d\n", dati_salvati[0][i] ,dati_salvati[1][i], dati_salvati[2][i], i,
                        accumulatore[0][i], accumulatore[1][i]);
        }


        fclose(logfile);

        return RESULT_SUCCESS;

    }
```

Listato 5.14. Funzione *mcc172_save_data()* aggiunta all'interno della libreria `mcc172.c`

Innanzitutto, nella funzione, vengono create due matrici, ovvero *dati_salvati* e *accumulatore*, nelle quali vengono memorizzati, rispettivamente, le accelerazioni misurate e il numero di campioni trasferiti, ad ogni lettura, dalla MCC172 alla Raspberry.

Tali valori vengono presi dai vettori *info→scan_buffer* e *info→samples_count*. In particolare, le componenti che identificano le prime due accelerazioni vengono prese dalla HAT con indirizzo 0, mentre la terza accelerazione viene presa dalla HAT con indirizzo 1. Invece, in *accumulatore*, vengono salvati il numero di campioni acquisiti in ciascuna lettura; in particolare, nella prima riga vi sono i dati relativi alla prima HAT, mentre nella seconda vi sono i dati relativi alla seconda HAT. In questo caso, in un comportamento ideale, si vorrebbe che, per ciascuna lettura, vengano acquisiti due campioni dalla prima HAT (uno per canale) e un campione dalla seconda HAT.

Poiché i due vettori da cui bisogna estrarre i dati sono delle code circolari, bisogna distinguere diversi casi nella copia dei campioni per entrambe le HAT. Il primo, quello più semplice, è quando non si è arrivati a riempire completamente il buffer; perciò basta partire dall'indice 0 e arrivare all'indice *info→rear*, per effettuare la copia di tutti i campioni in maniera ordinata. Il secondo caso è, invece, quello in cui si è riempito l'intero buffer e, perciò, il campione più vecchio non si trova nel primo elemento del vettore. Si parte, così, per la copia, dall'elemento con indice *info→front*, fino ad arrivare all'ultimo elemento del vettore; si riprende, poi, dall'indice 0 fino ad arrivare all'indice *info→rear*. In questo modo si sono copiati tutti i campioni, in maniera ordinata rispetto all'istante di acquisizione, nella matrice *dati_salvati*.

Infine, si salvano le accelerazioni sui 3 assi e si salva il numero di campioni acquisiti in ciascuna lettura per entrambe le schede su un file `.csv` di prova, con percorso e nome uguale a `home/pi/measurements/Analisi_temporale_2HAT.csv`.

Questa funzione, nell'implementazione finale fatta all'interno dell'End-Node, subirà piccole modifiche, per permettere di caricare in un unico file `.csv` l'ac-

quisizione di 20 secondi antecedente all'arrivo del TRIGGER, e quella di 20 minuti successiva. Verranno, a tale scopo, definite nel Capitolo 6, analogamente a quanto mostrato nel Capitolo 4, le due funzioni *mcc172_save_data_pre_t()* e *mcc172_save_data_post_t()*.

Le ultime modifiche fatte riguardano le funzioni usate dopo aver fermato l'acquisizione per chiudere il collegamento con la specifica HAT, ovvero *mcc172_a_in_scan_cleanup()* e *mcc172_close()*. Non appena invocata la funzione *mcc172_a_in_scan_stop()*, vengono immediatamente richiamate le due funzioni sopra elencate:

- la *mcc172_a_in_scan_cleanup()* viene chiamata per "valorizzare" a *True* il parametro *info→stop_thread*, e liberare la memoria di *info→scan_buffer* e della struttura *info*;
- la *mcc172_close()* viene chiamata per chiudere la comunicazione SPI con la MCC172 e liberare l'area di memoria di *_devices*.

La modifica che viene fatta riguarda lo spostamento della liberazione di memoria di *info→scan_buffer* e della struttura *info* su *mcc172_close()*. In particolare, viene fatto ciò perché prima del salvataggio dei dati è necessario inviare il segnale di stop alla MCC172 e "valorizzare" la variabile *info→stop_thread*; invece, non si può liberare la memoria proprio perché si devono ancora copiare quei dati. Inoltre, sempre su *mcc172_close()*, viene aggiunta la liberazione della memoria del nuovo vettore aggiunto, ovvero *info→samples_count*. Infine, sempre in questa funzione, viene aggiunta la riconfigurazione del clock della scheda, che viene di nuovo posto con sorgente *SOURCE_LOCAL*, lasciando, in questo modo, la scheda pronta per qualsiasi tipo di acquisizione futura. La nuova funzione *mcc172_close()* è visibile nel Listato 5.15.

```
int mcc172_close(uint8_t address)
{
    int result;
    if (!_check_addr(address))
    {
        return RESULT_BAD_PARAMETER;
    }

    mcc172_a_in_scan_cleanup(address);
    free(_devices[address]->scan_info->scan_buffer);
    free(_devices[address]->scan_info->samples_count);
    free(_devices[address]->scan_info);
    _devices[address]->scan_info = NULL;
    result= mcc172_a_in_clock_config_write(address, SOURCE_LOCAL, 1000);
    if(result != RESULT_SUCCESS)
        {
                fprintf(stderr, "\nError: %s\n", hat_error_message(result));
        }
    _devices[address]->handle_count--;
    if (_devices[address]->handle_count == 0)
    {
        pthread_mutex_destroy(&_devices[address]->scan_mutex);
        close(_devices[address]->spi_fd);
        free(_devices[address]);
        _devices[address] = NULL;
    }

    return RESULT_SUCCESS;
}
```

Listato 5.15. Modifica della funzione *mcc172_close()* all'interno della libreria `mcc172.c`

A questo punto, non rimane che testare la nuova libreria, in cui è stata tolta la doppia copia dei campioni. Per effettuare il test, viene utilizzato il solito programma con 2 HAT, visto nella Sezione 5.3. In questo caso, però, una volta terminata la fase di inizializzazione e attivata la funzione *mcc172_a_in_scan_start()*, non viene

attivato il loop di lettura, ma si cicla senza fare niente finché non viene premuto il tasto *enter*. Viene, così, fermata l'acquisizione e vengono salvati i dati sul file `.csv` prima definito. Il codice da inserire dopo la funzione *mcc172_a_in_scan_start()* è quello mostrato nel Listato 5.16.

```
while (!enter_press());

for (device = 0; device < DEVICE_COUNT; device++)
{
       print_error(mcc172_a_in_scan_stop(address[device]));
       print_error(mcc172_a_in_scan_cleanup(address[device]));
}

//save data with the new function
print_error(mcc172_save_data(address[0],address[1]));

for (device = 0; device < DEVICE_COUNT; device++)
{
       print_error(mcc172_close(address[device]));
}
```

Listato 5.16. Codice da inserire dopo la funzione *mcc172_a_in_scan_start()* per testare la modifica della libreria `mcc172.c`

Con l'ausilio di Python, vengono visualizzati i campioni che sono trasferiti in ciascuna lettura, dalla MCC172 alla Raspberry. I risultati sono mostrati nella Figura 5.6.

Come si può vedere, si ottiene un comportamento che è prossimo a quello desiderato; esso è, appunto, pari a 2 campioni inviati in ciascuna lettura per la HAT0 (uno per canale) e 1 campione per la HAT1.

In un'acquisizione di circa 3 minuti, solo un paio di letture per scheda vengono fatte male, mentre, per tutto il resto del tempo si ha un funzionamento perfetto. Inoltre, nel confronto con gli altri test, vi è da notare, anche, la durata dell'acquisizione; infatti, in questo caso, come già detto, essa è più di 3 minuti, mentre, per gli altri test, non superava i 10 secondi, tempo entro il quale si osservava già una prestazione non buona.

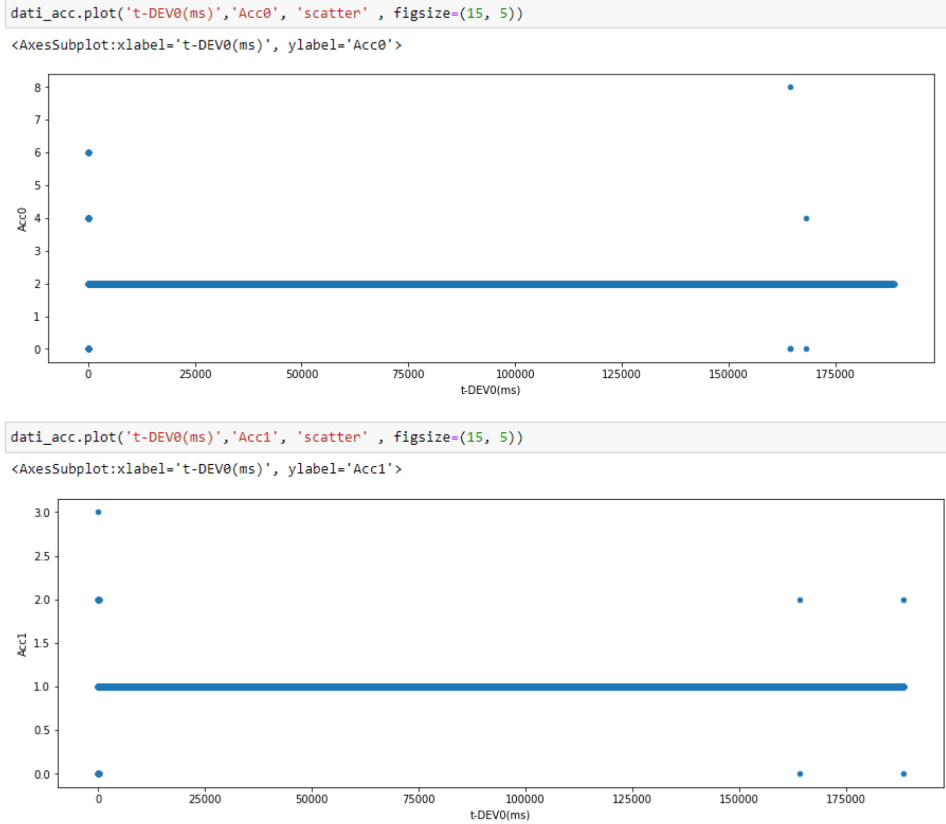


Figura 5.6. Campioni inviati dalla MCC172 alla Raspberry in ciascuna lettura, per entrambe le HAT, utilizzando la versione Real Time del sistema operativo Raspbian Lite e la versione modificata della libreria `mcc172.c`

6

Implementazione della componente software sull'End-Node

In questo capitolo verrà discussa l'ultima fase di sviluppo dell'End-Node. In particolare, dopo aver terminato le modifiche al sistema operativo e ai driver della MCC172, ottenendo un comportamento ottimale, si passa alla sua implementazione all'interno del programma finale, con l'inserimento dei segnali di sincronizzazione nel thread di acquisizione, e di due funzioni per il salvataggio dei dati antecedenti e successivi all'arrivo del TRIGGER, al fine di creare un unico file `.csv`*, rinominato con i corretti dati relativi all'acquisizione. Infine, verrà velocemente descritto lo script Python utilizzato per caricare il file* `.csv` *sull'FTP server.*

6.1 Aggiunta dei segnali di sincronizzazione

Come già accennato nel capitolo precedente, si deve, ora, utilizzare la libreria `mcc172.c`, con le modifiche apportate, all'interno del software finale.

L'implementazione del codice presente sulla Raspberry, dedicato all'acquisizione delle accelerazioni, era stata già mostrata nel Capitolo 4, nei Listati 4.6, 4.7 e 4.8.

Anche le funzioni che vengono richiamate sono state già ampiamente descritte.

Per l'implementazione finale del codice, secondo gli obiettivi iniziali, manca l'aggiunta dei segnali di sincronizzazione, che viene effettuata direttamente all'interno di *_scan_thread()*.

In particolare, nella struttura *mcc172ScanThreadInfo*, viene aggiunto il puntatore ad intero *uint16_t* time*, che verrà usato per costruire il vettore che contiene il conteggio dei BEACON. Esso verrà allocato in memoria all'interno della funzione *mcc172_a_in_scan_start()*, come avveniva già per i vettori *info→scan_buffer* e *info→samples_count*. Le righe di codice aggiunte sono mostrate nel Listato 6.1.

```
// allocate the time vector
info->time = (uint16_t*)calloc(1, info->buffer_size * sizeof(uint16_t));
if (info->time == NULL)
{
    // can't allocate memory
    free(info);
    dev->scan_info = NULL;
    return RESULT_RESOURCE_UNAVAIL;
}
```

Listato 6.1. Allocazione in memoria per il vettore *info→time* all'interno di *mcc172_a_in_scan_start()*

All'arrivo del TRIGGER e, quindi, all'inizio dell'acquisizione di 20 minuti, nella funzione *Check_Acquisition()* viene impostato uguale ad 1 il flag *beacon_event_flag*; la stessa cosa viene eseguita all'arrivo di ogni BEACON, all'interno del *UART_thread*. Così, questo flag viene posto uguale ad 1 ogni volta che nel canale UART della Raspberry arriva il segnale di TRIGGER o uno di BEACON.

La variabile *beacon_event_flag* è stata dichiarata nel file `main.c`, ed è quindi globale per l'intero programma. Poiché il file `main.c` include la libreria `mcc172.h`, all'interno di quest'ultima sarà possibile riprendere le variabili dichiarate nel primo file (attraverso l'istruzione *extern*). Verrà usata proprio questa variabile per inserire il conteggio dei BEACON sul vettore *info→time*.

Nonostante le due schede siano sincronizzate, è bene inserire il segnale di sincronizzazione per entrambe. Perciò saranno istanziati due vettori *info→time*, uno per HAT, i quali avranno i seguenti valori per ciascun loro elemento:

- uguale a *beacon_count*, se in tale campione è arrivato il segnale di BEACON sul canale UART;
- uguale a 0 in tutti gli altri casi.

Il contatore *beacon_count* viene anch'esso dichiarato nel file `main.c` e, quindi, è utilizzabile in tutto il programma. Infatti, esso viene inizializzato ad 1 prima di avviare l'acquisizione e, all'arrivo di ogni BEACON, viene incrementato di uno.

Per inserire tale contatore all'interno dei due vettori *info→time*, è stato usato il codice mostrato nel Listato 6.2.

```
if (read_count > 0)
{
        // handle wrap at end of buffer
    if ((info->buffer_size - info->rear) < read_count)
    {
        read_count = (info->buffer_size - info->rear);
    }
    if (info->buffer_size == info->buffer_depth)
    {
        info->front= (info->front + read_count) % info->buffer_size;
        info->buffer_depth= info->buffer_depth - read_count;
    }

    if ((error = _a_in_read_scan_data(address, read_count,
        scaled, calibrated,
        &info->scan_buffer[info->rear])) ==
        RESULT_SUCCESS)
    {
        info->samples_count[info->rear]=read_count;
        pthread_mutex_lock(&beacon_mutex);
        if (beacon_event_flag > 0)
        {
            if (beacon_event_flag==1)
            {
                info->time[info->rear]=beacon_count;
                beacon_event_flag++;
                beacon_address=address;
            }
            else if (beacon_event_flag==2 && beacon_address!=address)
            {
                info->time[info->rear]=beacon_count;
```

```
                beacon_event_flag = 0;
                beacon_count ++;
            }
        }
        pthread_mutex_unlock(&beacon_mutex);
        info->rear = (info->rear + read_count);
        info->buffer_depth= info->buffer_depth + read_count;
        if (info->buffer_size<50000)
        {
            while(read_count>1)
            {
                info->samples_count[info->rear-read_count+1]= 0;
                read_count--;
            }
            info->rear = info->rear % info->buffer_size;
        }
    }
}
```

Listato 6.2. Codice aggiunto a *_scan_thread* per la scrittura del vettore *info→time*

In aggiunta alla versione mostrata nel Listato 5.13, nel caso in cui *read_count* è maggiore di 0, ovvero quando può essere fatta una lettura, viene aggiunta la parte di scrittura del vettore *info→time*. Innanzitutto, viene utilizzato un mutex per permettere la gestione del *beacon_event_flag* e del *beacon_count*. Infatti, le due variabili appena nominate sono manipolate da entrambe le istanze di *_scan_thread* create; quindi, risulta necessario permettere la loro modifica solo per un thread alla volta.

Non sapendo quale dei due thread accederà per primo alle risorse condivise, essendo i due eseguiti in parallelo, si gestisce la variabile *beacon_event_flag* come di seguito specificato:

- Quando essa viene posta uguale ad 1, il primo thread che la legge inserisce, nel vettore *info→time* corrispondente, il valore di *beacon_count*, nello stesso indice in cui vengono inseriti i campioni, ovvero *info→rear*; inoltre, viene incrementato *beacon_event_flag* a 2. Infine, viene salvato, all'interno della variabile *beacon_address*, il valore dell'indirizzo della scheda che ha letto per prima il flag. La variabile *beacon_address* è stata dichiarata a livello globale all'interno della libreria, perciò, essa è condivisa da entrambi i thread.
- Quando essa è uguale a 2 e l'indirizzo della scheda è diverso da quello salvato in *beacon_address*, ovvero quella in questione non è la stessa HAT che ha valutato il *beacon_event_flag* quando era uguale ad 1, viene inserito, nel vettore *info→time* corrispondente, il valore di *beacon_count*, nello stesso indice nel quale vengono inseriti i campioni, ovvero *info→rear*. Così, entrambi i vettori hanno salvato, nell'istante di arrivo del BEACON, lo stesso valore di *beacon_count*. Infine, viene azzerato il valore di *beacon_event_flag* e viene incrementato di 1 quello di *beacon_count*, per poter, così, salvare i dati all'arrivo del prossimo BEACON.

È stata utilizzata questa logica perché non si può sapere quale dei due thread accederà alla variabile per primo, e può succedere che addirittura uno stesso thread legga due volte la variabile *beacon_event_flag* mentre l'altro non la legga mai, poiché essi sono in parallelo, e quindi creano indeterminismo sull'ordine globale di esecuzione. Con i mutex si obbliga a far accedere un thread alla volta al codice di gestione del BEACON e, con le condizioni *if*, si fa in modo che entrambi i thread salvino lo stesso valore di *beacon_count*.

La variabile mutex *beacon_mutex* è stata dichiarata a livello globale nella libreria. Inoltre, sono state definite due nuove funzioni, *init_beacon_mutex()* e *de-*

stroy_beacon_mutex(). Le due funzioni, utilizzate nel thread *Measurement* per inizializzare e distruggere il *beacon_mutex* all'inizio e alla fine di ogni acquisizione, sono soltanto due funzioni di interfaccia, per richiamare le funzioni *pthread_mutex_init()* e *pthread_mutex_destroy()*, visibili solo nello scope della libreria.

L'implementazione è descritta dal Listato 6.3.

```
pthread_mutex_t beacon_mutex;
void init_beacon_mutex(void)
{
        pthread_mutex_init(&beacon_mutex, NULL);
}

void destroy_beacon_mutex(void)
{
        pthread_mutex_destroy(&beacon_mutex);
}
```

Listato 6.3. Dichiarazione delle funzioni per la gestione del *beacon_mutex* all'interno della libreria `mcc172.c`

6.2 Aggiunta delle funzioni per il salvataggio dei dati nel pre e post TRIGGER

Nel Capitolo 5 era stata presentata solo una funzione di prova per il salvataggio dei dati.

Nell'implementazione del programma finale, è necessario rinominare il file `.csv` con il giusto nome, la cui struttura è stata già presentata nella Sezione 4.4.

Inoltre, all'interno dello stesso file, dovranno essere presenti le due acquisizioni eseguite, ovvero quelle precedente e successiva all'arrivo del TRIGGER.

Quindi, verranno definite due funzioni, ovvero *mcc172_save_data_pre_t()* e *mcc172_save_data_post_t()*.

La prima è praticamente uguale alla funzione *mcc172_save_data* descritta nel Capitolo 5. L'unica differenza è che, in questo caso, basta salvare i dati relativi alle acquisizioni sulle matrici *dati_salvati* e *accumulatore*, senza salvarli su un file `.csv`. Ovviamente, per rendere disponibili questi valori nella funzione *mcc172_save_data_post_t()*, le due matrici sono dichiarate a livello globale nella libreria. Inoltre, viene anche dichiarata una variabile *depth* per salvare il numero di elementi che erano stati acquisiti nella scansione precedente al TRIGGER, che verrà usata nella funzione *mcc172_save_data_post_t()* per la scrittura su file.

La funzione *mcc172_save_data_post_t()* è riportata nel Listato 6.4.

```
int mcc172_save_data_post_t(uint8_t address0, uint8_t address1)
{
    extern char template[50];
    extern char proj_code[50];
    extern struct tm *tm;
    extern rasp_ids ids;
    extern double dati_salvati[3][20010];
    extern uint16_t accumulatore[2][20010];
    extern uint32_t depth;
    FILE *fpt;
    char filename[200];
    char buffer_time[26];
    // Format the file name to be able to retrieve the information for uploading to FTP in Python
    strftime(buffer_time, 26, "%Y-%m-%dT%H_%M_%SZ", tm);
    sprintf(filename, "/home/pi/measurements/_%s-%s!%s&%s.csv",proj_code, template, ids.rasp_id, buffer_time);

    //Open the csv file
```

```
    fpt = fopen(filename, "w+");
    fprintf(fpt, "CH0-DEV0(V),CH1-DEV0(V),CH0-DEV1(V),t-DEV0(ms),Acc0,Acc1,time0,time1\n" );

    for (uint32_t i=0; i < depth; i++)
    {
        fprintf(fpt, "%f,%f,%f,%d,%d,%d,%d,%d\n", dati_salvati[0][i], dati_salvati[1][i], dati_salvati[2][i], i,
                accumulatore[0][i], accumulatore[1][i], 0, 0);
    }
    for(uint32_t i=0; i< _devices[address1]->scan_info->rear; i++)
    {
        fprintf(fpt, "%f,%f,%f,%d,%d,%d,%d,%d\n", _devices[address0]->scan_info->scan_buffer[2*i], _devices[
                address0]->scan_info->scan_buffer[2*i+1], _devices[address1]->scan_info->scan_buffer[i], i+depth,
                _devices[address0]->scan_info->samples_count[2*i], _devices[address1]->scan_info->samples_count[i],
                 _devices[address0]->scan_info->time[2*i], _devices[address1]->scan_info->time[i]);
    }
    fclose(fpt);

    return RESULT_SUCCESS;

}
```

Listato 6.4. Definizione della funzione *mcc172_save_data_post_t()* all'interno della libreria `mcc172.c`

Nella funzione, come prima cosa, vengono recuperati i dati arrivati tramite UART che dovranno essere usati per determinare il nome del file, ovvero il codice del template (*template*), il codice del progetto (*proj_code*), l'ID della Raspberry (contenuto nella struttura *ids*, nel campo *rasp_id*) e il puntatore *tm*, nel quale erano salvati la data e l'ora dell'istante di inizio acquisizione. Inoltre, vengono anche recuperati i dati salvati dalla funzione *mcc172_save_data_pre_t()*, ovvero *dati_salvati* e *accumulatore*.

Successivamente, viene determinato il nome del file, con formattazione già descritta nella Sezione 4.4, creato all'interno della cartella `/home/pi/measurements/`

Infine, vengono eseguiti due cicli; il primo è necessario per salvare i dati precedenti all'arrivo del TRIGGER, mentre il secondo serve per salvare i dati dell'acquisizione appena finita, ancora presenti all'interno dei vettori *info→scan_buffer*, *info→samples_count* e *info→time*. Per questo secondo ciclo, sapendo che i vettori non verranno mai effettivamente riempiti, si avrà che l'indice *info→front* punterà al primo elemento dei vettori; quindi, si itera dall'indice 0 a *info→rear* per salvare i dati in maniera ordinata rispetto al tempo di acquisizione.

6.3 Script per il caricamento del `.csv` sull'FTP server

L'ultimo passaggio rimasto, per implementare il software finale all'interno dell'End-Node, è il caricamento del file `.csv` sull'FTP server. Per eseguire ciò, all'interno del thread *Measurement*, come ultima operazione, una volta terminati l'acquisizione e il salvataggio dei dati, viene inserita la riga di codice presente nel Listato 6.5, per attivare lo script Python che si occuperà del caricamento.

```
//run the upload script
system("sudo python3 data_upload.py &");
```

Listato 6.5. Riga di codice aggiunta nel thread *Measurement* per attivare lo script Python che si occuperà del caricamento del file

Il contenuto dello script `data_upload.py` è mostrato nel Listato 6.6.

```
import os
import time
import csv
from ftplib import FTP
import os
import re

file_path = '/home/pi/measurements/'
ftp=FTP('192.168.105.224')
ftp.login('smartspace','sm4rtsp4c3')


os.chdir(file_path)
records = os.listdir()


if(len(records) != 0):
       name_file = records[0]
       id_rasp = re.search('!(.+?)&', name_file).group(1)
       project_code = re.search('_(.+?)-', name_file).group(1)
       template_code = re.search('-(.+?)!', name_file).group(1)
       timestamp = re.search('&(.+?)Z', name_file).group(1)
       #check if directory exists
       path1 = project_code
       ftp.cwd(path1)
       if template_code in ftp.nlst():
          path = template_code+'/'+id_rasp
          new_name = timestamp +'Z'+".csv"
          name_temp_file = timestamp +'Z'+".txt"
          os.rename( file_path+name_file,file_path+new_name)
          ftp.cwd(path)
          with open(file_path+new_name, 'rb') as file:
             ftp.storbinary('STOR '+ os.path.basename(new_name), file)
          with open(file_path+name_temp_file, 'w') as tmpfile:
             tmpfile.write('tmp')
          with open(file_path+name_temp_file, 'rb') as tmpfile:
             ftp.storbinary('STOR '+ name_temp_file, tmpfile)
          os.remove(name_temp_file)
          os.remove(new_name)
       else:
          os.remove(new_name)
```

Listato 6.6. Script `data_upload.py` usato per il caricamento del file sull'FTP server

Nel codice viene utilizzata la libreria `FTP`, che implementa la classe lato Client nel protocollo FTP. Innanzitutto, si determina l'indirizzo IP dell'FTP server, e si impostano le credenziali di accesso, ovvero nome utente e password. Inoltre, viene anche indicata la cartella dalla quale verrà selezionato il file.

Successivamente dal nome del file vengono estratti i 4 campi inseriti, ovvero l'ID della Raspberry, il codice del progetto, il codice del template e il timestamp. Attraverso il comando *ftp.cwd()* viene aperta la directory nel server con nome uguale al codice del progetto. Si cerca, poi, se, all'interno di tale directory, è presente la cartella con lo stesso nome del codice del template dell'acquisizione che si sta salvando; se esso non esiste, viene fermato l'upload del file. Invece, se esso è presente, si rinomina il file `.csv` con solo il nome del timestamp seguito dal carattere "Z". Successivamente, si accede alla cartella con percorso `template_code/id_rasp` sull'FTP server.

Poiché, il trasferimento del file deve essere fatto in modalità binaria, viene creato e scritto il file di testo `.txt`, con lo stesso nome del file `.csv`. Infine, con il metodo *storbinary()*, si avvia il trasferimento del file da un client FTP a un server FTP utilizzando il comando *STOR*, nella directory in cui si è effettuato l'accesso.

6.4 Configurazione in servizi e librerie da installare nell'End-Node

Per quanto riguarda la board STM32 dell'End-Node, basterà caricare il progetto descritto nella Sezione 4.1 all'interno del microcontrollore; ogni volta che la scheda verrà alimentata, tale programma andrà automaticamente in esecuzione.

Invece, nella Raspberry, è necessario svolgere prima alcune operazioni. Innanzitutto, è necessario creare un `Makefile` all'interno del percorso `home/pi/progetto-finale/C_Project-End_Node`, che si occuperà della compilazione del programma e di collegare le giuste librerie, creando il codice eseguibile `mcc172_test`. Il codice inserito in tale file è quello mostrato nel Listato 6.7.

```
#DIR_FONTS = ./Fonts
DIR_OBJ = ./obj
DIR_BIN = ./bin

OBJ_C = $(wildcard ${DIR_OBJ}/*.c)
OBJ_O = $(patsubst %.c,${DIR_BIN}/%.o,$(notdir ${OBJ_C}))

TARGET = mcc172_test
#BIN_TARGET = ${DIR_BIN}/${TARGET}

CC = gcc

DEBUG = -g -O0 -I/usr/local/include -Wall
CFLAGS += $(DEBUG)

LIB = -ldaqhats -lm -lwiringPi -lpthread -lconfig

${TARGET}:${OBJ_O}
	$(CC) $(CFLAGS) $(OBJ_O) -o $@ $(LIB)

${DIR_BIN}/%.o : $(DIR_OBJ)/%.c
	$(CC) $(CFLAGS) -c $< -o $@ $(LIB)

clean :
	rm $(DIR_BIN)/*.*
	rm $(TARGET)
```

Listato 6.7. Codice presente all'interno del `Makefile`

Inoltre, sono stati configurati dei servizi, come è stato fatto anche con la Raspberry presente nel Gateway. In particolare, i servizi creati sono due:

- Il *wvdial*, per la gestione della connessione alla rete internet tramite il Modem Multitech 4G; la sua configurazione è la stessa utilizzata nel Gateway, già mostrata nel Listato 2.6.
- Il *node-accelerometro*, utilizzato per l'avvio del codice di acquisizione; la sua configurazione è mostrata nel Listato 6.8.

```
nodo-acceleometro.service

[Unit]
Description=Raspberry Node Accelerometer

[Service]
ExecStart=/home/pi/progetto-finale/C_Project-End_Node/mcc172_test
WorkingDirectory=/home/pi/progetto-finale/C_Project-End_Node
StandardOutput=journal
StandardError=inherit
Restart=always
RestartSec=5
User=boot

[Install]
WantedBy=multi-user.target
```

Listato 6.8. Configurazione del servizio *node-accelerometro*

Prima di poter avviare il codice di acquisizione, è necessario procedere all'installazione di alcune librerie; ciò avviene tramite il comando *sudo apt-get install*. Queste sono:

- La libreria `wiringpi`, necessaria per gestire la comunicazione UART della Raspberry.
- La libreria `libconfig`, utilizzata per processare file di configurazione strutturati; in questo caso, il file da processare è `config.cfg`, descritto nel Capitolo 4.
- La libreria `daqhats`; per installare quest'ultima libreria è necessario, dapprima, installare `git`. L'intera installazione è mostrata nel Listato 6.9

```
sudo apt install git
cd ~
git clone https://github.com/mccdaq/daqhats.git
cd ~/daqhats
sudo ./install.sh
```

Listato 6.9. Installazione della libreria `daqhats`

Prima di procedere con l'ultima riga del codice, ovviamente, bisogna sostituire i file `mcc172.c` e `mcc172.h` precedentemente modificati, rispettivamente nei percorsi `/daqhats/lib` e `/daqhats/include`.

Infine, bisogna disattivare il bluetooth della Raspberry. Infatti, esso è configurato come l'UART primario che, invece, deve essere utilizzato per la comunicazione con la STM32. Tale operazione viene svolta tramite il codice mostrato nel Listato 6.10 e riavviando il sistema.

```
sudo nano/boot/config.txt
dt-overlay=disable-bt
```

Listato 6.10. Disabilitazione del bluetooth

7

Testing del sistema

In questo capitolo verranno presentati dei test del sistema completo; in particolare, verrà mostrato il risultato di un'acquisizione schedulata dalla piattaforma ALGIZ.STRUCT, eseguita con un solo End-Node per motivi pratici di disponibilità di risorse. Verrà, inoltre, mostrato il numero di campioni inviati in ciascun istante di acquisizione, valutando, così, le prestazioni della comunicazione tra MCC172 e Raspberry, a conferma di quanto sviluppato durante la fase di progettazione ed implementazione.

7.1 Avvio dell'acquisizione da ALGIZ.STRUCT

Come già spiegato in precedenza, la configurazione di un'acquisizione inizia dalla piattaforma ALGIZ.STRUCT, con la selezione del template che si vuole eseguire. Lo sviluppo completo della piattaforma, comprensivo di front-end e back-end, non è stato effettuato in questo progetto, ma è stato sviluppato da un altro reparto aziendale; perciò, non è stata vista la sua analisi e non verrà nemmeno testata.

Per quanto riguarda il Gateway, una volta avviati tutti i servizi descritti nei Capitoli 2 e 3, esso viene messo in funzione per testare l'efficacia della comunicazione con l'End-Node, che dovrà, poi, compiere l'acquisizione vera e propria. Anche in questo caso non verranno mostrati tutti i test sull'inserimento, sull'aggiornamento e sulla cancellazione dei Gateway, degli End-Node e dei template, con le relative visualizzazioni di messaggi di conferma sullo schermo, poiché questa parte non è stata affrontata in maniera diretta in fase di progettazione.

Per iniziare, bisogna avviare uno o più template dalla piattaforma web (nel test viene avviavo il *TEMPLATE_T1*), dopo aver aggiunto le informazioni riguardanti i Gateway e gli End-Node da utilizzare. Infatti, possono essere avviati più template, poiché essi possono rappresentare sia acquisizioni schedulate, attraverso l'uso di una cron expression, sia acquisizioni ondemand. Il Gateway schedula tutte le acquisizioni da eseguire e, quando sarà il momento di avviarne una, esegue lo script `send_synch.py`. Come già affermato nel Capitolo 2, l'output di questo script viene direzionato sul file `output.txt`. In esso, viene subito stampata la lista degli End-Node coinvolti nell'acquisizione e, se non vi sono dei conflitti con altri template in esecuzione, parte la scansione, scrivendo, prima, sempre su `output.txt`, che non

sono stati scoperti dei conflitti tra i template in esecuzione e che l'acquisizione è stata avviata, e, poi, inoltrando, dal Gateway LoRaWAN, i messaggi all'End-Node.

Nel momento dell'invio del primo messaggio per l'avvio dell'acquisizione, ovvero quello con la durata del campionamento, il codice del template, la frequenza di acquisizione e la lista dei dispositivi associati al template, viene scritta sul file `output.txt` la stringa *Send samples*; successivamente, per ogni invio di uno dei tre segnali di sincronizzazione, viene scritto, rispettivamente, *Send trigger*, *Send beacon* e *Send end*.

Si può, quindi, testare l'effettivo funzionamento dell'acquisizione programmata all'interno del Gateway verificando se tale file di testo è stato scritto correttamente. In Figura 7.1 è possibile osservare quanto prodotto.

```
Lista devices template:
['1', '2']
Discovered conflicting templates:
False
Running acquisition!

Send samples
Send trigger
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send beacon
Send end
```

Figura 7.1. Output prodotto dallo script `send_synch.py` nel file `output.txt`

Invece, sul file `logfile.txt` sono registrate tutte le acquisizioni che vengono eseguite, salvando il timestamp di fine acquisizione, il codice del template che è stato appena eseguito, e se si trattava di un'acquisizione programmata o ondemand. Se tale template è già in esecuzione e viene riattivato, si procede alla segnalazione di questo fatto; invece, se tale template non può andare in esecuzione perché è in

conflitto con uno già avviato, viene scritta la stringa *Conflicting templates already running!*. Un esempio di quello che può essere aggiunto al file `logfile.txt` (esso viene aperto in scrittura in modalità *append*, andando così di volta in volta ad aggiungere nuove righe) dopo un'acquisizione è mostrato in Figura 7.2.

```
2021-11-17 10:45:41.158495
TEMPLATE_T1:
Template periodic acquisition executed
```

Figura 7.2. Output prodotto dallo script `send_synch.py` nel file `logfile.txt`

7.2 Avvio dell'applicazione su End-Node

La sola visualizzazione dell'output del Gateway non garantisce che i segnali di sincronizzazione vengano inviati nel modo corretto, ma verifica solo che i template ricevuti siano valutati correttamente e che il ciclo per l'invio dei segnali di sincronizzazione venga eseguito nel modo giusto.

Per valutare se l'invio dei segnali su rete LoRaWAN avviene in maniera appropriata, bisogna testare ciò che viene ricevuto negli End-Node. Ovviamente, bisognerà prima avviare sia la Raspberry che la STM32, mandando in esecuzione i programmi discussi nei capitoli precedenti.

Appena viene mandato in esecuzione il codice eseguibile `mcc172_test` nella Raspberry, vengono visualizzati su schermo i dati presenti sul file `config.cfg`, relativi al codice del progetto, all'ID del dispositivo e al low level ID; inoltre, vengono visualizzati i parametri della MCC172 relativi all'acquisizione appena avviata, che memorizza costantemente gli ultimi 20 secondi di campionamento. In Figura 7.3 è mostrato ciò che viene visualizzato.

```
proj_code loaded: UNIVPM

UID loaded: RASP_NODE_2

Low Level UID loaded: 2

UART correctly initialized!
    Requested scan rate: 1000.00
    Actual scan rate: 1003.00
    MCC 172 0:
      Address: 0
      Channels: 0, 1
    MCC 172 1:
      Address: 1
      Channels: 0
Misurando in attesa del trigger...
```

Figura 7.3. Output sullo schermo all'avvio dell'applicazione sulla Raspberry

Nell'STM32, ogni volta che viene ricevuto un messaggio via LoRaWAN, viene stampata la stringa *PACKET RECEIVED ON PORT x*, dove x indica, appunto,

la porta dove viene inviato il messaggio. Tale stringa non può essere visualizzata poiché non vi sono schermi connessi, a meno che non ci si connetta con il proprio PC, in maniera seriale, tramite l'utilizzo di Putty. A seconda del messaggio ricevuto, il comportamento dell'STM32 è diverso, come già spiegato nel Capitolo 4.

Il primo messaggio in arrivo all'End-Node è quello relativo alla creazione del device sulla piattaforma ALGIZ.STRUCT. La Raspberry riceverà il messaggio, inoltrato dalla STM32, contenente il codice del progetto, l'ID del dispositivo e il low level ID appena aggiornati, che verranno salvati sul file `config.cfg` e, di nuovo, scritti sullo schermo, come mostrato in Figura 7.4.

```
Salvo codice progetto e UID
UNIVPM
RASP_NODE_1
1
```

Figura 7.4. Output sullo schermo della Raspberry all'aggiunta dell'End-Node su ALGIZ.STRUCT

Come appena mostrato, prima dell'avvio può essere cambiato l'ID del dispositivo rispetto ad una vecchia acquisizione, che aveva salvato i propri valori sul file `config.cfg`.

Successivamente, si procede con l'invio dei dati di configurazione del template; questi contengono il codice del template, la frequenza di campionamento e la lista degli End-Node a cui è rivolto il messaggio. Tali informazioni vengono estrapolate, visualizzate ed, infine, si determina se tale End-Node viene attivato o meno. Ciò che viene visualizzato è mostrato in Figura 7.5.

```
Informazioni template in esecuzione: TEMPLATE_T1/1000/1,2
Device attivi:
 1

 2
ID Device: 1

Device attivo!
```

Figura 7.5. Output sullo schermo della Raspberry all'arrivo dei dati di configurazione del template

Infine, vengono inviati i segnali di sincronizzazione. All'arrivo del TRIGGER viene fermata l'acquisizione in corso e ne viene avviata una nuova, di durata prefissata sul template, che generalmente è di 20 minuti, e viene visualizzata sullo schermo la scritta *Trigger arrivato!*. A quel punto, vengono di nuovo visualizzati i parametri della MCC172, relativi alla nuova acquisizione, come mostrato in Figura 7.6.

All'arrivo di ogni BEACON sulla Raspberry, viene visualizzato il messaggio *Beacon arrivato!*, mentre all'arrivo dell'END viene mostrato il messaggio *End arrivato!*. Quando viene ricevuto quest'ultimo segnale, viene fermata l'acquisizione e ne vengono salvati i dati; questi comprendono, anche, quelli relativi ai 20 secondi precedenti all'arrivo del TRIGGER, per poi avviare una nuova acquisizione in attesa di un nuovo TRIGGER.

```
A
Receiveing Synch
Trigger arrivato!
front0 13176
 rear0 13176
 front1 6587
 rear1 6587
     Requested scan rate: 1000.00
    Actual scan rate: 1003.00
    MCC 172 0:
      Address: 0
      Channels: 0, 1
    MCC 172 1:
      Address: 1
      Channels: 0
Trigger arrivato!
```

Figura 7.6. Output sullo schermo della Raspberry all'arrivo del segnale di TRIGGER

In Figura 7.7 è possibile vedere cosa viene visualizzato all'arrivo di ciascun BEACON (in particolare, gli ultimi due) e all'arrivo dell'END, che comporta, anche, l'avvio di una nuova acquisizione pre-TRIGGER.

```
B
Receiveing Synch
Beacon arrivato!
B
Receiveing Synch
Beacon arrivato!
C
Receiveing Synch
End arrivato!
End arrivato!
    Requested scan rate: 1000.00
    Actual scan rate: 1003.00
    MCC 172 0:
      Address: 0
      Channels: 0, 1
    MCC 172 1:
      Address: 1
      Channels: 0
Misurando in attesa del trigger...
```

Figura 7.7. Output sullo schermo della Raspberry all'arrivo dei segnali di BEACON e di END

I dati vengono salvati all'interno del file `_UNIVPM-TEMPLATE_T1!RASP_NODE_1&-2021-11-17T10_25_41Z.csv`, con nome che identifica il codice del progetto, il codice del template, l'ID dell'End-Node, e il timestamp di arrivo del TRIGGER.

7.3 Presentazione del file `.csv` creato

Il file `.csv` creato contiene 8 colonne; le prime tre sono le accelerazioni misurate grazie alla MCC172 sui tre assi; poi, c'è il tempo trascorso in ms; subito dopo ci sono

le due colonne che indicano il numero di campioni inviati, da ciascuna MCC172 alla Raspberry, in ogni lettura; infine, ci sono le due colonne nelle quali viene inserito il conteggio dei BEACON nell'istante di arrivo, per ciascuna HAT. In Figura 7.8 vengono mostrate le colonne presenti, con le prime 5 righe del dataset.

```
dati_acc=pd.read_csv('_UNIVPM-TEMPLATE_T1!RASP_NODE_1&2021-11-17T10_25_41Z.csv')
dati_acc.head()
```

	CH0-DEV0(V)	CH1-DEV0(V)	CH0-DEV1(V)	t-DEV0(ms)	Acc0	Acc1	time0	time1
0	0.000286	-0.000019	0.001761	0	2	1	0	0
1	0.001854	0.001433	0.001426	1	2	1	0	0
2	0.002267	0.002026	0.001451	2	2	1	0	0
3	0.001597	0.001427	0.000534	3	2	1	0	0
4	0.001355	0.001203	-0.000084	4	2	1	0	0

Figura 7.8. Visualizzazione delle prime 5 righe del dataset

Poiché viene eseguita un'acquisizione di 20 minuti, quindi di 1200000 ms, vengono eseguite circa 1200000 letture (in realtà leggermente di più poiché si campiona con una frequenza effettiva di 1003Hz, come spiegato nel Capitolo 4), che sommate ai 20 secondi di acquisizione precedente all'arrivo del TRIGGER (quindi 20000 letture), formano un dataset di circa 1220000 righe, dove ogni riga indicherà, appunto, un istante di campionamento.

Attraverso l'utilizzo della libreria `Pandas` di Python, è possibile memorizzare tale dataset, trasformandolo in un DataFrame, ed effettuare una visualizzazione dei dati presenti. Nelle prime 3 colonne vengono salvate le tre tensioni analogiche proporzionali alle accelerazioni misurate. Come era stato già spiegato, la MCC172 opera un filtraggio alle basse frequenze, eliminando così i segnali continui; per tale ragione, i segnali avranno valore medio sempre pari a 0. Però, negli istanti nei quali vengono introdotte in maniera manuale delle vibrazioni al sistema, si avrà in ingresso una tensione diversa da 0, che sarà proporzionale all'ampiezza dell'accelerazione, mentre il segno ne determinerà il verso. I segnali acquisiti sono mostrati nelle Figure 7.9, 7.10 e 7.11, tenendo a mente che le accelerazioni lungo l'asse x sono misurate nel canale CH0 della prima HAT, quelle lungo l'asse y nel canale CH1 della prima HAT, e quelle lungo l'asse z nel canale CH0 della seconda HAT.

È interessante osservare il numero di campioni inviati, in ciascuna lettura, dalla MCC172 alla Raspberry. Idealmente, essi dovrebbero essere pari a 2 nella prima HAT (uno per canale) e pari a 1 nella seconda HAT. Nonostante le modifiche fatte al sistema operativo Raspbian e alla libreria `mcc172.c`, che permettevano un comportamento quasi ideale nel caso di esecuzione del solo codice di acquisizione, nel momento che tale codice viene implementato all'interno del programma completo della Raspberry, le prestazioni peggiorano leggermente, mantenendo, però, un comportamento del tutto soddisfacente. Questo leggero calo di prestazioni è dovuto al fatto che nella Raspberry vengono introdotti nuovi processi, come il thread per la gestione dell'UART, che aumentano il carico computazionale del processore e abbassano le prestazioni della comunicazione con le due MCC172. In Figura 7.12 sono mostrati il numero di campioni acquisiti in ciascuna lettura, per entrambe le board.

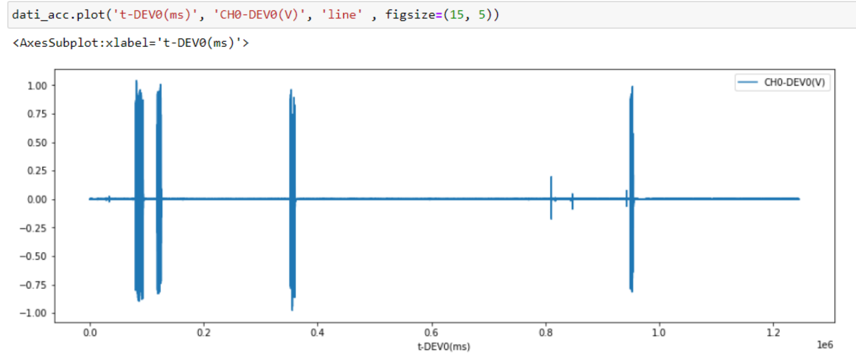


Figura 7.9. Visualizzazione della tensione analogica acquisita sul canale CH0 della HAT0, proporzionale all'accelerazione misurata sull'asse x

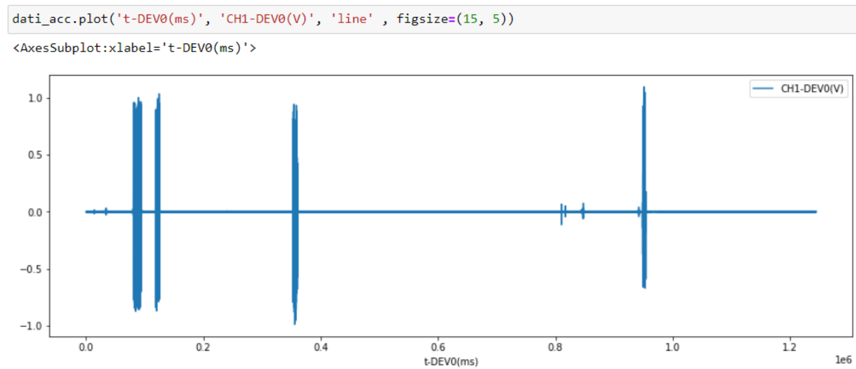


Figura 7.10. Visualizzazione della tensione analogica acquisita sul canale CH1 della HAT0, proporzionale all'accelerazione misurata sull'asse y

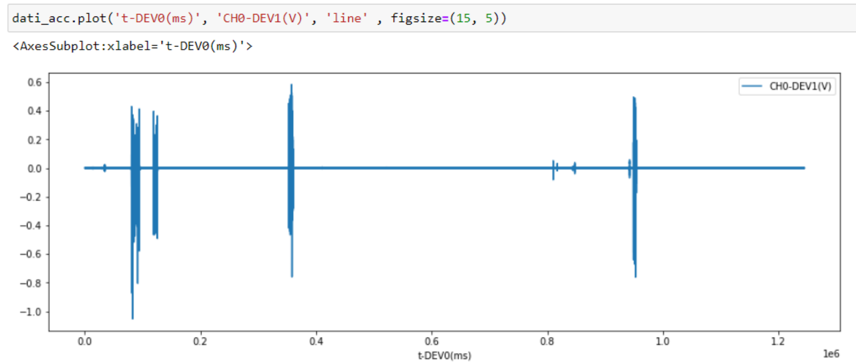


Figura 7.11. Visualizzazione della tensione analogica acquisita sul canale CH0 della HAT1, proporzionale all'accelerazione misurata sull'asse z

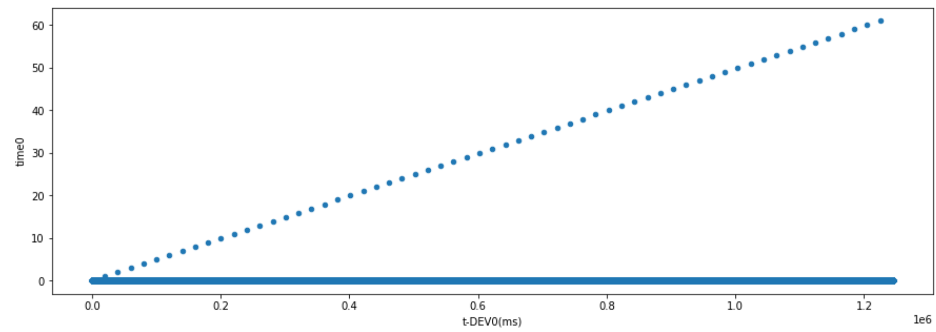
Figura 7.12. Numero di campioni inviati, in ciascuna lettura, dalla HAT0 e dalla HAT1 alla Raspberry

È possibile osservare che le prestazioni restano molto buone, anche perché non bisogna dimenticare che quella mostrata è un'acquisizione di più di 20 minuti, rispetto ai quasi 3 minuti di Figura 5.6; ciò implica che, nella durata complessiva dell'acquisizione, ci sia una quantità totale di campioni persi maggiore, che però viene distribuita su un arco temporale decisamente più lungo e risulta essere solo una piccolissima percentuale rispetto ai campioni totali (in questo caso essi sono circa l'1% dei campioni totali).

Infine, mancano da visualizzare solo le colonne che contengono, per entrambe le HAT e nell'istante di arrivo del BEACON, il conteggio di quest'ultimo. Poiché viene inviato un BEACON ogni 20 secondi e l'acquisizione dura 20 minuti, verranno inviati 60 BEACON; nella Figura 7.13 è possibile vedere l'esatto istante di arrivo di ciascuno di essi per entrambe le schede.

```
dati_acc.plot('t-DEV0(ms)', 'time0', 'scatter' , figsize=(15, 5))
```

```
<AxesSubplot:xlabel='t-DEV0(ms)', ylabel='time0'>
```




```
dati_acc.plot('t-DEV0(ms)', 'time1', 'scatter' , figsize=(15, 5))
```

```
<AxesSubplot:xlabel='t-DEV0(ms)', ylabel='time1'>
```

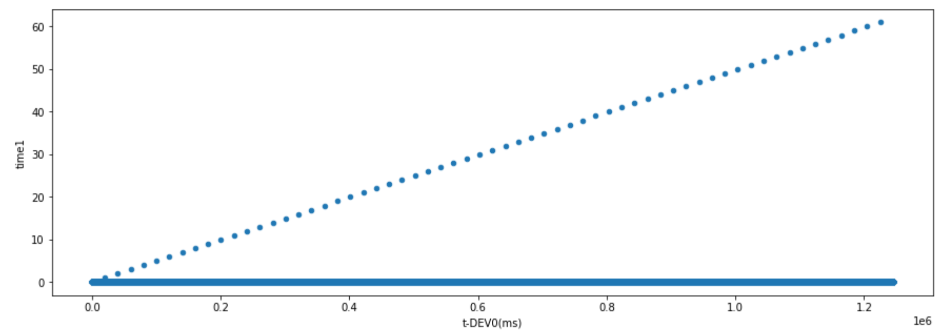


Figura 7.13. Visualizzazione del conteggio dei BEACON, determinando, per entrambe le HAT, l'esatto istante di campionamento di arrivo di quest'ultimo

Conclusioni

Nella presente tesi è stato presentato lo sviluppo dell'infrastruttura hardware e software utilizzata per l'acquisizione delle vibrazioni di un edificio. Quella che è stata sviluppata è una soluzione con costi contenuti, e che utilizza protocolli di comunicazione sicuri tra i dispositivi, poiché sono stati impiegati il protocollo MQTT e il protocollo LoRaWAN. Nella prima parte della tesi sono state analizzate le parti sviluppate da altri membri del team all'interno di Smart Space, azienda nell'ambito della quale è stato svolto il tirocinio. Esse riguardavano l'analisi del Gateway, l'analisi del protocollo MQTT, l'analisi del protocollo LoRaWAN e degli elementi principali che formano la sua architettura, e la configurazione dei parametri di Chirpstack.

Nella seconda parte si è passati allo sviluppo dell'End-Node, che è stato l'argomento principale trattato durante il tirocinio. Nello specifico, è stato analizzato l'intero codice presente sulla STM32, che si occupa della ricezione dei messaggi LoRaWAN e dell'inoltro di questi ultimi sul canale UART della Raspberry. Inoltre, è stato presentato l'intero codice in esecuzione nella Raspberry, che si occupa della ricezione dei messaggi e dell'avvio e del salvataggio delle acquisizioni.

Nella terza parte, sono state affrontate la progettazione e l'implementazione del software sull'End-Node, che doveva rispecchiare diverse specifiche prestazionali. In particolare, sono state riscontrate molte problematiche nell'utilizzo della MCC172, programmata per restituire blocchi di campioni ogni volta che si raggiunge un numero soglia prefissato di campioni acquisiti, e non un singolo campione alla volta, poiché la velocità di comunicazione tra la MCC172 e la Raspberry non risultava essere abbastanza elevata. Per permettere la restituzione di un campione alla volta, appena esso viene campionato, è stato necessario installare la patch Real-Time del sistema operativo Raspbian, in grado di configurare il kernel in Real-Time. Successivamente, è stata modificata la libreria fornita da MCC per la gestione della board MCC172, diminuendo notevolmente il carico computazionale della Raspberry e permettendo, così, una migliore comunicazione con la MCC172. Infine sono stati mostrati i test effettuati.

L'argomento di tirocinio, sviluppato nella presente tesi, si è rilevato molto interessante sotto diversi punti di vista. Innanzitutto, sono state affrontate tantissime tematiche del vasto mondo dell'ingegneria informatica e dell'automazione, come, ad esempio, la parte di telecomunicazioni, di sviluppo di firmware e di software em-

bedded e di utilizzo di sensori e schede di acquisizione ad alta precisione. Inoltre, in questo progetto, si è avuto modo di studiare e sperimentare nozioni del tutto nuove, che ci hanno anche aiutato a migliorare nelle metodologie di apprendimento autonomo.

Probabilmente, la soluzione presentata in questa tesi non sarà quella definitiva di Smart Space. Possibili modifiche potranno essere effettuate sull'acquisizione dei campioni, poiché, nonostante le modifiche apportate al kernel del sistema operativo della Raspberry e quelle effettuate sui driver della board MCC172, le prestazioni di comunicazione possono ancora migliorare. Possibili sviluppi potrebbero riguardare un'ulteriore modifica del sistema operativo della Raspberry, in particolare dello scheduler, per lasciare uno dei quattro processori libero di gestire la comunicazione SPI con la MCC172, senza dover compiere altri task.

Un'altra modifica potrebbe riguardare il modo con cui vengono attivate le acquisizioni delle accelerazioni. In particolare, se ne potrebbe avviare una quando viene rilevata una vibrazione anomala, che supera un determinato livello di soglia, anche se dal Gateway non è arrivato nessun segnale di TRIGGER.

Infine, si potrebbe optare per una soluzione con un più basso costo energetico, nella quale la Raspberry non è sempre accesa e in continua acquisizione. Nel dettaglio, il microcontrollore STM32, quando riceve il segnale con la richiesta di avviare un'acquisizione, potrebbe azionare un relè, posto tra l'alimentazione e la Raspberry, in grado di alimentare quest'ultima e permetterne l'avvio.

Riferimenti bibliografici

1. Anlog Devices, EVAL-ADXL356/EVAL-ADXL356. https://www.analog.com/media/en/technical-documentation/user-guides/EVAL-ADXL356-357-UG-1119.pdf, 2018.
2. STM32 LoRaWAN software expansion for STM32Cube. https://www.st.com/resource/en/data_brief/i-cube-lrwan.pdf, 2018.
3. STM32 LoRaWAN software expansion for STM32Cube: User Manual. https://comm.eefocus.com/media/download/index/id-1014169, 2018.
4. Diagnostica delle costruzioni e monitoraggio strutturale. https://it.wikipedia.org/wiki/Diagnostica_delle_costruzioni, 2019.
5. Modalità di trasmissione Multicast. https://it.wikipedia.org/wiki/Multicast, 2019.
6. Protocollo di comunicazione UART. https://it.wikipedia.org/wiki/UART, 2019.
7. An In-depth look at LoRaWAN Class A Devices. https://lora-developers.semtech.com/library/tech-papers-and-guides/lorawan-class-a-devices/, 2021.
8. An In-depth look at LoRaWAN Class B Devices. https://lora-developers.semtech.com/library/tech-papers-and-guides/lorawan-class-b-devices/, 2021.
9. An In-depth look at LoRaWAN Class C Devices. https://lora-developers.semtech.com/library/tech-papers-and-guides/lorawan-class-c-devices/, 2021.
10. MCC172, the IEPE Measurement DAQ HAT for Raspberry Pi. https://www.mccdaq.com/PDFs/specs/DS-MCC-172.pdf, 2021.
11. Modulazione LoRa e protocollo di comunicazione LoRaWAN. https://lora-developers.semtech.com/documentation/tech-papers-and-guides/lora-and-lorawan, 2021.
12. MQTT Essentials. https://www.hivemq.com/mqtt-essentials/, 2021.
13. Protocollo di comunicazione SPI. https://it.wikipedia.org/wiki/Serial_Peripheral_Interface, 2021.
14. The ChirpStack project. https://www.chirpstack.io/project/, 2021.
15. The things fundamentals on LoRaWAN. https://www.thethingsnetwork.org/docs/lorawan/, 2021.
16. Ferran Adelantado, Xavier Vilajosana, Pere Tuset-Peiro, Borja Martinez, Joan Melia-Segui, and Thomas Watteyne. *Understanding the limits of LoRaWAN.* IEEE Communications magazine, 2017.

17. M.T. Buyukakkaslar, M.A. Erturk, M.A. Aydin, and L. Vollero. *LoRaWAN as an e-Health Communication Technology.* IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), 2017.
18. D. Calvanese. *Programmazione con strutture dati in C.* Società Editrice Esculapio, 2020.
19. S.T. Dietrich and D. Walker. *The evolution of real-time linux.* 7th RTL Workshop, 2005.
20. A. De Gloria. *Fondamenti di progettazione elettronica analogica e digitale.* Franco Angeli, 2002.
21. M. Gorelick. *Python alla massima potenza: Programmazione pratica ad alte prestazioni.* HOEPLI EDITORE, 2015.
22. Henrik Herranen, Alar Kuusik, Tõnis Saar, Marko Reidla, Raul Land, Olev Märtens, and Jüri Majak. *Acceleration data acquisition and processing system for structural health monitoring.* 2014 IEEE Metrology for Aerospace (MetroAeroSpace), 2014.
23. Xiaoya Hu, Bingwen Wang, and Han Ji. *A wireless sensor network-based structural health monitoring system for highway bridges.* Computer-Aided Civil and Infrastructure Engineering), 2013.
24. U. Hunkeler, H.L. Truong, and A. Stanford-Clark. *MQTT-S–A publish/subscribe protocol for Wireless Sensor Networks.* IEEE, 2008.
25. David Jaramillo, Duy V Nguyen, and Robert Smart. *Leveraging microservices architecture by using Docker technology.* IEEE Communications magazine, 2016.
26. F. Reghenzani, G. Massari, and W. Fornaciari. *The real-time linux kernel: A survey on preempt_rt.* ACM Computing Surveys (CSUR), 2019.
27. J. C. Shovic. *Raspberry pi IoT projects.* Apress, 2016.
28. D. Vohra. *Kubernetes microservices with Docker.* Apress, 2016.
29. R.H. Weber and R. Weber. *Internet of things (Vol. 12).* Heidelberg: Springer, 2010.
30. Matthew J Whelan and Kerop D Janoyan. *Design of a robust, high-rate wireless sensor network for static and dynamic structural monitoring.* Journal of Intelligent Material Systems and Structures), 2009.
31. M.B. Yassein, M.Q. Shatnawi, S. Aljwarneh, and R. Al-Hatmi. *Internet of Things: Survey and open issues of MQTT protocol.* International Conference on Engineering & MIS (ICEMIS), 2017.

Ringraziamenti

Il primo ringraziamento va alla mia famiglia, che mi ha dato la possibilità, per nulla scontata, di intraprendere e portare a termine questo percorso universitario, sostenendomi nei momenti di difficoltà e condividendo le gioie dei momenti felici.

Vorrei ringraziare il prof. Ursino, per aver accettato di essere il mio relatore, per essersi speso nell'aiutarmi a trovare il percorso di tirocinio più adatto alle mie esigenze e per la straordinaria disponibilità dimostrata nel corso della stesura di questa tesi.

Ringrazio il mio collega Luca di Smart Space, e il responsabile Emiliano, per l'opportunità che mi hanno offerto e per il supporto che mi hanno fornito, guidandomi nelle attività svolte e dandomi preziosi aiuti nella scrittura di questa tesi.

Ringrazio di cuore la mia fidanzata Valentina, che è stata la persona che mi è stata più vicina in questo percorso, supportandomi in qualsiasi decisione presa e sostenendomi in tanti momenti di sconforto e di difficoltà.

Infine, vorrei ringraziare tutti i miei colleghi universitari e i miei amici di sempre, per essere stati sempre al mio fianco e per avermi permesso di trascorrere questi anni indimenticabili.