Università Politecnica delle Marche

Dipartimento di Ingegneria dell'Informazione

LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA E DELL'AUTOMAZIONE

Progettazione e sviluppo di un sistema per la valutazione in tempo reale della crescita di mitili in scenari di acquacoltura di precisione

Design and development of a IoT system to esimate the growth of mussels in precision aquaculture scenarios

Relatore:
Prof. Adriano Mancini

Candidato: 1101284
Nicola Silveri

Anno Accademico 2022 / 2023

Sommario

L'era dell'IoT (Internet of Things) ha segnato un periodo di significative innovazioni tecnologiche, influenzando diversi settori, tra cui quello dell'acquacoltura di precisione. In questo contesto, emerge la tesi dal titolo "Progettazione e sviluppo di un sistema IoT per la valutazione in tempo reale della crescita di mitili in scenari di acquacoltura di precisione", che mira a esplorare l'impiego di tecnologie avanzate per ottimizzare la crescita e la gestione dei mitili.

La tesi si pone l'obiettivo di descrivere tutte le fasi che rappresentano la progettazione, lo sviluppo e i test di un dispositivo IoT in un contesto di acquacoltura di precisione, evidenziando i punti di forza ed i vantaggi di questa soluzione. Nel primo capitolo verranno introdotti il progetto `Mussels Grow Tracker`, presentandone gli obiettivi ed i requisiti progettuali seguito da nozioni riguardanti il mondo dell'IoT; Nel secondo capitolo verranno descritte le tecnologie di supporto che hanno permesso a questo progetto di raggiungere gli obiettivi; Nel terzo capitolo entreremo nel vivo della progettazione, affrontando sia la parte `hardware` che la parte `software`, nel quarto capitolo analizzeremo i risultati ottenuti. Nell'ultimo capitolo discuteremo delle conclusioni e dei possibili sviluppi futuri.

La tesi si pone l'obiettivo sulla progettazione e sviluppo di un dispositivo IoT che non solo monitora e traccia la crescita dei mitili, ma assicura anche l'integrità e la validità dei dati raccolti attraverso l'impiego della tecnologia blockchain anche attraverso l'utilizzo di servizi cloud. Questa innovativa soluzione offre un approccio rivoluzionario alla gestione dell'acquacoltura, permettendo un monitoraggio accurato e in tempo reale, fondamentale per garantire l'efficienza e la sostenibilità del settore.

Di seguito vengono elencati alcuni dei principali risultati ottenuti:

- **Monitoraggio accurato**:
- **Tracciabilità migliorata**:
- **Sostenibilità**:
- **Sicurezza dei dati**:
- **Gestione semplificata**:
- **Adattabilità e scalabilità**:

Indice

Capitolo 1

Introduzione

L'acquacoltura, ovvero l'allevamento controllato di organismi acquatici come pesci e molluschi, sta vivendo una rivoluzione grazie all'integrazione delle tecnologie Internet delle Cose (IoT). Con l'IoT, sensori e dispositivi connessi permettono il monitoraggio in tempo reale di parametri critici come temperatura e qualità dell'acqua, migliorando l'efficienza e la sostenibilità dell'acquacoltura. Questa tecnologia facilita la gestione delle risorse acquatiche, automazione dei processi e assicura una migliore qualità e sicurezza dei prodotti.

In questo contesto, il progetto Mussels Grow Tracker, che chiameremo MGT, mira ad implementare una soluzione IoT sviluppata appositamente per i mitili, ponendosi come obiettivo il monitoraggio della crescita della coltura. Per realizzare il progetto `Mussels Grow Tracker`, è stato necessario definire e raggiungere una serie di obiettivi e soddisfare specifici requisiti di seguito elencati:

- **Obiettivi**:
  - **Monitoraggio Continuo**: Misurare costantemente il peso dei mitili e rilevare spostamenti anomali del dispositivo.
  - **Integrità e Validità dei Dati**: Garantire l'accuratezza e l'affidabilità delle informazioni raccolte.
  - **Sicurezza dei Dati**: Utilizzare tecniche avanzate per proteggere i dati raccolti da accessi non autorizzati o manomissioni.

  - **Trasparenza nel Processo di Raccolta Dati**: Fornire un metodo chiaro e verificabile per la raccolta e la gestione dei dati.

- **Requisiti**:
  - **Tecnologie di Firma dei Messaggi**: Implementazione di metodi avanzati per garantire la sicurezza dei dati trasmessi.
  - **Integrazione con Blockchain Ethereum**: Utilizzo della blockchain per aumentare la sicurezza e la trasparenza nella gestione dei dati.
  - **Edge Computing**: Applicazione dell'edge computing per migliorare la sicurezza dei dati, elaborando le informazioni vicino alla fonte e riducendo la vulnerabilità durante la trasmissione.
  - **Autonomia**: Assicurare una durata della batteria di circa 8 mesi, per ridurre la necessità di manutenzione frequente e garantire un monitoraggio costante.

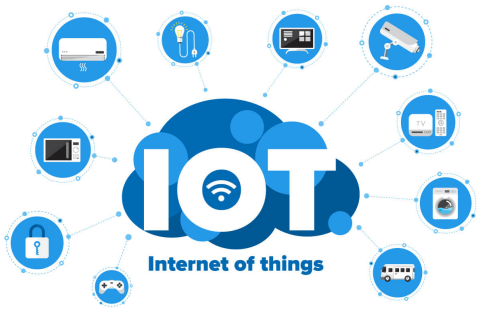

Figura 1.1: Internet of Things.(Fonte [1])

1.1 Internet of Things (IoT)

L'era dell'Internet delle Cose (IoT [1]) rappresenta una delle più significative rivoluzioni tecnologiche dell'era moderna, segnando un cambiamento epocale nel modo in cui interagiamo con il mondo che ci circonda. Questa era si caratterizza per l'integrazione di sensori, software e altre tecnologie in oggetti e dispositivi di uso quotidiano, permettendo loro di connettersi e scambiare dati con altri dispositivi e sistemi attraverso l'Internet.

L'avvento dell'IoT ha origine dall'evoluzione della connettività Internet e dall'abbattimento dei costi di produzione dei sensori e dei dispositivi elettronici. La diffusione capillare di reti wireless e l'aumento esponenziale della capacità di calcolo e di memorizzazione hanno reso possibile l'interconnessione di miliardi di dispositivi, dai più comuni come smartphone e televisori, fino a oggetti meno ovvi come frigoriferi, lampadine e persino impianti agricoli.

Il potenziale dell'IoT risiede nella sua capacità di fornire dati in tempo reale e *insights* approfonditi attraverso l'analisi di enormi quantità di dati raccolti dai dispositivi connessi. Questo flusso di informazioni offre possibilità senza precedenti in termini di automazione, efficienza, sicurezza e *decision-making*, trasformando radicalmente settori come l'industria manifatturiera, la sanità, l'agricoltura, la gestione delle città (*smart cities*) e l'ambiente domestico (*smart homes*).

Un aspetto cruciale dell'IoT è la sua capacità di facilitare la raccolta di dati in situazioni dove l'intervento umano sarebbe impraticabile o inefficiente. Ad

esempio, sensori IoT posizionati in ambienti agricoli possono monitorare condizioni come umidità, temperatura e livelli di nutrienti, fornendo dati preziosi per ottimizzare le tecniche di coltivazione. In ambito sanitario, dispositivi indossabili possono monitorare parametri vitali dei pazienti in tempo reale, migliorando la qualità delle cure e la prevenzione di malattie.

Con il crescente sviluppo dell'IoT, emergono sfide importanti, soprattutto in termini di sicurezza dei dati e privacy. La connettività sempre più estesa tra gli oggetti intelligenti solleva questioni cruciali riguardanti la protezione dei dati personali e la sicurezza delle reti da potenziali attacchi informatici. Con l'aumento di questi rischi, si intensifica anche la possibilità di attacchi mirati che possono minacciare non solo la sicurezza dei dati, ma anche l'integrità operativa dei sistemi fisici gestiti da dispositivi IoT. Problemi come l'intercettazione dei dati, l'accesso non autorizzato e il sabotaggio dei sistemi IoT possono portare a conseguenze serie, particolarmente in ambiti critici come la sanità, le infrastrutture essenziali e, in contesti specifici come quello dell'acquacoltura di precisione. Per contrastare le sfide legate alla sicurezza nell'ambito dell'IoT, è cruciale adottare misure di sicurezza efficaci e complete. Questo include l'implementazione di crittografia end-to-end, l'adozione di meccanismi di autenticazione forte e una gestione scrupolosa delle patch di sicurezza. Un ulteriore passo importante è aderire al principio di "*security by design*", che incoraggia l'integrazione della sicurezza nelle fasi iniziali dello sviluppo di un progetto IoT, garantendo che le precauzioni siano incorporate strutturalmente nel sistema sin dal suo concepimento. Questo approccio pro-attivo non solo rafforza la protezione dei dati e dei dispositivi, ma contribuisce anche a prevenire potenziali vulnerabilità nel ciclo di vita del prodotto.

1.1.1 Dispositivi Embedded

Un sistema embedded rappresenta una componente fondamentale dell'evoluzione tecnologica moderna, contribuendo in modo significativo all'Internet delle Cose (IoT). Questi sistemi integrano microprocessori all'interno di oggetti o sistemi informatici, consentendo il monitoraggio e il controllo delle loro funzionalità.

Sebbene possano passare inosservati, i sistemi embedded sono ubiqui, presenti in una vasta gamma di dispositivi, dai cellulari alle console, dai decoder alle lavatrici, e persino nei sistemi di volo. Questi sistemi sono alimentati da microcontrollori che elaborano informazioni, monitorano le condizioni e gestiscono le operazioni.

Il termine "embedded" in inglese significa "incorporato" o "incapsulato". Un sistema embedded è un tipo di elaboratore nascosto all'interno di un oggetto o di un sistema informatico, e svolge specifiche funzioni di monitoraggio e controllo. A differenza dei computer personali, questi sistemi sono progettati per eseguire compiti specializzati e spesso una singola operazione. Non sono generalmente ri-programmabili come i computer tradizionali e sono strettamente integrati con il sistema in cui operano. I sistemi embedded sono noti per le loro dimensioni ridotte, l'assenza di interazione umana e la capacità di ripristinare autonomamente il funzionamento in caso di guasto.

I sistemi embedded svolgono operazioni ripetute in modo tempestivo ed efficiente. La loro specificità li rende adatti per compiti di misurazione e controllo in tempo reale, con consumi energetici contenuti e bassi costi di manutenzione. Ad esempio, in caso di incendio, un sistema embedded può chiudere automaticamente le porte antincendio per prevenire la propagazione delle fiamme. Nei pacemaker, questi sistemi verificano che il dispositivo funzioni correttamente. Nelle lavatrici, monitorano i parametri operativi e segnalano eventuali anomalie.

Categorie di Sistemi Embedded Esistono diverse categorie di sistemi embedded, tra cui:

- **PLC (Programmable Logic Controller)**: Progettati per l'automazione industriale, sono programmati per eseguire funzioni specifiche, monitorare segnali di ingresso e attivare segnali di uscita.
- **Microcontrollori**: Integrano componenti del microprocessore in un piccolo pacchetto e sono ampiamente utilizzati per misurazioni e raccolta di dati ambientali.

- **single-board computer (SBC)**: rappresentano una categoria di computer embedded che includono tutti i componenti hardware essenziali su una singola scheda elettronica. Questi dispositivi offrono una soluzione completa e compatta per elaborare dati e svolgere varie funzioni informatiche senza la necessità di componenti aggiuntivi.

Sistemi Embedded e Internet delle Cose (IoT) I sistemi embedded integrati in oggetti o sistemi informatici più complessi sono diventati parte integrante dell'Internet delle Cose (IoT). Questi sistemi consentono ai dispositivi di comunicare tra loro e con le persone, acquisendo, elaborando e scambiando informazioni con l'ambiente per eseguire azioni in tempo reale. Tuttavia, non tutti i sistemi embedded sono considerati IoT, poiché alcuni sono progettati per uno scambio "chiuso" di dati tra dispositivi, senza interazione con l'ambiente. L'IoT opera su reti con protocollo IP e gestisce dati su gateway o cloud, rendendo la comunicazione più aperta.

Progettazione di un Sistema Embedded La progettazione di un sistema embedded richiede un'analisi approfondita del contesto in cui verrà utilizzato. Questi sistemi devono essere efficienti dal punto di vista energetico, con software a basso consumo di risorse e hardware compatto. Devono anche garantire affidabilità, sicurezza e, a seconda delle esigenze, operare in tempo reale. I sistemi embedded possono avere o meno un sistema operativo, con vantaggi e svantaggi associati.

1.1.2 Sistemi Embedded in commercio

Esistono diversi sistemi *embedded* disponibili in commercio, ciascuno con le proprie caratteristiche che lo distinguono.

Micro-controllori

- **STM32**: Gli STM32 sono una famiglia di micro-controllori prodotti da ST-Microelectronics. Questi micro-controllori sono noti per la loro flessibilità, prestazioni elevate e ampia gamma di varianti.

- **Caratteristiche**:
  - **Architettura CPU**: La maggior parte degli STM32 utilizza il core ARM Cortex-M, che offre prestazioni elevate e una vasta gamma di funzionalità. Questo li rende adatti a una varietà di applicazioni, dalle più semplici alle più complesse.
  - **Modelli**: Sono disponibili in numerosi modelli, ognuno con caratteristiche diverse, tra cui velocità di clock, quantità di memoria e periferiche integrate. Questa diversità permette agli sviluppatori di selezionare il microcontrollore più adatto alle esigenze del loro progetto.
  - **Ambienti di sviluppo**:
    - **STM32CubeIDE**: Tra i diversi ambienti di sviluppo, possimo citare il principale, STM32CubeIDE, integrato ufficiale fornito da STMicroelectronics per la programmazione degli STM32. Questo ambiente è basato su Eclipse e offre una vasta gamma di funzionalità, tra cui la configurazione delle periferiche, il debugging, il supporto per il linguaggio di programmazione C/C++, e integrazione con il framework STM32CubeMX per la generazione del codice di configurazione.
    - **Arduino IDE**: In alcuni schede basate su STM32 è possibile utilizzare anche l'ambiente di sviluppo Arduino IDE, il quale, oltre a rendere più semplice lo sviluppo, permette di accedere a tutte le risorse messe a disposizione della community.

- **Arduino**: è uno dei microcontrollori più conosciuti e utilizzati in tutto il mondo, particolarmente apprezzato per la sua accessibilità, versatilità e ampio supporto da parte della comunità degli appassionati.
  - **Caratteristiche**:
    - **Architettura CPU**: Arduino offre diverse tipologie di modelli con diverse architetture, dagli AVR agli ARM-Cortex

* **Modelli**: Sono disponibili in numerosi modelli, ognuno con caratteristiche diverse, tra cui velocità di clock, quantità di memoria e periferiche integrate. Questa diversità permette agli sviluppatori di selezionare il microcontrollore più adatto alle esigenze del loro progetto.
* **Open Source**: Essendo una piattaforma open source, Arduino ha una vasta comunità di sviluppatori e appassionati che condividono progetti, librerie e risorse. Questo rende più facile l'apprendimento e la risoluzione dei problemi.
    - **Arduino IDE**: Arduino IDE (Integrated Development Environment): Arduino IDE è l'ambiente di sviluppo ufficiale di Arduino ed è ampiamente utilizzato nella comunità Arduino. Si tratta di un'applicazione open-source basata su Java, progettata per semplificare il processo di sviluppo e programmazione delle schede Arduino. Come linguaggio di programmazione viene utilizzato Wiring, derivato da C/C++, ed è noto per la sua facilità d'uso fornendo un'interfaccia intuitiva che consente agli sviluppatori di scrivere il proprio codice e caricarlo facilmente sulle schede Arduino, il che lo rende adatto sia ai principianti che agli sviluppatori esperti. Inizialmente, l'IDE era progettato per le schede Arduino ufficiali, ma nel tempo, anche grazie ai vari porting della cummunity, ha esteso il supporto a una vasta gamma di schede Arduino compatibili e micro-controllori di terze parti. Ciò significa che è possibile utilizzare Arduino IDE con una varietà di hardware. l'IDE offre una vasta raccolta di librerie predefinite. Queste librerie semplificano la programmazione di funzioni comuni e permettono agli sviluppatori di accelerare lo sviluppo dei propri progetti. Una delle principali forze di Arduino è la sua grande community di sviluppatori e utenti. Questa community condivide progetti, librerie personalizzate e risorse online,

rendendo più facile l'apprendimento e la risoluzione dei problemi. Arduino IDE è una scelta popolare per coloro che desiderano iniziare a sviluppare progetti con schede Arduino grazie alla sua facilità d'uso, al supporto esteso e alla vasta documentazione disponibile.

single-board computer (SBC)

I single-board computer (SBC) sono dispositivi completi basati su un'unica scheda elettronica che integra CPU, memoria, periferiche e connettività. Sono progettati per scopi generali e possono essere utilizzati per una vasta gamma di applicazioni, dalla prototipazione all'automazione domestica, all'informatica embedded. Le SBC sono spesso compatte e a basso costo, rendendole ideali per progetti di sviluppo e applicazioni embedded. Essendo dei veri e propri computer in miniatura, hanno a bordo un sistema operativo (tra i quali spiccano le distribuzioni Linux, Windows IoT ed altri sistemi) e presentano una hardware di bordo molto più potente rispetto ad un micro-controllore.

- **Raspberry Pi**: Tra le SBC più famose non possiamo non nominare la Raspberry Pi. La Raspberry Pi è una linea di single-board computer sviluppata dalla Raspberry Pi Foundation. Questi dispositivi sono noti per la loro versatilità, prezzo accessibile e ampio supporto da parte della comunità di sviluppatori. La Raspberry Pi è disponibile in diverse versioni, ciascuna con specifiche diverse, ma in genere includono una CPU, memoria RAM, porte USB, connettività Ethernet e HDMI, oltre a pin GPIO (General-Purpose Input/Output) che consentono di collegare sensori e dispositivi esterni. La Raspberry Pi è ampiamente utilizzata per progetti di programmazione, domotica, robotica, ed è una piattaforma popolare per l'apprendimento dell'informatica e dell'elettronica. Grazie al suo ecosistema di software e risorse online, la Raspberry Pi è diventata un punto di riferimento nel mondo degli SBC.

- **Sistema Operativo**: Ufficialmente, dalla Raspberry Pi Foundation viene fornito il sistema operativo RaspberryOS, una distribuzione basata su Debian (Linux), installabile attraverso un tool ufficiale su una MicroSD; Sono disponibili anche sistemi operativi di terze parti che supportano la Board.
- **Ambiente di sviluppo**: Grazie al sistema operativo integrato, la Raspberry Pi è in grado di supportare diversi linguaggi di programmazione, tra cui Python, C, C++, e NodeJS. Gli sviluppatori possono creare programmi o script in questi linguaggi per eseguire una varietà di operazioni desiderate sulla Raspberry Pi. Questi programmi possono interagire con i sensori e gli attuatori esterni collegati attraverso la porta GPIO, consentendo un controllo completo e la possibilità di realizzare progetti personalizzati e applicazioni specifiche.

Differenze tra micro-controllore ed single-board computer

I micro-controllori sono dispositivi creati per eseguire compiti specifici in applicazioni embedded. La loro progettazione è ottimizzata per il controllo di dispositivi, sensori o sistemi dedicati, ed è caratterizzata da:

- **Specificità**: i micro-controllori sono progettati per eseguire un compito molto specifico, come controllare la temperatura in un termostato o gestire un motore in un dispositivo industriale. La loro architettura è mirata a questa specifica funzione.
- **Stabilità**: i micro-controllori di solito hanno risorse limitate, comprese dimensioni ridotte di memoria e capacità di calcolo limitate. Questo li rende adatti per applicazioni con requisiti modesti di elaborazione.
- **Consumo Energetico Ridotto**: spesso sono progettati per funzionare con un basso consumo energetico, il che li rende adatti per dispositivi alimentati a batteria o con alimentazione limitata, permettendo di rimanere attivi anche per lunghissimi periodi.

- **Risorse Limitate**: i micro-controllori di solito hanno risorse limitate, comprese dimensioni ridotte di memoria e capacità di calcolo limitate. Questo li rende adatti per applicazioni con requisiti modesti di elaborazione.

D'altra parte, i single-board computer (SBC) sono dispositivi più versatili con l'abilità di eseguire una vasta gamma di applicazioni. Le loro principali differenze includono:

- **Versatilità**: gli SBC sono estremamente versatili e possono essere utilizzati per eseguire una varietà di applicazioni, dal web server all'elaborazione multimediale e allo sviluppo software. Possono essere adattati per una vasta gamma di utilizzi.

- **Architettura Più Potente**: Gli SBC sono dotati di architetture più potenti, CPU più veloci e più memoria rispetto ai microcontrollori. Questo li rende adatti per eseguire software complessi e multitasking.

- **Sistema Operativo Completo**: Gli SBC di solito supportano sistemi operativi completi come Linux, che consente di eseguire una vasta gamma di applicazioni e software di terze parti.

- **Applicazioni Varie**: Gli SBC sono utilizzati in progetti di hobby, sviluppo software, educazione, server leggeri e molto altro. La loro flessibilità li rende adatti per molte applicazioni diverse.

In sintesi, mentre i micro-controllori sono progettati per eseguire compiti specifici e dedicati, i single-board computer sono dispositivi più versatili con un'architettura più potente e sono in grado di eseguire una vasta gamma di applicazioni, rendendoli ideali per progetti più generici e adattabili.

Caratteristiche comuni

- **GPIO (General-Purpose Input/Output)**: Sia i micro-controllori che gli SBC sono spesso dotati di pin GPIO che consentono di interagire con il mondo esterno. Questi pin possono essere configurati come input o output

per connettersi a sensori, attuatori o altri dispositivi esterni. La capacità di utilizzare la GPIO è essenziale per controllare e monitorare dispositivi esterni.

- **Comunicazione Seriale**: Entrambi i dispositivi supportano la comunicazione seriale, come UART, SPI e I2C, che consente loro di comunicare con altri dispositivi digitali. Questa funzionalità è utile per connettersi a sensori, display e altri componenti.

Nonostante le differenze e grazie a caratteristiche comuni, gli SBC e i micro-controllori possono essere utilizzati in simbiosi in molti progetti, sfruttando la potenza di elaborazione degli SBC e la precisione dei micro-controllori per creare soluzioni complesse e efficienti.

1.1.3 LoRaWAN

Figura 1.2: Logo LoRa Alliance [1]

Nel contesto del nostro progetto, abbiamo adottato la tecnologia LoRaWAN, una scelta che si è rivelata strategica per le sue eccezionali caratteristiche di connettività ed efficienza energetica.

- **Architettura di rete**: L'architettura di rete LoRaWAN si basa su una topologia a stella di stelle, in cui i gateway fungono da intermediari nelle comunicazioni tra i dispositivi finali e un server di rete centrale. Questi gateway sono connessi al server di rete tramite connessioni IP standard e operano come ponti trasparenti, convertendo semplicemente i pacchetti RF (Radio Frequency) in pacchetti IP e viceversa.

  La comunicazione wireless sfrutta le caratteristiche di lungo raggio del livello fisico LoRa, permettendo un collegamento diretto e unico tra il dispositivo finale e uno o più gateway. Questo aspetto è fondamentale per garantire una copertura capillare e affidabile, consentendo ai dispositivi di comunicare efficacemente anche da posizioni remote o in condizioni ambientali complesse.

  Una delle forze di LoRaWAN è la sua capacità di comunicazione bidirezionale, che non solo permette di inviare dati dai dispositivi al server di rete, ma anche di ricevere comandi o aggiornamenti in direzione opposta. Questo è particolarmente utile per operazioni come gli aggiornamenti Firmware Over-The-Air (FOTA), che richiedono l'invio di pacchetti dati a un gruppo di dispositivi, ottimizzando l'uso dello spettro e riducendo il consumo energetico.

Inoltre, LoRaWAN supporta la creazione di gruppi di indirizzamento multicast, una funzionalità che migliora ulteriormente l'efficienza della rete durante la distribuzione di messaggi a un vasto numero di dispositivi. Questo approccio non solo semplifica la gestione di larga scala degli aggiornamenti software, ma contribuisce anche a mantenere bassi i costi operativi e a massimizzare la durata della batteria dei dispositivi.

Grazie a queste caratteristiche, la scelta di LoRaWAN per il nostro progetto ci ha permesso di costruire una soluzione IoT robusta, scalabile e energeticamente efficiente. La rete LoRaWAN, con la sua architettura flessibile e le sue capacità di comunicazione avanzate, si è dimostrata una piattaforma ideale per affrontare le sfide dell'IoT e per realizzare applicazioni innovative che possono trasformare il modo in cui interagiamo con il mondo che ci circonda.

- **Classi**: Per soddisfare le diverse esigenze delle applicazioni IoT, LoRaWAN prevede tre diverse classi di dispositivi finali:

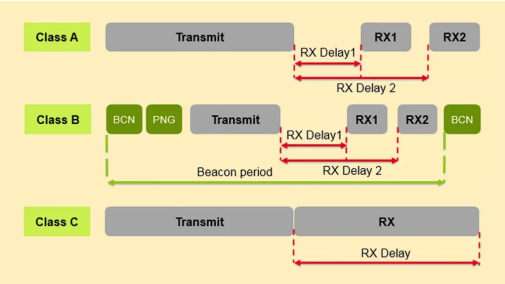

Figura 1.3: Classi LoRa [2]

  - **Classe A**: Questa classe è caratterizzata dal consumo energetico più basso e prevede dispositivi finali bi-direzionali. La comunicazione è sempre iniziata dal dispositivo finale, che può trasmettere in qualsiasi momento e seguire la trasmissione con due brevi finestre per la ricezione dei dati. Questo protocollo di tipo ALOHA permette ai dispositivi di entrare in modalità sleep a basso consumo energetico per lunghi periodi, definibili in base alle esigenze dell'applicazione, rendendo la Classe A la modalità operativa a più basso consumo energetico.

– **Classe B**: I dispositivi di Classe B aggiungono finestre di ricezione pianificate oltre a quelle iniziate dalla Classe A, permettendo la sincronizzazione con la rete attraverso segnali periodici e l'apertura di finestre di ricezione "ping slot" a tempi programmati. Questo riduce la latenza delle comunicazioni in downlink a scapito di un leggero aumento del consumo energetico. La latenza è programmabile fino a 128 secondi, e il consumo energetico aggiuntivo è comunque compatibile con applicazioni alimentate a batteria.

– **Classe C**: Questa classe riduce ulteriormente la latenza mantenendo il ricevitore del dispositivo finale sempre attivo, eccetto quando trasmette. Questo permette al server di rete di iniziare trasmissioni in downlink in qualsiasi momento, eliminando la latenza ma aumentando il consumo energetico del ricevitore. La Classe C è ideale per applicazioni alimentate continuamente.

Per i dispositivi alimentati a batteria, è possibile passare temporaneamente tra le classi A e C, utilizzando questa flessibilità per compiti intermittenti come gli aggiornamenti firmware.

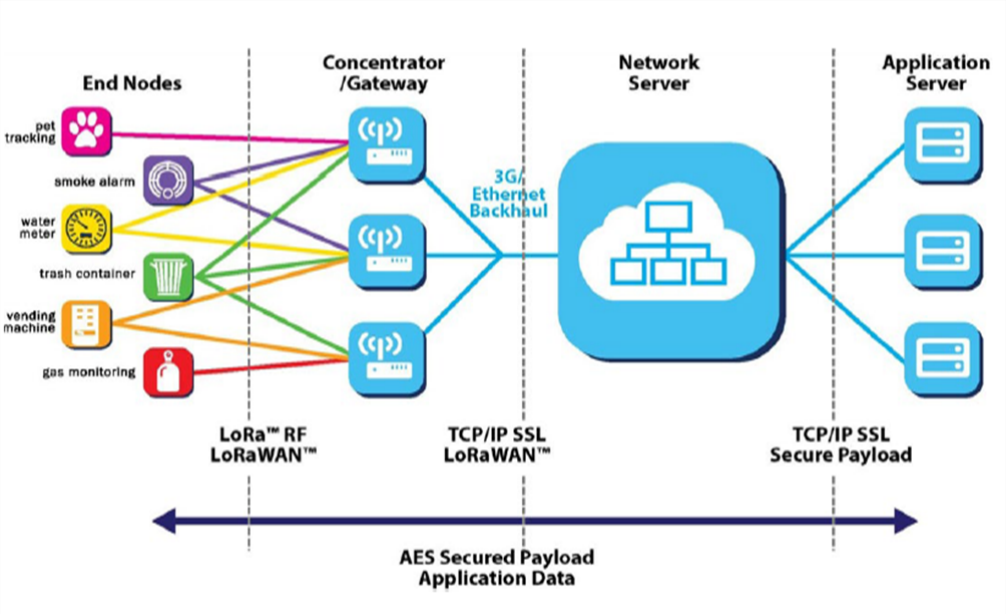

Figura 1.4: Esmpio degli utilizzi di LoRa [1]

I vantaggi di LoRaWAN includono:

- **Ampio raggio di copertura**: LoRaWAN può coprire distanze fino a 20 km in campo aperto, riducendo il numero di gateway necessari e abbattendo i costi operativi.
- **Deep Indoor Penetration**: la sua elevata immunità al rumore consente di raggiungere dispositivi situati in luoghi difficilmente accessibili, come contatori dell'acqua sotterranei o sensori in cantine.
- **Flessibilità della rete**: con gateway disponibili sia per interni che per esterni, LoRaWAN offre una versatilità senza pari nella realizzazione di reti IoT personalizzate.
- **Sicurezza**: la trasmissione dati criptata con doppia chiave assicura la protezione delle informazioni trasmesse.
- **Semplicità di implementazione**: il meccanismo delle chiavi di criptazione e la natura non selettiva dei gateway facilitano l'installazione e l'integrazione dei dispositivi.
- **Lunga durata delle batteri**: grazie alla trasmissione a basso consumo, i dispositivi possono operare per anni senza necessità di sostituire le batterie.
- **Ecosistema ricco**: un'ampia gamma di dispositivi e soluzioni disponibili supporta lo sviluppo di applicazioni IoT innovative.
- **Nessun Lock-in**: lo standard aperto permette flessibilità e libertà di scelta tra diversi fornitori e operatori.

In conclusione, la scelta di LoRaWAN per il nostro progetto si basa su queste solide fondamenta, offrendoci la possibilità di sfruttare una tecnologia all'avanguardia per superare le sfide dell'IoT e realizzare soluzioni innovative ed efficienti.

Capitolo 2

Tecnologie di supporto

2.1 La Blockchain

La tecnologia *Blockchain* è un avanzato sistema di database che facilita la condivisione trasparente di informazioni all'interno di una rete. Questo database archivia i dati in blocchi che sono cronologicamente collegati in una catena, rendendo impossibile la modifica o l'eliminazione dei dati senza il consenso della rete. Questa caratteristica rende la blockchain ideale per creare registri inalterabili, utili per tracciare transazioni di vario genere. Le tecnologie di database tradizionali presentano sfide nell'affidabilità e nella sicurezza delle transazioni. La blockchain supera questi ostacoli, offrendo un sistema decentralizzato e a prova di manomissione. Per esempio, in una transazione immobiliare, la blockchain elimina la necessità di una terza parte fidata, riducendo la complessità e aumentando la sicurezza. La blockchain trova applicazioni innovative in vari settori. Nell'energia, è usata per il commercio energetico peer-to-peer e l'accesso all'energia rinnovabile. Nel settore finanziario, migliora la gestione dei pagamenti e delle transazioni di mercato. Nei media, gestisce i diritti d'autore, mentre nel retail traccia l'autenticità delle merci.

2.1.1 Caratteristiche Principali della Blockchain:

- **Decentralizzazione**: la decentralizzazione è uno dei pilastri della blockchain. In una rete blockchain, il controllo e il processo decisionale sono distribuiti tra tutti i partecipanti anziché concentrati in un'entità centralizzata. Questo aspetto elimina la necessità di una figura di autorità centrale e riduce la dipendenza dalla fiducia in terze parti. Ogni partecipante della rete ha una copia del registro completo e può verificare le transazioni. Questo sistema rende difficile qualsiasi tentativo di manipolazione o frode, in quanto le modifiche devono essere approvate dalla maggioranza dei partecipanti. La decentralizzazione è fondamentale per la sicurezza e l'affidabilità della blockchain.

- **Immutabilità**: l'immutabilità è un'altra caratteristica cruciale della blockchain. Una volta che una transazione è registrata su un blocco e aggiunta alla catena, non può essere modificata o eliminata. Questa immutabilità deriva dall'uso di algoritmi di crittografia avanzati e da una struttura dati che collega i blocchi in modo permanente. Qualsiasi tentativo di alterare una transazione esistente comporterebbe la modifica di tutti i blocchi successivi, il che è praticamente impossibile a causa della potenza di calcolo richiesta. Questa caratteristica assicura che i dati sulla blockchain siano affidabili e sicuri, eliminando il rischio di frodi o manipolazioni.

- **Consenso**: il consenso è il processo mediante il quale le transazioni vengono validate e aggiunte alla blockchain. Nei sistemi blockchain, le regole di consenso sono stabilite all'inizio e devono essere seguite da tutti i partecipanti alla rete. Una transazione può essere registrata solo quando la maggioranza dei partecipanti alla rete è d'accordo sulla sua validità. Questo processo di consenso elimina la necessità di un'autorità centrale per verificare le transazioni, rendendo la blockchain indipendente e autonoma. Il consenso garantisce che solo le transazioni legittime vengano aggiunte alla blockchain, contribuendo alla sua integrità.

2.1.2 Componenti Chiave della Tecnologia Blockchain:

- **Libro Mastro Distribuito**: il registro distribuito è il cuore della blockchain. Si tratta di un database condiviso e pubblico che archivia tutte le transazioni registrate nella rete. Ogni partecipante alla blockchain ha una copia del registo completo. Questo registro è diverso dai tradizionali database centralizzati in quanto non può essere modificato senza il consenso della rete. Ogni nuova transazione viene registrata come un blocco di dati collegato cronologicamente ai blocchi precedenti, creando una catena di blocchi immutabile. Questa struttura garantisce la trasparenza e la sicurezza dei dati sulla blockchain.

- **Smart Contract**: i contratti smart sono programmi informatici autonomi memorizzati sulla blockchain. Essi definiscono le regole e le condizioni di un contratto aziendale e vengono eseguiti automaticamente quando queste condizioni vengono soddisfatte. I contratti smart consentono l'automazione di transazioni e processi aziendali senza la necessità di un intermediario. Ad esempio, un contratto smart può automatizzare il pagamento una volta che una certa quantità di merci è stata consegnata, garantendo che le transazioni siano eseguite in modo sicuro ed efficiente.

- **Crittografia asimmetrica**: la crittografia asimmetrica rappresenta un componente essenziale per garantire la sicurezza e l'identificazione dei partecipanti alla rete blockchain. Ogni partecipante dispone di due chiavi: una chiave pubblica e una chiave privata. La chiave pubblica è condivisa e nota a tutti, mentre la chiave privata è segreta e conosciuta solo dal suo proprietario. Quando viene effettuata una transazione, la chiave privata viene utilizzata per crittografare la transazione, che può quindi essere decriptata solo con la chiave pubblica corrispondente. Questo meccanismo garantisce che solo i legittimi proprietari delle chiavi possano effettuare transazioni sulla blockchain e che le transazioni siano sicure e autentiche.

2.2 ECDSA

L'ECDSA (Elliptic Curve Digital Signature Algorithm) è un meccanismo crittografico utilizzato per garantire l'integrità e l'autenticità delle comunicazioni digitali. Questo algoritmo di firma digitale è basato sulla crittografia a curve ellittiche, una forma avanzata di crittografia che permette di ottenere un elevato livello di sicurezza con chiavi di dimensioni minori rispetto ad altri metodi crittografici come RSA. Il suo utilizzo si è diffuso in vari ambiti dell'informatica, inclusi i protocolli di sicurezza per Internet e le tecnologie blockchain, grazie alla sua efficienza e sicurezza.

2.2.1 Funzionamento di ECDSA

Il cuore dell'ECDSA risiede nell'uso di un'equazione matematica che definisce una curva ellittica. Su questa curva, si seleziona un punto che funge da origine e, attraverso operazioni matematiche specifiche, si generano coppie di chiavi pubbliche e private correlate tra loro:

2.2.1.1 Chiave Privata:

Un numero casuale che agisce come un segreto conosciuto solo dal proprietario. Questa chiave è il punto di partenza per generare la chiave pubblica e firmare i dati, garantendo che solo il proprietario possa produrre firme valide.

2.2.1.2 Chiave Pubblica:

Generata a partire dalla chiave privata tramite calcoli matematici sulla curva ellittica. La chiave pubblica può essere liberamente condivisa e utilizzata da terzi per verificare l'autenticità delle firme digitali create con la corrispondente chiave privata, senza compromettere la sicurezza.

2.2.2 Processo di Firma e Verifica

2.2.2.1 Firma:

Quando un utente desidera firmare digitalmente un documento o un messaggio, utilizza la propria chiave privata e l'hash del documento per generare una firma unica. Questa firma, composta dai valori "r" e "s", viene poi allegata al documento.

2.2.2.2 Verifica:

Chi riceve il documento può utilizzare la chiave pubblica del mittente e la firma per verificare che il documento non sia stato alterato e che la firma sia stata effettivamente generata dalla chiave privata corrispondente. Questo processo conferma l'integrità del documento e l'identità del firmatario.

2.2.3 Vantaggi dell'ECDSA:

2.2.3.1 Sicurezza Elevata:

Grazie alla complessità della crittografia a curve ellittiche, ECDSA offre un alto livello di sicurezza, rendendo praticamente impossibile falsificare le firme digitali con la tecnologia computazionale attuale.

2.2.3.2 Efficienza:

Le chiavi più corte necessarie per un livello di sicurezza comparabile a quello di altri algoritmi riducono il sovraccarico computazionale e di storage, rendendo ECDSA particolarmente adatto per ambienti con risorse limitate.

2.2.3.3 Applicabilità:

L'ECDSA è utilizzato in una vasta gamma di tecnologie, inclusi i protocolli di sicurezza web come SSL/TLS e le reti blockchain, dove la sicurezza delle transazioni e la legittimità degli scambi sono critiche.

2.3 SHA-256

Lo SHA-256, acronimo di Secure Hash Algorithm 256-bit, è una funzione di hash crittografica che rappresenta una componente essenziale nella sicurezza e integrità delle informazioni in numerosi sistemi informatici, inclusi quelli basati su tecnologia blockchain. Sviluppato dall'Agenzia per la Sicurezza Nazionale degli Stati Uniti (NSA) e dal National Institute of Standards and Technology (NIST), SHA-256 fa parte della famiglia SHA-2 di algoritmi di hash e si distingue per la sua robustezza e affidabilità.

2.3.1 Funzionamento di SHA-256:

Il principio alla base di SHA-256 è la generazione di un "impronta digitale" univoca o hash da un input di dati qualsiasi, indipendentemente dalla sua dimensione. Questa impronta digitale è una stringa di 64 caratteri (32 byte) che riassume il contenuto originale in una forma compatta. Il processo di hash è unidirezionale, il che significa che, sebbene sia semplice generare un hash a partire dai dati originali, è praticamente impossibile ricostruire i dati di partenza dall'hash senza conoscere l'input specifico.

2.3.2 Caratteristiche Principali:

2.3.2.1 Resistenza alle Collisioni:

SHA-256 è progettato per essere resistente alle collisioni, ovvero la difficoltà di trovare due input diversi che producono lo stesso output di hash. Questa caratteristica è fondamentale per mantenere l'unicità e l'integrità dei dati.

2.3.2.2 Determinismo:

A parità di input, SHA-256 genera sempre lo stesso hash, garantendo la coerenza nella verifica dei dati. Efficienza: Nonostante l'elevata sicurezza, SHA-256 rimane efficiente in termini di costi computazionali, bilanciando sicurezza e performance.

Lunghezza Fissa dell'Hash: Indipendentemente dalla lunghezza dell'input, l'hash generato da SHA-256 ha sempre una lunghezza fissa di 256 bit (32 byte).

2.3.2.3 Applicazioni di SHA-256:

SHA-256 trova impiego in una varietà di contesti, dalla validazione dell'integrità dei dati alla sicurezza delle transazioni in reti blockchain. Nel campo delle criptovalute, come Bitcoin, SHA-256 è utilizzato non solo nel processo di mining, contribuendo alla creazione di nuovi blocchi e alla sicurezza della rete, ma anche nella generazione di indirizzi Bitcoin, assicurando l'anonimato e la sicurezza delle transazioni.

All'interno delle blockchain, SHA-256 aiuta a mantenere una registrazione immutabile e sicura di tutte le transazioni. Qualsiasi tentativo di alterare retroattivamente i dati di un blocco altererebbe l'hash associato, rendendo evidente la manipolazione grazie alla verifica distribuita su tutti i nodi della rete.

In conclusione, SHA-256 rappresenta uno standard crittografico di riferimento per la protezione dei dati digitali, offrendo un equilibrio ottimale tra sicurezza e efficienza, essenziale per l'infrastruttura delle moderne tecnologie informatiche e delle reti blockchain.

2.4 Tecnologie wireless:

Nell'era digitale, la diversità delle tecnologie di comunicazione gioca un ruolo cruciale, offrendo una vasta gamma di opzioni per connettere dispositivi e sistemi. L'esistenza di queste molteplici tecnologie permette una maggiore flessibilità e personalizzazione in vari ambiti. Tuttavia, nell'ambito dell'IoT, la scelta della tecnologia di connessione adeguata diventa particolarmente importante. La decisione deve essere basata su fattori come la portata del segnale, il consumo energetico, la capacità di trasferimento dei dati e la sicurezza. Una selezione accurata può avere un impatto significativo sul successo e l'efficienza di un progetto IoT, ottimizzando l'interazione tra i dispositivi e migliorando l'efficacia complessiva del sistema.

Tra le numerose opzioni per l'IoT possiamo citare: Wi-Fi, bluetooth, Zigbee, Z-Wave, Anocean, 2G, 3G, 4G, 5G, NB-IoT, Sigfox, LoRaWAN, LTE-M, Wireless M-Bus e tante altre.

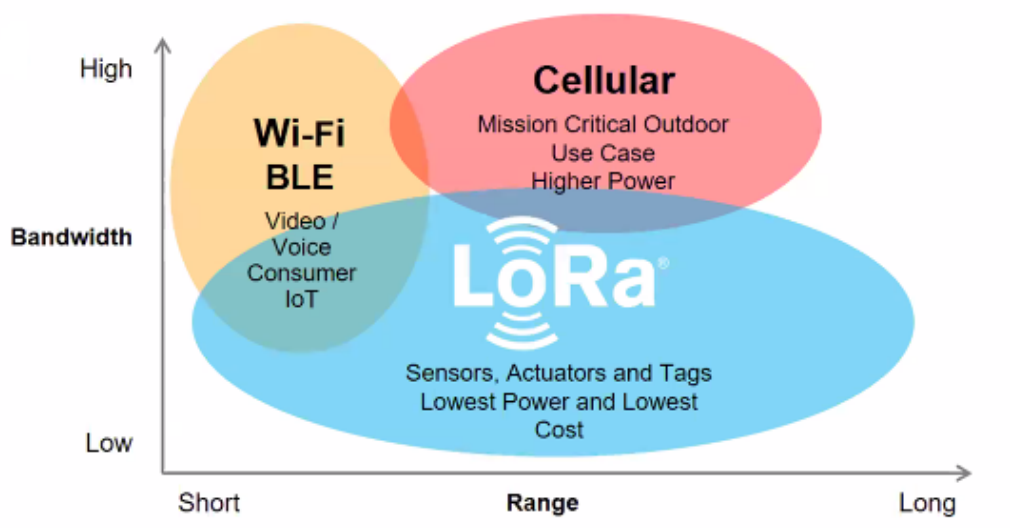

Figura 2.1: Confronto tra Bandwidth e Range tra alcune tecnologie wireless

Nella figura si mettono a confronto diverse tecnologie di connettività, focalizzandosi su due parametri cruciali per la scelta della soluzione più adeguata: la larghezza di banda (Bandwidth) e la portata del segnale (Range). Questo confronto aiuta a valutare quale tecnologia sia più appropriata a seconda delle specifiche esigenze di larghezza di banda e distanza di copertura richieste dall'applicazione IoT.

2.4.1 GNSS

Il sistema GNSS (Global Navigation Satellite System) rappresenta una tecnologia fondamentale per il posizionamento e la navigazione globale, offrendo dati dettagliati sulla posizione geografica, velocità e direzione dei dispositivi in tempo reale. Questo termine collettivo abbraccia una varietà di sistemi satellitari che orbitano intorno alla Terra, trasmettendo segnali che permettono di determinare informazioni geospaziali precise. Il GNSS è impiegato in svariati ambiti, inclusi quelli della geologia, topografia, ingegneria, agricoltura e trasporti, grazie alla sua capacità di fornire dati di posizionamento estremamente accurati su scala globale.

Oltre al noto GPS (Global Positioning System), creato dal Dipartimento della Difesa degli Stati Uniti, il GNSS comprende altri sistemi di navigazione come il GLONASS della Russia, il Galileo dell'Unione Europea e il BeiDou della Cina. Questi sistemi contribuiscono collettivamente a migliorare la precisione e l'affidabilità delle informazioni di posizionamento disponibili.

Capitolo 3

Sviluppo del progetto

In questo capitolo, verrà fornita una dettagliata e approfondita disamina circa le attività di progettazione e sviluppo del progetto di tesi. Si esamineranno le varie fasi e le tecniche impiegate, mettendo in luce come ciascuna di esse contribuisca a rispettare i requisiti e raggiungere gli obiettivi del progetto.

Il progetto di tesi si concentra sullo sviluppo del Mussels Grow Tracker, un innovativo dispositivo IoT finalizzato al monitoraggio della crescita dei mitili.

3.1 Mussels Grow Tracker

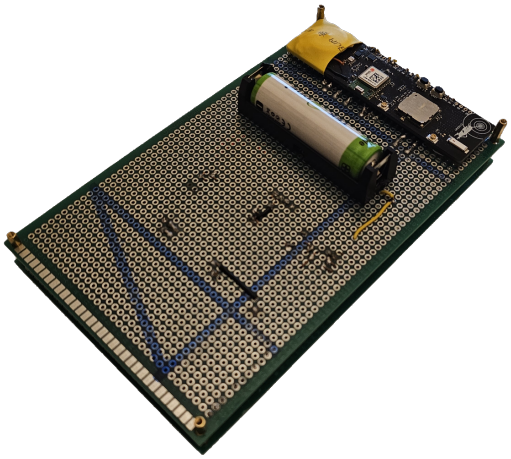

Figura 3.1: MGT visto dall'alto

Nello sviluppo del progetto, possiamo evidenziare due aree principali: l'Hardware e il Software. Questa distinzione consente di focalizzarsi separatamente sui componenti fisici e sulle applicazioni informatiche, ognuna con le proprie specificità e sfide, che insieme formano l'ecosistema integrato del "Mussels Grow Tracker" che chiameremo d'ora in avanti "MGT".

3.1.1 Hardware

Nella componente Hardware del progetto, è stato realizzato un prototipo su base millefori. Questo approccio ha permesso di assemblare e integrare tutte le componenti elettroniche necessarie per lo sviluppo e la messa in funzione del "MGT", fornendo una piattaforma solida per i test e l'ulteriore sviluppo del dispositivo.

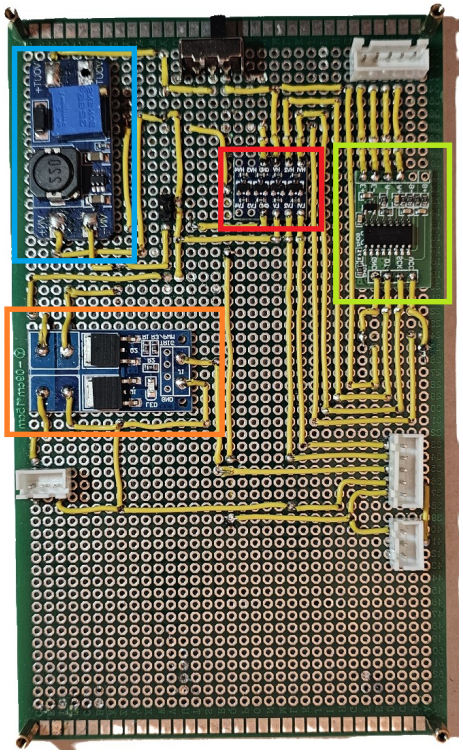

Figura 3.2: Vista interna dettagliata `top-view` del MGT

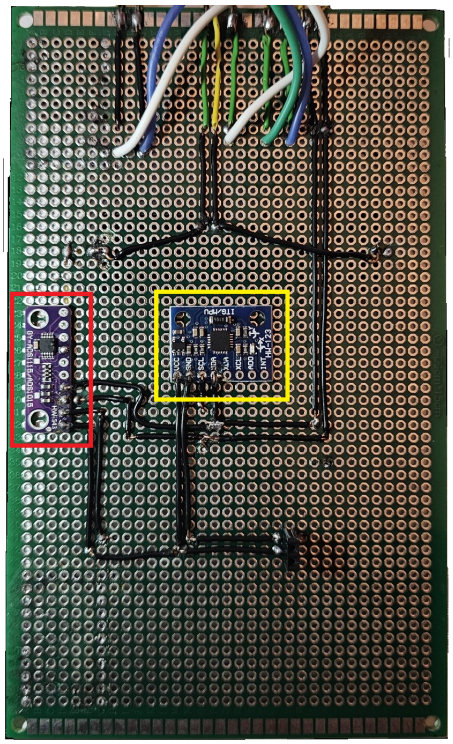

Figura 3.3: Visione interna dettagliata `bottom-view` del MGT

Per una maggiore precisione nella descrizione del sistema, possiamo effettivamente distinguere tra l'hardware "interno" o integrato nel micro-controllore,

cuore del sistema, e l'hardware "esterno" aggiuntivo, che viene montato sulla millefori. Questa distinzione aiuta a delineare con chiarezza le funzioni integrate nel micro-controllore rispetto a quelle che vengono estese o potenziate tramite componenti hardware supplementari, offrendo una comprensione più dettagliata e strutturata dell'intero sistema.

In figura 3.2 e 3.3 possiamo anticipare i moduli che andremo ad analizzare. In particolare, nella Figura 3.2 troviamo il modulo **HX711** (rettangolo **verde**), indicato in 3.11, il **Level Shifter** (rettangolo **rosso**), indicato in 3.9, e lo **switch MOSFET** (rettangolo **arancione**), indicato in 3.12. Mentre nella Figura 3.3 troviamo l'**ADS1115** (rettangolo **rosso**), indicato in 3.14, e il **MPU6050** (rettangolo **giallo**), indicato in 3.13.

3.1.1.1 Cricket Asset Tracker

Figura 3.4: Long Cricket Asset Tracker

Come appena accennato, il cuore del sistema hardware del "MGT" è il Cricket Asset Tracker (che chiameremo CAT), un micro-controllore basato su processore con architettura STM32. Questo dispositivo, prodotto dalla Tlera Corp[3], si distingue per la presenza di moduli e sensori già integrati, che aggiungono funzionalità essenziali al di là delle caratteristiche standard dell'architettura STM32, rendendolo particolarmente adatto per il progetto in questione. Questa integrazione offre un vantaggio significativo in termini di efficienza e praticità per il monitoraggio dei mitili.

Entrando nelle specifiche tecniche computazionali troviamo:

- **Processore**: Processore Cortex M0+ 32 bit con clock che può variare tra 4, 16, or 32 MHz
- **RAM**: 20kB di SRAM a disposizione per l'esecuzione del firmware
- **ROM**: 192kB di ROM su cui poter caricare il firmware

3.1.1.2 Hardware integrato

A bordo possiamo trovare sia i sensori e moduli integrati, sia la GPIO. La GPIO ci consente di interfacciare la Cricket, o i micro-controllori in generale, con sensori ed attuatori esterni.

Figura 3.5: GPIO (General Purpose Input/Output)

Come mostrato in figura 3.5 il Cricket mette a disposizione 18 Pin che ora andremo a descrivere brevemente:

- **SDA, SCL**: Consentono di estendere le capacità della board con sensori ed attuatori che comunicano con protocollo I2C.
- **Output 3V3**: Consente di alimentare sensori ed attuatori esterni con una tensione di 3V3.
- **RST**: Consente di riavviare la board quando collegato a GND.
- **A2, A3**: Consentono di leggere input analogici oltre a supportare funzionalità aggiuntive:
  - **UART**: RX2 (A3)/TX2 (A2) utilizzabili quindi per la comunicazione UART.
  - **ADC e PWM**: ADC disponibile sui pin A2 e A3, che possono essere utilizzati anche per PWM.
- **D8, D9**: Consentono di leggere input digitali e supportano funzionalità aggiuntive:
  - **UART**: RX3 (D8)/TX3 (D9) per la comunicazione UART.
- **+, -**: Consentono di collegare il polo positivo (+) e polo negativo (-) di una batteria al litio (ricaricabile e non), la scheda ha soltanto il supporto per essere alimentata da batteria (*power booster*), ma non supporta la funzione di ricarica (*power charger*).

- **Pin interni**: Sono presenti anche alcuni Pin interni, non visibili ad occhio, che possono essere sfruttati per funzioni particolari:
  - **D10 (Blue LED)**: LED blu collegato al GPIO D10, attivo in modalità LOW.
  - **D8, D9**: Utilizzati per l'interfaccia SWDIO su GPIO D8 e SWDCLK su GPIO D9.
  - **D10, D11, D12, D13**: Utilizzati per la comunicazione SPI:
    * D10 (nCS), D11 (MOSI), D12 (MISO), e D13 (CLK).
  - **A1**: Utilizzato per il monitoraggio della tensione della batteria collegata ai Pin + e -.
  - **D25**: funzione di nCS per abilitare il controllo dell'SPI Flash integrata.
  - **D0, D1**: UART RX1 (D0)/TX1 (D1) utilizzati per comunicare con il modulo GNSS CAM M8Q di cui parleremo tra poco.
  - **D3, A4**: Per gestire gli interrupt del BMA400.
  - **D4, D5, A0**: Utilizzati per la gestione dei pin Enable (D5), PPS (D4), GNSS backup enable (A0), relativi al modulo GNSS.
  - **D2**: Utilizzato per leggere la pressione del pulsante Utente.

A seconda delle board, la **GPIO** può variare per numero di tipologie di pin di input ed output e protocolli di comunicazione da essi gestiti, tra quelli nominati poc'anzi riportiamo brevemente:

- **Protocolli di comunicazione**:
  - **UART (Universal Asynchronous Receiver/Transmitter)**: Un protocollo di comunicazione seriale che permette la trasmissione di dati tra dispositivi. È noto per la sua semplicità e l'uso in molti tipi di comunicazione hardware. Utilizza cavi seriale per la trasmissione di dati, tipicamente un paio di fili per la trasmissione (**TX**) e la ricezione (**RX**).

– **I2C (Inter-Integrated Circuit)**: Un protocollo di bus seriale utilizzato per collegare dispositivi a bassa velocità, come sensori e altri componenti, con un controller principale. Supporta più dispositivi **slave** connessi a un singolo bus. Impiega un cavo seriale bidirezionale per i dati (**SDA**) e un cavo per il clock (**SCL**), connessi a tutti i dispositivi sul bus.

– **SPI (Serial Peripheral Interface)**: Un protocollo di comunicazione seriale che consente il trasferimento di dati ad alta velocità tra un dispositivo **master** e uno o più dispositivi **slave**. Utilizza un cavo per il clock (**CLK**), un cavo Master Output Slave Input (**MOSI**), un cavo Master Input Slave Output (**MISO**) e un chip select (**CS**) per ogni dispositivo slave.

– **SWDIO (Serial Wire Debug)**: Una interfaccia di debug a due fili utilizzata principalmente per il debug dei micro-controllori, offrendo funzionalità di programmazione e di test del sistema. Richiede soltanto due fili: uno per la trasmissione dei dati (**SWDIO**) e uno per il clock (**SWCLK**).

Passando alla sensoristica di bordo, i cui nomi abbiamo avuto modo di vedere precedentemente, elenchiamo:

Figura 3.6: Ricevitore GNSS ublox CAM-M8 [4]

Figura 3.7: LoRa SX1276 [5]

- **Modulo GNSS U-blox CAM-M8**:

  Il modulo CAM-M8 di u-blox (fig.3.6) è un modulo antenna GNSS compatto e ad alte prestazioni. Supporta la ricezione simultanea di tre sistemi

GNSS (GPS/Galileo con BeiDou o GLONASS), garantendo un'eccellente precisione di posizionamento. I moduli CAM-M8 sono progettati con un'antenna GNSS integrata (nella versione montata sul CAT è possibile montare un'antenna esterna per migliorarne la ricezione), hanno dimensioni ridotte e capacità di soppressione delle interferenze. Questi moduli sono adatti sia per applicazioni industriali che consumer che richiedono la ricezione simultanea dalle flotte satellitari GPS/Galileo e GLONASS o BeiDou.

- **Modulo LoRa Semtech SX-1276 [6]**:

  I moduli SX1276/77/78/79 (fig.3.7) utilizzano il modem LoRa per comunicazioni a lungo raggio e alta immunità alle interferenze, mantenendo basso il consumo energetico. Grazie alla tecnica di modulazione LoRa brevettata da Semtech, questi moduli raggiungono una sensibilità superiore a -148dBm, ottimizzando costi e materiali. Con una potenza di +20dBm e alta sensibilità, offrono un bilancio di collegamento leader nel settore, ideale per applicazioni che richiedono lunga portata o robustezza. LoRa supera le tecniche di modulazione convenzionali in termini di blocco e selettività, bilanciando efficacemente portata, immunità alle interferenze e consumo energetico.

- **Altri sensori**:

  Tra gli altri sensori, non meno importanti ma semplicemente non utilizzati nel MGT, troviamo il **sensore BMA400** [7], accelerometro a bassissimo consumo, ed il **BME280**[8], sensore di umidità e pressione atmosferica.

3.1.1.3 Hardware esterno

Passando alla descrizione dell'hardware esterno, montato sulla millefori, introduciamo prima di tutto i moduli aggiuntivi e successivamente parleremo di come sono stati integrati con la CAT e del loro funzionamento. Tra i moduli troviamo:

- **Modulo DC-DC Step-up**:

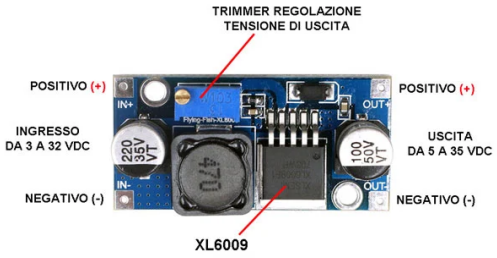

Figura 3.8: DC-DC Step-up [9]

I moduli **DC-DC** sono dispositivi elettronici progettati per convertire una tensione di ingresso in una tensione di uscita di diverso valore. Questi moduli sono specificatamente utilizzati per adattare i livelli di tensione in applicazioni che operano con corrente continua, come suggerito dal nome stesso "DC-DC" (*Direct Current to Direct Current*). Se il modulo è in grado di incrementare la tensione di uscita rispetto a quella di ingresso, viene definito come modulo **step-up**. Al contrario, se il modulo abbassa la tensione di uscita rispetto a quella di ingresso, viene classificato come modulo **step-down**. Nel contesto del nostro progetto, il Cricket fornisce un'uscita a 3,3V, ma il modulo HX711**3.11**, che esamineremo più dettagliatamente in seguito, necessita di una tensione di alimentazione di 5V. Per superare questa differenza di tensione, facciamo affidamento su un modulo DC-DC step-up che eleva la tensione di uscita da 3,3V a 5V, garantendo così la compatibilità energetica tra il Cricket e il modulo HX711.

- **Modulo Level-Shifter**:

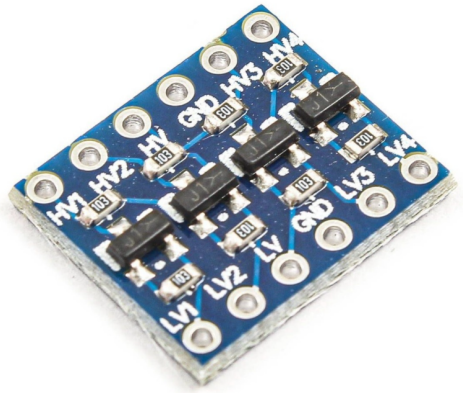

Figura 3.9: Level Shifter 3V3 - 5V [10]

I moduli "*Level Shifter*" sono dispositivi progettati per facilitare la comunicazione tra apparecchiature elettroniche che operano a differenti livelli di tensione. Questi moduli sono particolarmente utili quando si desidera connettere dispositivi che utilizzano diversi protocolli di comunicazione, come I2C o UART.

Il funzionamento del modulo Level Shifter è semplice ma efficace. Esso è dotato di diversi canali suddivisi in due categorie: da una parte, i canali LV (Low Voltage), e dall'altra, i canali HV (High Voltage). I canali LV sono destinati alla connessione con il dispositivo che opera a una tensione più bassa, ad esempio 3,3V. Qui si collegano i cavi corrispondenti al protocollo scelto (come I2C o UART) del dispositivo a bassa tensione. Viceversa, i canali HV sono riservati al dispositivo che funziona a una tensione più alta, come 5V.

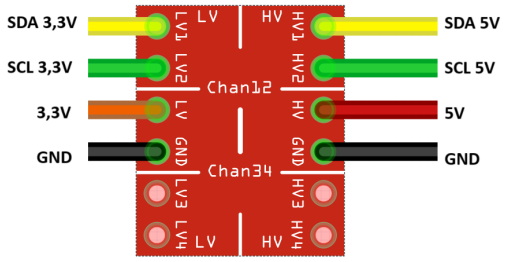

Figura 3.10: Level Shifter con cavi collegati

Nella configurazione illustrata del modulo *Level Shifter*, troviamo quattro canali dedicati alla comunicazione, identificati come **LV1-HV1**, **LV2-HV2**, **LV3-HV3**, e **LV4-HV4**, oltre a un canale specifico per l'alimentazione, contrassegnato come **LV-HV** (con una corrispondenza **GND-GND** per il **Ground**).

Per esemplificare l'utilizzo del modulo in una situazione concreta, consideriamo il caso in cui si desidera stabilire una comunicazione **I2C** (che si basa sull'uso di due linee di comunicazione, **SDA** e **SCL**) tra un dispositivo operante a 3,3V, posizionato sul lato sinistro, e uno a 5V, sul lato destro del modulo.

Per realizzare ciò, è necessario collegare la linea **SDA** del dispositivo a 3,3V al corrispondente canale **LV1** sul lato a bassa tensione del modulo e la linea **SDA** del dispositivo a 5V al corrispondente canale **HV1** sul lato ad alta tensione. Questa connessione è rappresentata dai cavi gialli nella figura 2.73.10. Analogamente, la linea **SCL** a 3,3V va collegata al canale **LV2** e la corrispondente linea **SCL** a 5V al canale **HV2**, utilizzando cavi verdi per indicare questa connessione.

Per completare l'installazione, si collegano le tensioni di alimentazione dei dispositivi (3,3V e 5V) al canale dedicato LV-HV e si effettua il collegamento a terra (**GND**) attraverso il canale **GND-GND**. Questa configurazione assicura che la comunicazione tra i dispositivi avvenga correttamente, rispettando le differenti tensioni di operatività.

- **Modulo HX711**:

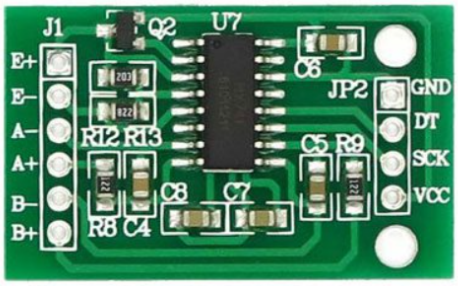

Figura 3.11: Modulo HX711[11]

Il modulo **HX711** svolge un ruolo cruciale nell'acquisizione di dati da una cella di carico, convertendo il segnale analogico in segnale digitale per una più semplice elaborazione. Questo modulo opera a 5V, e qui si evidenzia l'importanza del **DC-DC Step-up** fig.**3.8** per l'alimentazione e del **Level Shifter** fig.**3.9** per facilitare la comunicazione tra il Cricket, che opera a 3.3V, e l'**HX711**.

Come mostrato nella figura **3.11**, l'**HX711** dispone di quattro pin dedicati alla comunicazione con un microcontrollore e sei pin per il collegamento con una cella di carico. L'analisi dettagliata dei sei pin relativi alla cella di carico verrà affrontata successivamente.

Concentrandoci sui quattro pin di comunicazione, al di là dei pin VCC (5V) e GND per l'alimentazione, troviamo il pin **DT**, che è responsabile della trasmissione dei dati, e il pin **SCK**, utilizzato per il segnale di clock. Questa configurazione permette una comunicazione efficace e precisa tra il microcontrollore e il modulo **HX711**, assicurando l'accurata conversione dei segnali provenienti dalla cella di carico.

- **Modulo interruttore MOSFET**:

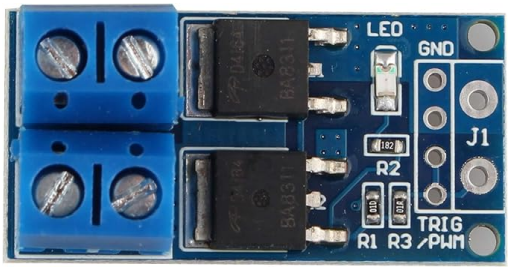

Figura 3.12: Interruttore MOSFET [12]

Dato l'obiettivo di massimizzare l'autonomia del sistema e considerando la complessità nel gestire direttamente l'alimentazione dei moduli esterni attraverso il Cricket, come si fa con i moduli interni, abbiamo adottato una soluzione l'utilizzo di un interruttore basato su **MOSFET**. Questo dispositivo sfrutta due *MOSFET* integrati per funzionare effettivamente come un interruttore, consentendo di interrompere l'alimentazione ai moduli esterni quando non sono in uso. Questo approccio permette di ridurre significativamente il consumo energetico del sistema, contribuendo così a soddisfare il requisito di un'elevata autonomia. Come possiamo osservare nella figura **3.12**, l'interruttore a *MOSFET* è progettato per una facile gestione dell'alimentazione dei circuiti esterni. Sul lato sinistro del dispositivo, troviamo dei morsetti appositamente predisposti per il collegamento del **VCC** e del **GND**, ovvero i capi dell'alimentazione che si desidera controllare. Sul lato destro, invece, sono posizionati due pin: uno dedicato all'input digitale, che consente di comandare l'accensione (**ON**) o lo spegnimento (**OFF**) dell'alimentazione, e un altro pin per il collegamento al **GND** della board che lo controlla. Un elemento aggiuntivo di questo interruttore è la presenza di un **LED** integrato, che fornisce un *feedback* visivo immediato riguardo lo stato dello *switch*, facilitando così il monitoraggio dell'operatività del dispositivo.

- **Hardware esterno secondario**: Nel progetto sono inclusi moduli secondari, impiegati esclusivamente durante la fase di testing. Questi sono stati disattivati al termine dello sviluppo del progetto.

– **MPU-6050**:

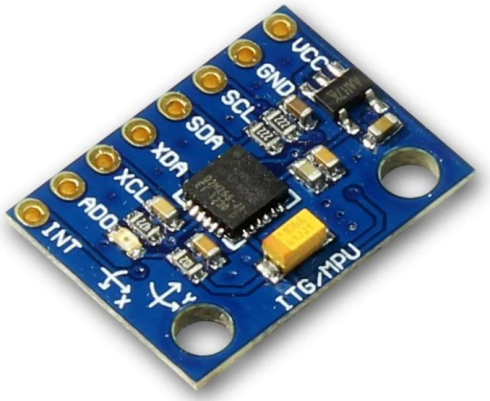

Figura 3.13: Modulo MPU6050 [13]

Il modulo **MPU6050** è un sensore MEMS (Micro Electro Mechanical Systems) che integra un **giroscopio** a 3 assi e un **accelerometro** a 3 assi, offrendo quindi 6 gradi di libertà (6 DOF) in un unico dispositivo. Utilizza l'interfaccia **I2C** per la comunicazione con la board. Questo modulo è capace di misurare sia **l'accelerazione lineare** che la **rotazione angolare**, rendendolo estremamente utile per una vasta gamma di applicazioni. L'idea iniziale di utilizzare un accelerometro, inizialmente quello interno al Cricket, per calcolare il peso dei mitili immersi in acqua rappresentava un approccio innovativo, utilizzando le leggi della fisica relative all'accelerazione. Tuttavia, si sono presentate delle difficoltà nell'utilizzo dell'accelerometro interno, che hanno portato al tentativo vano dell'installazione del modulo esterno MPU6050 che ha presentato problemi simili, inoltre avrebbe introdotto complessità significative nel progetto, come la necessità di calibrare accuratamente il sensore per le condizioni specifiche dell'acqua e di gestire la conversione dei dati di accelerazione in misure di forza e quindi in peso. La soluzione adottata è stata quella di determinare il peso mediante l'uso esclusivo della cella di carico, eseguendo una serie di misurazioni ripetute su un arco temporale esteso. Questo approccio ha permesso di calcolare la media dei valori campionati, garantendo

così una stima più accurata e affidabile del peso.

– **ADS1115**:

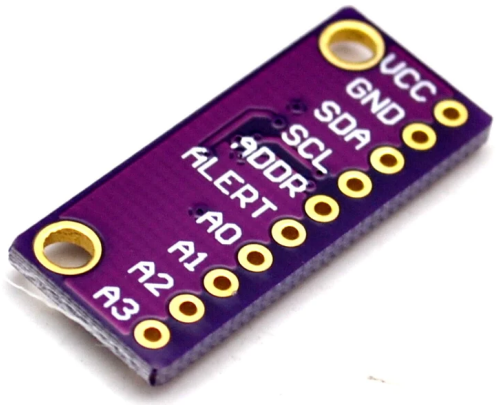

Figura 3.14: Modulo ADS1115 [14]

Il modulo **ADS1115** funge da convertitore analogico-digitale (**ADC**) a **16-bit**, collegandosi mediante il protocollo **I2C** alla scheda principale. Questo modulo è dotato di quattro ingressi analogici (**A0**, **A1**, **A2**, **A3**), permettendo così la lettura simultanea di quattro segnali analogici differenti. In aggiunta, dispone di un pin **ALERT**, programmabile per emettere un *alert* quando il valore di tensione supera o scende al di sotto di soglie predefinite, e un pin **ADDR**, che permette di modificare l'indirizzo I2C del dispositivo collegandolo a **VCC**, **GND**, **SDA**, o **SCL**.

Nel corso della fase di **testing** del progetto, il modulo ADS1115 è stato impiegato per **monitorare la tensione** di una **batteria** al fine di determinarne il **livello di carica**. Il dispositivo Cricket è in grado di leggere la tensione della batteria attraverso l'ingresso analogico indicato con il pin +. Tuttavia, quando la scheda è alimentata attraverso una fonte esterna via porta **micro-USB**, ad esempio un computer utilizzato per la programmazione, il circuito interno si occupa di disconnettere automaticamente la batteria, precludendo di fatto la possibilità di misurare la sua tensione.

Quando si monitora una batteria al **litio** con tensione nominale, ad esempio, di **3.7V**, è possibile determinare il livello di carica osservando la variazione della tensione. La batteria è considerata completamente carica, al **100%**, quando la tensione raggiunge circa **4.2V**. Man mano che la carica si riduce, anche la tensione scende progressivamente: si registra una tensione di **3.7V** quando la carica è ridotta al **50%**, fino ad arrivare a circa **3.2V**, indicativo di una batteria quasi esaurita, prossima allo **0%**.

- **Cella di carico**:

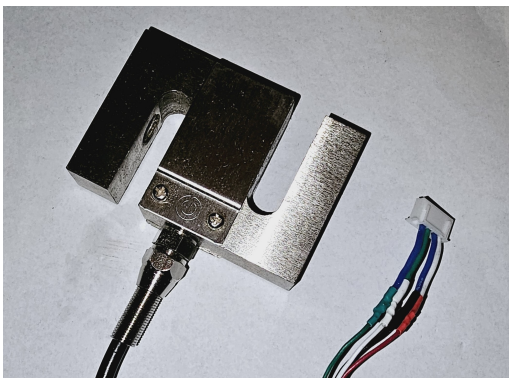

Figura 3.15: Cella di carico YZC-516C

Uno dei componenti chiave del progetto è rappresentato dalla cella di carico. Il modello in questione è l'YZC-516C che consente di leggere fino a 100Kg.

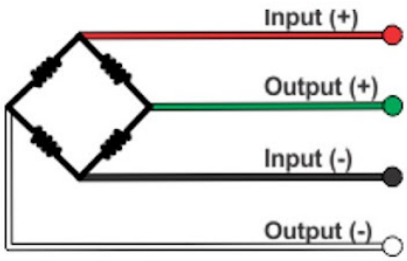

Figura 3.16: Circuito e pinout dell'YZC-516C [15]

Come illustrato nella figura 3.16, la cella di carico **YZC-516C** è caratterizzata da un design del circuito particolarmente semplice che permette la lettura del sensore mediante quattro cavi, che nel progetto sono stati raggruppati con un connettore **JST**.
Possiamo ora completare quanto detto sul modulo **HX711** illustrato in 3.11, si nota che la parte sinistra del modulo dispone di sei pin. L'HX711 è progettato per essere compatibile con vari modelli di celle di carico, che non necessariamente utilizzano tutti i pin disponibili. Specificatamente, l'**YZC-516C** sfrutta solo quattro di questi sei pin, vengono di seguito riportati i collegamenti:

- cavo **rosso** al pin **E**+
- cavo **nero** al pin **E-**
- cavo **bianco** al pin **A-**
- cavo **verde** al pin**A**+

Ciò comporta che i pin **B-** e **B**+ rimangano inutilizzati.

3.1.2 Strumenti hardware ausiliari

3.1.2.1 Gateway LoRaWAN

Figura 3.17: Sentrius RG1xx LoRaWAN Gateway[16]

Il Sentrius™ RG1xx LoRaWAN-Enabled Gateway si è rivelato un elemento cruciale all'interno del nostro progetto, fungendo da ponte vitale per la trasmissione di dati dal MGT alla piattaforma TTN (The Things Network), che introdurremo più avanti, tramite LoRaWAN. Questo gateway, progettato per accogliere segnali fino a 10 miglia di distanza utilizzando la tecnologia LoRaWAN e per sincronizzare questi dati con il cloud attraverso connessioni Wi-Fi o Ethernet, ha svolto un ruolo fondamentale per lo sviluppo del progetto.

Il Gateway offre supporto per un'ampia gamma di frequenze operative, adeguandosi così alle esigenze specifiche di diverse regioni geografiche. Nel contesto del nostro progetto, è stata selezionata la frequenza di 868 MHz, riservata per l'uso nell'Unione Europea. Tale scelta garantisce la conformità alle normative locali e ottimizza la trasmissione dei dati. L'accesso al Gateway avviene in maniera intuitiva, simile alla configurazione di un router domestico, mediante un browser web e inserendo l'indirizzo IP del dispositivo.

Le impostazioni cruciali del Gateway, che meritano una menzione particolare, includono la configurazione del Wi-Fi, essenziale per stabilire una connessione

Internet affidabile. Inoltre, le frequenze operative e la scelta del servizio cloud rappresentano aspetti fondamentali della configurazione. A questo proposito, come anticipato, il nostro sistema si affida a The Things Network (TTN), una piattaforma leader nel settore dell'IoT, per la gestione e l'analisi dei dati trasmessi dal Gateway. Queste configurazioni, accessibili e personalizzabili attraverso un'interfaccia web user-friendly, rendono il Gateway estremamente versatile e perfettamente integrato nell'ecosistema del nostro progetto.

3.1.3 Software

Dopo aver esaminato l'hardware configurato per il nostro progetto, è il momento di focalizzarci sul software sviluppato per gestirlo.

Il codice è stato strutturato in modo da essere il più modulare possibile, segregando le principali funzionalità utilizzate nel progetto in differenti sezioni. Questo approccio consente una maggiore flessibilità e facilità di manutenzione, permettendo di modificare o estendere specifiche funzioni senza influenzare l'intero sistema.

3.1.3.1 Ambienti e linguaggi di programmazione utilizzati

Prima di approfondire i dettagli dello sviluppo software del MGT, è opportuno presentare brevemente gli strumenti e le tecnologie impiegate. Questa sezione fornisce una panoramica degli ambienti di sviluppo e dei linguaggi di programmazione selezionati per portare a termine il progetto.

- **GitHub**:

Figura 3.18: Logo GitHub

Piattaforma di hosting per la gestione del codice sorgente. Consente agli sviluppatori di collaborare su progetti, condividendo e revisionando il codice in modo efficiente. Oltre alla gestione del codice, offre funzionalità come la tracciabilità dei problemi, richieste di pull, e gestione di progetti, rendendolo uno strumento indispensabile per team di sviluppo di tutte le dimensioni. La piattaforma facilita l'open source, consentendo a chiunque di contribuire a progetti esistenti o di iniziare i propri, promuovendo così l'innovazione e la collaborazione nella comunità globale degli sviluppatori.

- **Arduino**:

Figura 3.19: Logo Arduino

Il CAT, pur basandosi sull'architettura STM32, è fornito con documentazione orientata ad Arduino. In particolare, viene offerto uno sketch di esempio che dimostra come attivare e utilizzare tutti i sensori presenti sulla scheda.

- **Visual Studio Code**:

Figura 3.20: Logo Visual Studio Code

Ambiente di sviluppo scelto per la programmazione del *CAT*. Sebbene sia presente l'**IDE Arduino** ufficiale, si è optato per Visual Studio Code per i suoi numerosi vantaggi, tra cui un'interfaccia utente più sofisticata, capacità avanzate di editing del codice, e una vasta gamma di estensioni disponibili, tra cui appunto Arduino, che facilitano la gestione di diverse piattaforme e librerie. Questa scelta ha permesso di migliorare l'efficienza dello sviluppo e di sfruttare al meglio le funzionalità avanzate offerte da Visual Studio Code rispetto all'IDE Arduino.

- **Wiring**: La piattaforma Arduino3.19 adotta il linguaggio di programmazione Wiring per lo sviluppo dei suoi sketch. Basato sul linguaggio C++, Wiring offre una sintassi semplificata e un ambiente di programmazione

estremamente accessibile, che lo rende particolarmente adatto sia per i principianti sia per chi si approccia per la prima volta al mondo dell'elettronica e della programmazione. Grazie a questa fondamenta in C++, gli utenti possono sfruttare una vasta gamma di funzioni standard e librerie, facilitando l'implementazione di funzionalità quali la lettura di sensori, il controllo di motori e LED, e la comunicazione con altri dispositivi. Questa combinazione di accessibilità e potenza rende Arduino una scelta popolare per una vasta gamma di progetti di prototipazione e applicazioni hobbyistiche, permettendo agli utenti di realizzare progetti complessi con uno sforzo relativamente limitato.

- **TheThingsNetwork**:

Figura 3.21: TheThing Network[17]

The Things Stack rappresenta un pilastro fondamentale nell'ecosistema LoRaWAN, fungendo da server di rete LoRaWAN e svolgendo un ruolo critico per qualsiasi soluzione basata su questa tecnologia. Questa piattaforma non solo facilita la comprensione e l'implementazione delle basi di LoRaWAN per gli utenti, ma offre anche una suite completa e di livello aziendale che integra sia le funzionalità di Network Server che di Application Server, come delineato nell'architettura di riferimento LoRaWAN.

The Things Stack è progettato per gestire in modo sicuro e scalabile milioni di dispositivi LoRaWAN in contesti produttivi, incorporando servizi e strumenti avanzati per l'amministrazione dei dispositivi, la gestione delle politiche di sicurezza, e la facilitazione della comunicazione bidirezionale tra i dispositivi finali e le applicazioni. Questa piattaforma fornisce un'infrastruttura robusta per lo sviluppo di soluzioni IoT, supportando una va-

sta gamma di scenari applicativi, dalla gestione di asset e monitoraggio ambientale, fino alla smart agriculture e alle città intelligenti.

In sintesi, The Things Stack si distingue per la sua affidabilità, scalabilità e sicurezza, offrendo agli sviluppatori e alle aziende gli strumenti necessari per implementare soluzioni IoT complesse e di ampia portata con la tecnologia LoRaWAN. La sua architettura all'avanguardia e la sua conformità agli standard LoRaWAN garantiscono che le soluzioni basate su The Things Stack siano non solo tecnicamente solide, ma anche future-proof e adattabili alle esigenze in continua evoluzione del mercato IoT.

3.1.4 Sviluppo software

Dopo aver delineato gli strumenti software e i linguaggi di programmazione selezionati per il nostro progetto, è il momento di focalizzarci sul cuore del MGT: il workflow del firmware sviluppato, qui sotto in figura 3.22. Questa parte della discussione mira a fornire una visione chiara e dettagliata del flusso operativo che anima il dispositivo, evidenziando come le varie componenti software interagiscono per garantire funzionalità, efficienza e affidabilità.

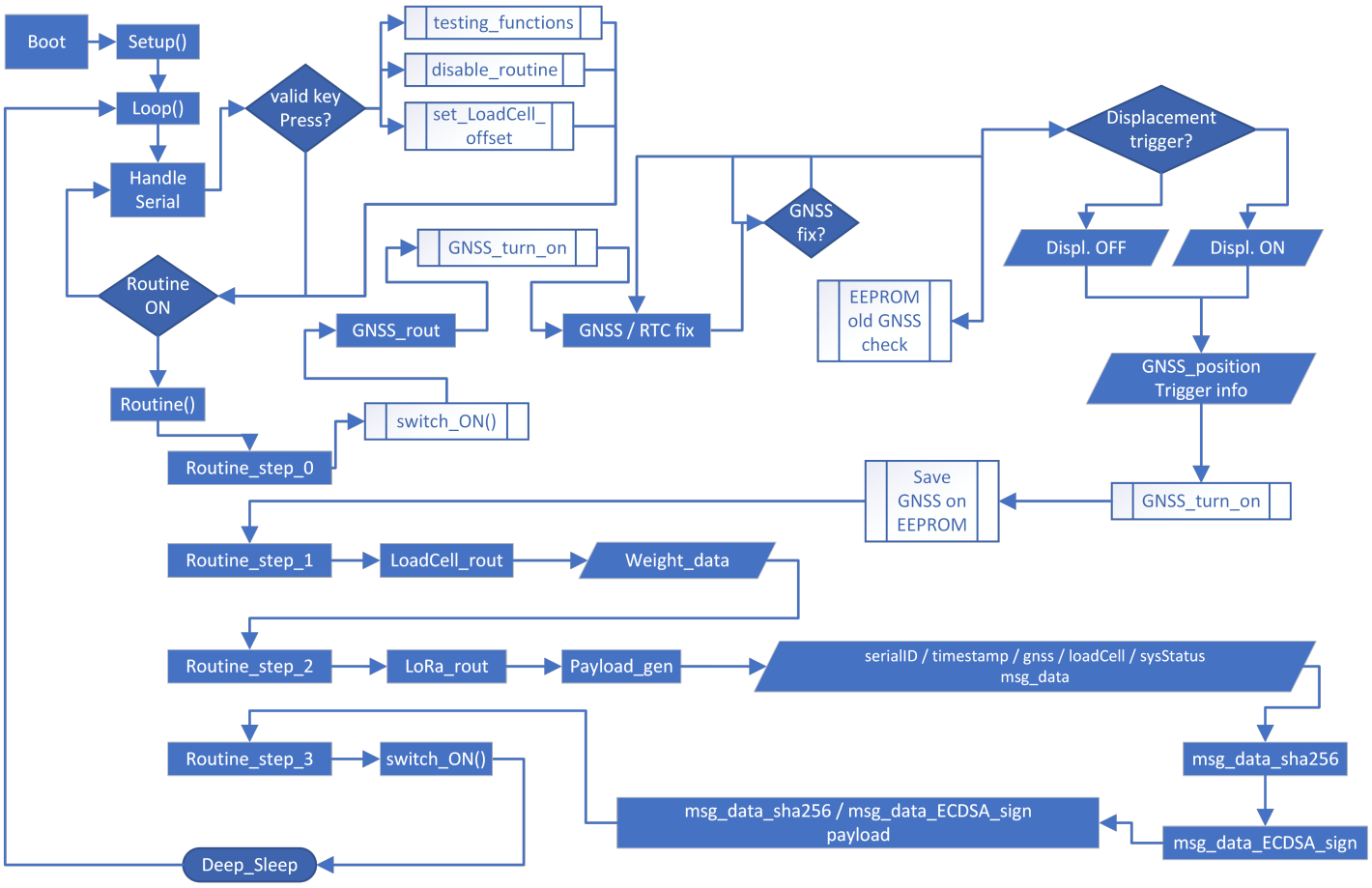

Figura 3.22: Workflow codice

Per rispondere efficacemente alle esigenze del progetto, il Mussel Grow Tracker esegue una serie di operazioni programmate su base oraria, racchiuse all'interno di un processo che definiamo come "routine". Questa routine è strutturata in quattro fasi principali:

- **Routine**:
  - **GNSS**: Il **primo step** della routine ha come obiettivi principali:
    * Acquisire dati di geolocalizzazione con precisione elevata attraverso il modulo **GNSS**[3.6], fondamentale per il tracciamento e il monitoraggio del dispositivo.
    * Sincronizzare l'orologio interno del **CAT**, utilizzando il modulo **RTC** (Real Time Clock), grazie alle informazioni temporali precise fornite dal sistema **GNSS**. Questo assicura che tutte le operazioni siano temporizzate con estrema accuratezza.
  - **Cella di carico**: Il **secondo step** si concentra sull'acquisizione precisa dei dati di peso dei mitili mediante la cella di carico, interfacciata attraverso il modulo **HX711** [11].
  - **Trasmissione LoRa**: Nel **terzo step**, si eseguono quattro operazioni chiave:
    * **Composizione del messaggio** che aggrega tutti i dati raccolti per la trasmissione.
    * **Calcolo dell'hash** del messaggio utilizzando l'algoritmo **SHA-256**, per garantire l'integrità dei dati inviati.
    * **Firma digitale** del messaggio attraverso l'algoritmo **ECDSA**[18], per assicurare autenticità dei dati.
    * **Invio del messaggio firmato** attraverso la tecnologia **LoRa**.
  - **Chiusura Routine**: Il **quarto** ed ultimo **step** riguarda l'ottimizzazione del consumo energetico:
    * **Disattivazione** di tutti i **moduli esterni** attraverso l'impiego di uno **switch** (3.12), minimizzando così il consumo energetico in fase di inattività.
    * Impostazione del **CAT** in modalità **deep sleep**, riducendo al minimo il **dispendio energetico** e prolungando la **durata della batteria** tra le attività di monitoraggio.

3.1.5 Codice

In questa sezione, esploreremo e analizzeremo in dettaglio i diversi componenti che costituiscono la struttura del codice, mettendo in luce le loro funzionalità e il loro ruolo all'interno dell'intero sistema. Per rendere le porzioni di codice riportate più fruibili, verranno omessi alcuni comandi di log utilizzati per il debugging.

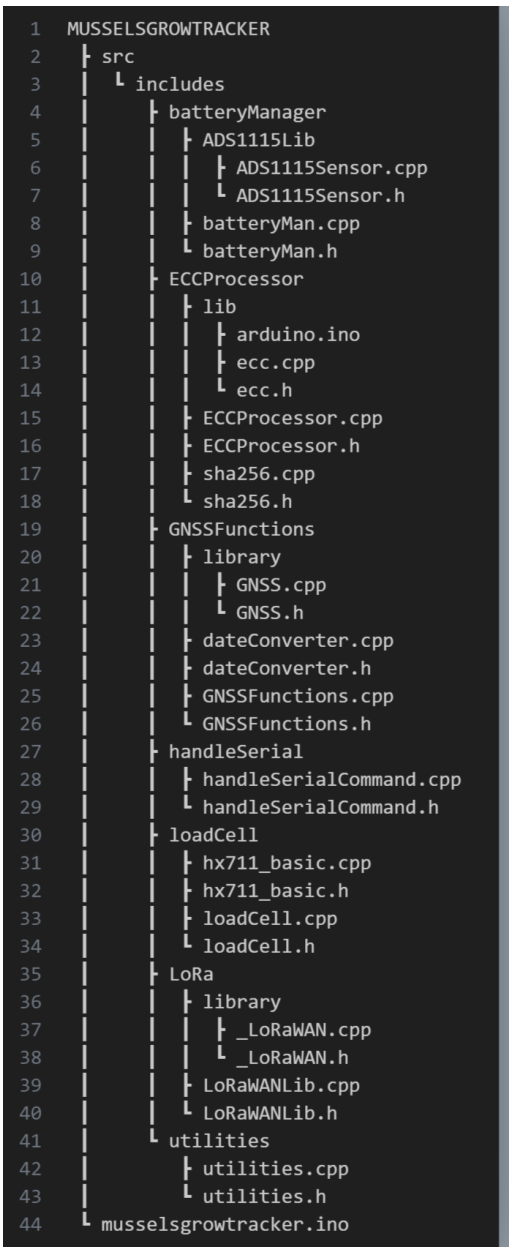

Figura 3.23: Struttura del codice

Come precedentemente sottolineato, il codice è stato progettato con un'attenzione particolare alla modularità. Questo approccio è chiaramente visibile nella struttura del progetto, illustrata nella figura.

Nella directory principale si trova il file principale **musselgrowtracker.ino**, con estensione per Arduino[19], affiancato dalla cartella **src**.

Questa organizzazione facilita la gestione e l'evoluzione del codice, consenten-

do una facile integrazione di nuove funzionalità e una manutenzione più efficace.

All'interno della cartella **src** è presente un ulteriore cartella **include** in cui sono presenti delle librerie personalizzate, suddivise anch'esse in cartelle) che fanno riferimento alle librerie standard fornite con i diversi moduli hardware utilizzati. Questo approccio ci ha consentito di sviluppare delle librerie su misura, ottimizzate per soddisfare le specifiche necessità del nostro progetto.

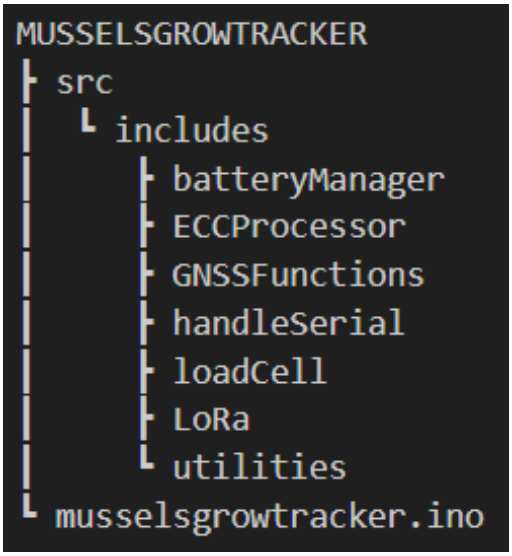

Figura 3.24: Librerie custom

Tra le librerie personalizzate visibili in figura abbiamo:

- **batteryManager**: Gestione della batteria
- **ECCProcessor**: Firma ECDSA
- **GNSSFunctions**: Gestione servizio GNSS
- **handleSerial**: Gestione dei comandi da seriale
- **loadCell**: Gestione cella di carico
- **LoRa**: Gestione per i messaggi LoRa
- **utilities**: Questa cartella racchiude un insieme di funzioni ausiliarie progettate per facilitare e supportare le operazioni delle altre librerie. Un esem-

pio notevole è la funzione `log()`, impiegata universalmente dalle librerie menzionate, la cui definizione si trova all'interno di **utilities**.

Per offrire una comprensione più approfondita, adotteremo un approccio di analisi *Bottom-Up*, esaminando dettagliatamente le librerie a partire dalle componenti di base. Progressivamente, ci muoveremo verso l'analisi del file `musselsgrowtracker.ino`, che coordina l'intera logica del progetto.
Per rendere più legibili le librerie, riporteremo il solo header contendente le definizioni delle funzioni

- **batteryManager**:

Listing 3.1: Libreria batteryManager

```
1  #include <STM32L0.h>
2  #include "ADS1115Lib\ADS1115Sensor.h"
3
4  class powerManagement {
5      public:
6          void begin();
7          float get_VDDA();
8          uint8_t get_BatPercent();
9          float get_VBAT();
10
11     private:
12         float batt = 0.00;
13         uint8_t batteryPercentage = 0;
14         uint8_t BATTERY_ERROR_CODE = 150;
15 };
16 extern powerManagement pwrMan;
17 }
```

Come precedentemente menzionato, il CAT interrompe l'alimentazione dalla batteria quando è connesso attraverso la porta micro-USB. Questa carat-

teristica ci ha portati ad integrare il modulo esterno ADS1115, come illustrato in [3.14]. L'inizializzazione di questo modulo è affidata alla funzione `begin()`, che si incarica di prepararlo per l'uso.

La libreria mette a disposizione le funzionalità per ottenere sia una stima percentuale della carica della batteria (`get_BatPercent()`) che la misurazione della sua tensione (`get_VBAT()`).

La funzione dedicata alla lettura della batteria gestisce automaticamente il processo, sfruttando entrambe le opzioni disponibili: la lettura diretta dal pin V+ del Cricket o, in caso di tensione non valida (al di sotto dei 3V), la lettura attraverso il modulo ADS1115.

- **ECCProcessor**:

Listing 3.2: Libreria ECCProcessor

```
1  #include <Arduino.h>
2  #include <cstddef>
3  #include <cstdint>
4  class MyECC {
5      public:
6          void generateSHA256Hash(String data, uint8_t*
               hash_out);
7          void sign_message(uint8_t *hash_to_sign);
8          void printHex(const uint8_t* data, size_t
               length);
9          uint8_t* get_msgSigned();
10         size_t get_msgSignedSize();
11     private:
12         void p(char *fmt, ... );
13         void dump(char *text, uint8_t *d);
14         uint8_t msgSigned[64];
15 };
16 extern MyECC eccProcessor;
```

L'algoritmo ECDSA (Elliptic Curve Digital Signature Algorithm) e le relative funzionalità crittografiche sono incapsulate nella libreria `ECCProcessor`, sviluppata appositamente per facilitare l'implementazione di operazioni crittografiche. Ecco un'analisi dettagliata delle sue principali caratteristiche:

- **Metodi Pubblici**:
  - **Generazione di Hash SHA-256**: La libreria offre il metodo `generateSHA256Hash`, che permette di trasformare una stringa di dati in un hash SHA-256. Questo è particolarmente utile per creare un'impronta digitale univoca dei dati, che può poi essere utilizzata nelle operazioni di firma digitale.
  - **Firma Digitale**: Attraverso il metodo `sign_message`, la libreria consente di generare una firma digitale basata su un hash precedentemente calcolato. Questo garantisce che i dati non siano stati alterati da quando sono stati firmati, offrendo un meccanismo di autenticazione forte.
  - **Visualizzazione dei Dati in Esadecimale**: Il metodo `printHex` fornisce un modo semplice per visualizzare dati binari, come hash e firme digitali, in formato esadecimale. Questa funzionalità è utile per il debugging e la verifica delle operazioni crittografiche.
  - **Accesso alla Firma Digitale**: La funzione `get_msgSigned` rende accessibile l'array che contiene la firma digitale generata, permettendo di utilizzarla per verifiche o per la trasmissione a terzi.
  - **Dimensione della Firma**: Il metodo `get_msgSignedSize` informa sulla dimensione dell'array della firma digitale, che per l'ECDSA è tipicamente di 64 byte. Questo dettaglio è cruciale per la gestione corretta della memoria e per l'elaborazione dei dati firmati.

– **Metodi Privati**:

  * **Utility per la Stampa e il Debugging**: I metodi privati `p` e `dump` offrono strumenti aggiuntivi per la stampa formattata di stringhe e per il dumping di dati binari, facilitando così il debugging e la validazione delle operazioni crittografiche.
  * **Memorizzazione della Firma**: L'array `msgSigned`, protetto all'interno della classe, custodisce la firma digitale generata, assicurando che rimanga inalterata e sicura fino al suo utilizzo.

- **GNSSFunctions**: Data la lunghezza del codice di questa libreria, molte parti sono omesse, lasciando al lettore la libertà di consultare il codice disponibile su github [20].

Listing 3.3: Libreria GNSSFunctions

```
#include "Arduino.h"
#include "utility/gnss_api.h"

class GNSSLocation {
public:
    enum GNSSfixType {
        ...
    };

    enum GNSSfixQuality {
        ...
    };
    ...
private:
    gnss_location_t _location;
};

class GNSSSatellites {
public:
    GNSSSatellites(const gnss_satellites_t *
        satellites);
    GNSSSatellites();

    unsigned int count() const;
    ...
private:
```

```
26              gnss_satellites_t _satellites;
27          };
28
29          class GNSSClass {
30          public:
31              ...
32              GNSSClass();
33              void begin(Uart &uart, GNSSmode mode, GNSSrate
                    rate = RATE_1HZ);
34              void end();
35              bool setAntenna(GNSSantenna antenna);
36              ...
37              bool suspend();
38              bool resume();
39              bool busy();
40
41              bool location(GNSSLocation &location);
42              bool satellites(GNSSSatellites &satellites);
43
44              void enableWakeup();
45              void disableWakeup();
46              .....
47              bool timeSatFix();
48
49          private:
50              Uart *_uart;
51              uint32_t _baudrate;
52              bool _wakeup;
53              gnss_location_t _location_data;
54              volatile uint32_t _location_pending;
55              gnss_satellites_t _satellites_data;
```

```
56              volatile uint32_t _satellites_pending;
57              ...
58              void receiveCallback(void);
59              void completionCallback(void);
60              ...
61              static void ppsCallback(class GNSSClass*);
62          };
63
64          //extern GNSSSatellites GNSSSAt;
65          //extern GNSSClass GNSS;
66
67          #endif /* _GNSS_H */
```

Per quanto riguarda il GNNS, troviamo tre classi principali per la gestione dei dati GNSS (Global Navigation Satellite System), che permettono l'acquisizione e l'elaborazione di informazioni relative alla posizione, ai satelliti e alla configurazione del modulo GNSS. Di seguito, una panoramica semplificata delle sue caratteristiche principali:

- **Classe GNSSLocation**:

  Gestisce le informazioni di posizione ottenute dal sistema GNSS, il tipo (es. 3D_FIX, 2D_FIX,...) e qualità del fix, numero di satelliti, data e ora attraverso il quale è possibile sincronizzare l'RTC di sistema, coordinate geografiche (latitudine, longitudine), altezza e velocità.

- **Classe GNSSSatellites**:

  Fornisce dettagli sui satelliti visibili, inclusi l'identificativo del satellite (SVID), il rapporto segnale-rumore (SNR), l'elevazione e l'azimut e altri stati operativi.

- **Classe GNSSClass**:

  Configura e controlla il modulo GNSS. Permette di impostare la modalità di funzionamento (NMEA o UBLOX), la frequenza di aggior-

namento, il tipo di antenna (interna o esterna), la costellazione di satelliti utilizzata (GPS, GLONASS, Galileo e BeiDou), il tipo di piattaforma (per esempio, auto o aereo), e altre opzioni per ottimizzare l'acquisizione dei dati GNSS. Supporta anche la sospensione e la ripresa dell'attività del modulo GNSS, funzione importantissima per il raggiungimento dei nostri requisiti, la gestione degli interrupt per il risveglio e l'elaborazione degli eventi relativi alla posizione e ai satelliti.

- **handleSerial**: Per la presente libreria, vengono forniti sia il file di intestazione (header) che il file di implementazione (source file, .cpp), in quanto la presenza di una singola funzione principale richiede un'esposizione completa per garantire una piena comprensione delle sue funzionalità e del suo funzionamento.

Listing 3.4: libreria HandleSerial.h

```
1          #include <Arduino.h>
2          #include "..\utilities\utilities.h"
3
4          extern bool DEBUG;
5          extern uint8_t LOG_LEV;
6          extern volatile uint8_t Step;
7          extern bool enableRoutine;
8          void handleSerialCommand(char command);
9          #endif // HANDLE_SERIAL_COMMAND_H();
10         }
```

Listing 3.5: Funzione handleSerialCommand(...) in HandleSerial.cpp

```
1          void handleSerialCommand(char command) {
2                switch (command) {
3                  case 'u':
4                    log(uidCode.get_UID_String(), 1);
5                  break;
```

```
6                   case 'r':
7                     log("Data on EEPROM: " + String(gnssEeprom.
                          readLatitude(), 6) + ", " + String(
                          gnssEeprom.readLongitude(), 6), 1);
8                   break;
9                   case 'y':
10                    msgService.sendMsg();
11                  break;
12                  ...
13                  ...
14                  default:
15                    log("invalid command!", 1);
16                }
17              }
```

Il codice presentato è parte della libreria HandleSerial.cpp, responsabile della gestione dei comandi seriali ricevuti che esegue azioni specifiche in base al carattere ricevuto. Per brevità abbiamo omesso parte del codice riportando di seguito una descrizione dettagliata delle operazioni gestite da ciascun comando:

- `void handleSerialCommand(char command)`: Questa funzione prende come input un carattere che rappresenta il comando da eseguire. Possiamo dividere i comandi i due gruppi:
  - **Comandi di configurazione**:
    - `'1'` e `'2'`: Imposta il livello di log per il debug.
    - `'B'`: Avvia il processo di calibrazione della cella di carico utilizzando un peso noto.
    - `'d'`: Abilita o disabilita la modalità di debug.
    - `'e'`: Abilita o disabilita l'esecuzione della routine principale, permettendo di mettere in pausa le operazioni periodiche.

- `'F'`: Modifica il tipo di fix GNSS utilizzato per l'acquisizione dei dati.
- `'l'`: Attiva o disattiva un LED interno come indicatore visivo.
- `'x'`: Calibra la cella di carico impostando l'offset corrente come zero.
- `'z'`: Imposta la scala di misurazione della cella di carico basandosi su un peso noto (bottiglia d'acqua da un 1Kg).
- `'h'`: Help, stampa i comandi disponibili.

* **Comandi di debug e testing**:
  - `'b'`: Legge e stampa la percentuale stimata di carica della batteria.
  - `'c'`: Esegue una lettura dalla cella di carico e stampa il valore misurato.
  - `'G'`: Attiva o disattiva il modulo GNSS in base allo stato corrente.
  - `'g'`: Aggiorna i dati GNSS e li stampa.
  - `'I'`: Richiede e stampa informazioni di stato del modulo LoRaWAN.
  - `'i'`: Avvia la scansione del bus I2C per identificare i dispositivi connessi e stampa i risultati.
  - `'m'`: Attiva o disattiva un mosfet per gestire l'alimentazione di componenti esterni e stampa lo stato corrente.
  - `'o'`: Avvia la lettura dell'orario corrente dal modulo GNSS e lo stampa.
  - `'P'`: Stampa il valore di taratura corrente della cella di carico.
  - `'p'`: Avvia una procedura di deep sleep per un periodo di tempo specificato.
  - `'q'`: Crea un nuovo messaggio di dati basato sui sensori attuali, lo firma digitalmente per poi stamparlo su seriale.

- `'r'`: Legge e stampa i dati di latitudine e longitudine salvati nell'EEPROM.
- `'S'`: Avvia la procedura di invio del messaggio firmato tramite LoRa.
- `'s'`: Esegue le operazioni necessarie per completare e inviare il messaggio LoRa, inclusa la gestione delle dimensioni del messaggio firmato.
- `'T'`: Sincronizza l'orologio del dispositivo con l'orario fornito dal modulo GNSS.
- `'t'`: Avvia una lettura della temperatura interna del dispositivo e la stampa.
- `'u'`: Stampa l'identificativo unico del dispositivo (UID).
- `'v'`: Legge e stampa il valore di tensione VDDA, offrendo un'indicazione della tensione di alimentazione del microcontrollore.
- `'y'`: Invoca il metodo `sendMsg()` per inviare un messaggio predefinito.

- **loadCell**: Esaminiamo ora la libreria che gestisce l'elemento centrale del progetto: la cella di carico.

Listing 3.6: libreria HandleSerial.h

```
#ifndef LOAD_CELL_H
#define LOAD_CELL_H
#include "Arduino.h"

class LoadCell
{
    public:
        void begin();
        void setOffset();
        void setScale(float weightKnown);
        void calibrate(float weightKnown);
        float read_weight(byte times);
        void print_tare();
        float get_LastWeightReading();
    private:
        float lastWeightReading = -1;

};

extern LoadCell loadCell;
#endif /* LOAD_CELL_H */
```

  - **begin()**: Inizializza la cella di carico, in particolare il modulo **HX711**. Questa funzione viene richiamata all'avvio del CAT per preparare il dispositivo ad effettuare misurazioni con la cella di carico.
  - **setOffset()**: La funzione imposta l'offset della cella di carico, che rappresenta il valore di output quando non è applicato alcun peso.

Questo parametro è fondamentale per la taratura accurata del sensore, poiché variazioni, anche minime, come la posizione della cella di carico o l'influenza di elementi esterni quali il cavo di collegamento, possono alterare il valore di offset. Tale regolazione permette di assicurare misurazioni precise indipendentemente da queste variabili esterne.

  - **setScale(float weightKnown)**: Imposta la scala di misurazione della cella di carico. Questo metodo richiede un parametro che rappresenta un peso noto, utilizzato per calibrare la cella di carico. Permette quindi di ottenere il peso in grammi dal valore grezzo ottenuto dalla cella di carico attraverso l'**HX711**. Nel progetto è stata utilizzata una bottiglia d'acqua da 1Kg per tarare la cella di carico.
  - **calibrate(float weightKnown)**: Calibra la cella di carico utilizzando un peso noto. Questa funzione consente di ottenere misurazioni precise, impostando la cella di carico in base al peso fornito come parametro.
  - **read_weight(byte times)**: Legge il peso rilevato dalla cella di carico. Il parametro 'times' indica quante misurazioni prendere per calcolare la media, migliorando così l'accuratezza della lettura. Nel CAT vengono effettuati 100 campionmenti per ogni richiesta di letture, mediamente vengono impiegati circa 6-7 secondi per lettura.
  - **print_tare()**: Stampa il valore di taratura corrente della cella di carico. Utile per controlli diagnostici o per verificare l'offset impostato durante la calibrazione.
  - **get_LastWeightReading()**: Restituisce l'ultima lettura del peso effettuata dalla cella di carico. Questo permette di recuperare l'ultimo valore misurato senza necessità di effettuare una nuova lettura.

- **LoRa**:

  Questa libreria è progettata per gestire tutte le operazioni relative alla comunicazione LoRa, dalla creazione e configurazione dei messaggi fino al

loro invio. Vediamo in dettaglio le sue funzionalità e come contribuisce al processo di comunicazione:

Listing 3.7: Libreria LoRa

```
#ifndef LoRaWANLib_h
#define LoRaWANLib_h

#include "Arduino.h"
#include "library\_LoRaWAN.h"
#include "..\ECCProcessor\ECCProcessor.h"
#include "..\loadCell\loadCell.h"

class LoRaWANLib {
public:
    LoRaWANLib(const char *appEui, const char *appKey
        , const char *devEui);
    void begin();
    void getInfo();
    void sendMessage(byte* payload, size_t
        payloadSize);

private:
    const char *_appEui;
    const char *_appKey;
    const char *_devEui;
};

class LoRaPayload {
public:
    uint8_t* createDataMsg(const uint8_t* SerialID,
        uint32_t TimestampLinux, uint8_t* GpsData,
```

```
                uint16_t LoadCellMeasurement, uint8_t
                SysStatus);
25
26              String bytesToHexString(const uint8_t* buffer,
                    size_t bufferSize);
27              String bytesToString(const uint8_t* buffer,
                    size_t bufferSize);
28              void hexToSHA256(String data, uint8_t* hash_out)
                    ;
29
30              size_t get_MsgSignedSize() const;
31              size_t get_MsgSize();
32              uint8_t* get_DataMsg() const;
33              uint8_t* get_DataSignedMsg() const;
34              uint8_t* get_CombinedMsg() const;
35
36              String get_msgStr();
37              String get_msgHex();
38              uint8_t* get_msgHashUint8();
39              String get_msgHashString();
40
41              void set_msgByte(uint8_t* msg, size_t msgSize);
42              void setDataMsg(const uint8_t* msg, size_t size);
43              void setDataSignedMsg(const uint8_t* msg, size_t
                    size);
44              void setCombinedMsg(const uint8_t* msg, size_t
                    size);
45
46              String set_msgStr(String msg);
47              String set_msgHex(String msg);
48
```

```
49              uint8_t* sendMergedMsg();
50
51              void logBytes(const uint8_t* data, size_t size,
                    const char* message);
52
53          private:
54              void clearData(); // Potrebbe essere necessario
                    rimuovere o aggiornare in base alla nuova
                    logica
55              void clearDataMsg(); // Pulisce dataMsg
56              void clearDataSignedMsg(); // Pulisce
                    dataSignedMsg
57              void clearCombinedMsg(); // Pulisce combinedMsg
58
59              String msgToSendStr;
60              String msgToSendHex;
61              String msgToSendHashStr;
62
63              uint8_t* dataMsg = nullptr;
64              size_t dataMsgSize = 0;
65
66              uint8_t* dataSignedMsg = nullptr;
67              size_t dataSignedMsgSize = 0;
68
69              uint8_t* combinedMsg = nullptr;
70              size_t combinedMsgSize = 0;
71
72              uint8_t msgToSendHash[32];
73          };
74
75          class MSGSend {
```

```
76              public:
77                  void sendMsg();
78
79              private:
80                  void sendMsgECC();
81          };
82
83          extern LoRaPayload MSGPayload;
84          extern MSGSend msgService;
85          extern LoRaWANLib LoRaWANManager;
86
87          #endif
```

– **LoRaWANLib**:

Questa classe gestisce la configurazione e la comunicazione con la rete LoRaWAN. Viene inizializzata con le credenziali di accesso alla rete (**appEui**, **appKey**, **devEui**) e offre metodi per:

* Iniziare la comunicazione con la rete LoRaWAN (**begin**).
* Ottenere informazioni sullo stato della connessione (**getInfo**).
* Inviare messaggi attraverso la rete LoRaWAN (**sendMessage**).

– **LoRaPayload**:

Responsabile composizione dei messaggi LoRa, inclusa la creazione, la generazione dell'HASH associato e l'invio, la firma del messaggio è affidato, come abbiamo visto poco fa, alla libreria **ECCProcessor**. I suoi metodi principali includono:

* Creazione di un messaggio da inviare (‘**createDataMsg**‘).
* Conversione dei dati del messaggio in formato esadecimale o ASCII (‘**bytesToHexString**‘, ‘**bytesToString**‘).
* Generazione dell'hash del messaggio utilizzando l'algoritmo SHA-256 (‘**hexToSHA256**‘).

* Fusione del messaggio con la sua firma digitale e successivo invio tramite LoRa (‘sendMergedMsg‘).
* Pulizia dei dati del messaggio quando non sono più necessari (‘**clearData‘**, ‘**clearDataMsg**‘, ‘**clearDataSignedMsg**‘, ‘**clearCombinedMsg**‘).

– **MSGSend**:

Classe dedicata all’invio dei messaggi. Include il metodo ‘sendMsg‘ per l’invio dei messaggi, potenzialmente dopo averli firmati digitalmente.

– **Istanze globali**:

* **MSGPayload**: Istanza globale della classe ‘LoRaPayload‘, utilizzata per la gestione dei messaggi LoRa.
* **msgService**: Istanza globale della classe ‘MSGSend‘, utilizzata per inviare messaggi.
* **LoRaWANManager**: Istanza globale della classe ‘LoRaWANLib‘, utilizzata per gestire la comunicazione con la rete LoRaWAN.

- **utilities**: La libreria utilities.h fornisce una serie di funzioni ausiliarie e classi di supporto che facilitano varie operazioni all'interno del progetto, quali la gestione della modalità debug, la gestione dei timer, il controllo del LED, la scansione del bus I2C, e altro ancora. Vediamo in dettaglio le sue componenti principali:

Listing 3.8: Funzione di configurazione iniziale 'setup()'

```
1  ...
2  class bytePackaging {
3    public:
4      byte packData();
5  };
6
7  class timerManager {
8  public:
9      void startTimer();
10     void stopTimer();
11     void updateCurrTimer();
12     void setEnable();
13     void setDisable();
14     bool getTimerStatus();
15     unsigned long elapsedTimer(unsigned int delay_ms =
           0);
16     void saveElapsedTimer();
17     unsigned long getSavedElapsedTime();
18     bool delay(unsigned int delay_ms);
19
20 private:
21     bool activeTimer;
22     unsigned long startTime;
23     unsigned long currentTime;
```

```
24          unsigned long elapsedTime;
25          unsigned long savedElapsedTime;
26      };
27
28      struct DebugState {
29        bool mode;
30        int level;
31      };
32
33      void setDebugMode(bool value);
34      void setDebugLevel(uint8_t value);
35      void log(String log, uint8_t LOG_LEVEL);
36      void deep_sleep(uint32_t seconds);
37      float getVDDA();
38      float getTemp();
39
40      class intLED
41      {
42        public:
43          void begin();
44          void intLED_on();
45          void intLED_on_off();
46          void intLED_off();
47          void visualLog();
48      };
49
50      class WireScan
51      {
52        public:
53          void begin();
54          void scan();
```

```
55          private:
56            void i2cScan();
57        };
58
59        class systemDate
60        {
61          public:
62            void printRtcDate();
63            void printGnssDate();
64          private:
65        };
66
67        class mosSwitch
68        {
69          public:
70            void begin();
71            bool getMosfetState();
72            bool turn_on_off();
73            bool turn_off();
74            bool turn_on();
75
76          private:
77            bool MOSFET_STATE = false;
78            uint8_t MOSFET_PIN = 8;
79            bool mosfet_state_change(bool STATE);
80        };
81
82
83        class UID {
84        public:
85            static const int UID_LENGTH = 12;
```

```
86
87          static byte* get_UID();
88          static String get_UID_String();
89      };
90
91      class GNSSEeprom {
92        public:
93            void saveGNSSCoordinates(double lat, double lon);
94            double readLatitude();
95            double readLongitude();
96            //void formatEEPROM();
97
98        private:
99            void EEPROMWrite(int address, double value);
100           double EEPROMRead(int address);
101           const uint8_t EEPROM_START_LOC_BYTE = 0;
102           uint8_t latAddress = 0;
103           uint8_t lonAddress = 10;
104     };
105     ...
```

– **DEBUG e LOG**:

* `setDebugMode` e `setDebugLevel`: Queste funzioni consentono di controllare la modalità debug e il livello di log. Possono essere utilizzate per abilitare o disabilitare la stampa di messaggi di debug e per definire il livello di dettaglio dei log, migliorando l'efficienza del processo di troubleshooting.

* **bytePackaging**: è progettata per facilitare l'aggregazione e la manipolazione di dati in formato byte. Questa classe può essere utilizzata per impacchettare i dati raccolti dai sensori o da altre

fonti in un formato compatto e trasmissibile, ottimizzando così le operazioni di invio dati attraverso reti di comunicazione come LoRa.

* **Timer**:
  - La classe `timerManager` offre una serie di metodi per la gestione temporizzata delle operazioni. Inclusi metodi per avviare e fermare il timer, controllare se un determinato intervallo di tempo è trascorso, e per mantenere traccia del tempo trascorso. Ciò è particolarmente utile per le operazioni che richiedono precisione temporale, come il campionamento di dati a intervalli regolari.
* **LED Interno**:
  - La gestione del LED interno attraverso la classe `intLED` consente di segnalare visivamente lo stato dell'applicazione o di confermare l'esecuzione di particolari operazioni, migliorando l'interfaccia utente hardware.
* **Scansione I2C**:
  - `WireScan` implementa una funzione di scansione del bus I2C per identificare i dispositivi collegati. Questa caratteristica è fondamentale per la diagnostica e la configurazione iniziale del sistema, permettendo di verificare la corretta connessione dei componenti I2C.
* **Visualizzazione della data**:
  - Le funzioni fornite dalla classe `systemDate` permettono di acquisire e visualizzare la data e l'ora correnti, utili per timestamping.
* **Controllo Mosfet**:
  - `mosSwitch` offre metodi per gestire l'alimentazione di componenti esterni via MOSFET, una funzionalità chiave per il risparmio energetico e il controllo di dispositivi esterni.

* **Identificativo Unico (UID)**:
    - Fornisce metodi per recuperare e visualizzare l'identificativo unico del dispositivo (`UID`), essenziale per la sicurezza e l'identificazione univoca dei dispositivi in rete.
* **EEPROM e Coordinate GNSS**:
    - La classe `GNSSEeprom` permette di salvare e leggere le coordinate GNSS, facilitando la persistenza di dati critici tra i riavvii del dispositivo.

3.1.5.1 musselgrowtracker.ino

Andiamo ad analizzare prima di tutto il codice principale **musselgrowtracker.ino**:

Listing 3.9: Funzione di configurazione iniziale 'setup()'

```
1   void setup() {
2     Serial.begin(115200);
3     setDebugMode(DEBUG);
4     setDebugLevel(LOG_LEV);
5     uidBytes = UID::get_UID();
6     mosfetSwitch.begin();
7     mosfetSwitch.turn_on();
8     gnssHandler.restoreOldLoc();
9     gnssHandler.toggleGNSS(true);
10    loadCell.begin();
11    intBlueLED.begin();
12    gnssHandler.initializeStmGNSSPins();
13    gnssHandler.configureGNSS();
14    LoRaWANManager.begin();
15    mosfetSwitch.turn_off();
16  }
```

Iniziamo esaminando la funzione **setup()**, un elemento fondamentale negli sketch Arduino che esegue tutte le operazioni iniziali indispensabili per preparare l'ambiente di esecuzione. Questa funzione è chiamata automaticamente all'avvio del programma e ha il compito di configurare le impostazioni di base, come la definizione dei pin di input/output e l'inizializzazione di moduli esterni. La sua esecuzione è unica e precede il ciclo principale del programma, fornendo le basi necessarie per garantire che tutte le componenti hardware e software siano pronte per la fase operativa.

Elenchiamo brevemente tutti i comandi eseguiti:

- **Serial.begin(115200)**: Inizializzazione della comunicazione seriale a 115200 baud, fondamentale per la trasmissione di dati tra il dispositivo Arduino e il computer, cruciale per il debugging e il monitoraggio.
- **setDebugMode(DEBUG) e setDebugLevel(LOG_LEV)**: Configurazione della modalità e del livello di debug per fornire feedback dettagliati sugli eventi di sistema, facilitando la diagnosi e la risoluzione dei problemi.
- **UID::get_UID()**: Acquisizione dell'identificativo unico del dispositivo, un passaggio cruciale per l'identificazione sicura del dispositivo all'interno di una rete o di un sistema.
- **mosfetSwitch.begin() e mosfetSwitch.turn_on()**: Inizializzazione e attivazione del mosfet per controllare l'alimentazione dei moduli esterni e gestire il consumo energetico del sistema, il mosfet viene subito abilitato per permettere al modulo **hx711** di essere alimentato per poter essere configurato in questa fase.
- **gnssHandler.restoreOldLoc() e gnssHandler.toggleGNSS(true)**: Configurazione e attivazione del modulo GNSS per l'acquisizione dei dati di geolocalizzazione, essenziali per il tracciamento e la sincronizzazione temporale.
- **loadCell.begin() e intBlueLED.begin()**: Preparazione della cella di carico e del LED interno per il loro utilizzo nel progetto, attraverso

le inizializzazioni necessarie per le misurazioni di peso e l'indicazione del LED che permette di avere un feedback visivo del funzionamento della scheda.

- **gnssHandler.initializeStmGNSSPins() e gnssHandler.configureGNSS()**: Configurazione dei pin e delle impostazioni del modulo GNSS, preparando il modulo per le operazioni di tracciamento precise.
- **LoRaWANManager.begin()**: Avvio della configurazione del modulo LoRaWAN per stabilire le basi per la comunicazione e la trasmissione dei dati.
- **mosfetSwitch.turn_off()**: Disattivazione del mosfet a seguito di tutte le configurazioni iniziali, per minimizzare il consumo energetico prima dell'entrata nel ciclo principale del programma, ottimizzando l'efficienza energetica.

Passiamo ora al **loop()**:

Listing 3.10: Funzione loop principale

```
1       void loop() {
2         LOG_END_LOOP = false;
3         if (blueLedTimer.delay(1000)) {
4           intBlueLED.intLED_on_off();
5           log(String(Step),2);
6         }
7         routine();
8         if (Serial.available()) {
9           char c = Serial.read();
10          handleSerialCommand(c);
11        }
12        if(LOG_END_LOOP) {
13          log(LOG_END_LINE_STRING, 1);
14        }
15      }
```

La funzione **loop()** rappresenta il cuore dell'esecuzione di uno sketch Arduino, essendo la sezione in cui il codice viene eseguito in modo continuo dal momento in cui il dispositivo viene acceso fino a quando non viene spento o resettato. Dopo l'inizializzazione effettuata nella funzione **setup()**, la **loop()** prende il controllo, eseguendo ripetutamente le istruzioni al suo interno. Questa funzione permette di monitorare costantemente gli input, gestire le attività in tempo reale e controllare i dispositivi esterni, adattando il comportamento del sistema alle condizioni che cambiano dinamicamente.

Vediamo ora nel dettaglio le istruzioni eseguite al suo interno:

- **Inizializzazione del ciclo**:
  - La variabile `LOG_END_LOOP` è inizialmente impostata a `false`. Questo `flag` serve a segnalare il completamento delle operazioni all'in-

terno del ciclo `loop`. Quando impostato a `true`, consente la stampa di una specifica linea di log sulla porta seriale, facilitando così l'interpretazione degli output durante le fasi di debug e offrendo una chiara indicazione della conclusione del ciclo di esecuzione.

- **Gestione del LED**:
  - Un timer controlla se è trascorso un secondo (1000 millisecondi). Se sì, alterna lo stato di un LED interno utilizzando `intBlueLED.intLED_on_off()`, questo permette di capire lo stato del dispositivo: il LED infatti cambia il proprio stato ad ogni esecuzione del loop(), nei momenti in cui smette di lampeggiare, significa che il Mussel Grow Tracker sta compiendo operazioni che richiedono tempo, dalla lettura dela cella di carico, alla firma dei messaggi, se invece il led rimane fisso per un periodo molto prolungato (oltre i 30 secondi), il dispositivo è probabilmente bloccato.
- **Esecuzione della routine principale**:
  - Se abilitata, viene chiamata la funzione `routine()` per eseguire le tutte le operazioni di cui abbiamo parlato precedentemente.
- **Gestione dei comandi seriali**:
  - Verifica la disponibilità di dati in ingresso sulla porta seriale con `Serial.available()` e, nel caso fossero presenti, legge il carattere inviato con `Serial.read()` e lo passa alla funzione `handleSerialCommand(c)` (che vedremo successivamente), che elabora il comando ricevuto.
- **Log di fine ciclo**:
  - Se la variabile `LOG_END_LOOP` viene impostata a `true` durante l'esecuzione del ciclo, viene registrato un messaggio di log `LOG_END_LINE_STRING`, segnalando il termine delle operazioni per quel ciclo, raggruppando così visivamente tutti i log dei vari cicli.

Adesso è arrivato il momento di analizzare la tanto citata **routine()**:

Listing 3.11: Funzione di routine principale

```
1   void routine() {
2     if (!enableRoutine) {
3           return; // if routine disable exits from function
4       }
5     if(routineTimer.delay(1000)) {
6       log("ROUTINE:\n\tSTEP:" + String(ROUTINE_STEP) + "\n
            \tITERATION_NUMBER: " + String(
            ROUTINE_ITERATION_NUMBER), 1);
7       if(ROUTINE_STEP == 0) {
8         GNSS_routine();
9       }
10      if(ROUTINE_STEP == 1) {
11        LoadCell_routine();
12      }
13      if(ROUTINE_STEP == 2) {
14        SendMessage_routine();
15      }
16      if(ROUTINE_STEP == 3) {
17        log("Routine complete\n\tDeep sleeping for 1 hour
              ...", 1);
18        ROUTINE_STEP = 0;
19        ROUTINE_ITERATION_NUMBER++;
20        deep_sleep(3600); // Sleep for 3600 seconds or 1
              hour
21      }
22    }
23  }
```

La funzione routine() è strutturata per eseguire una serie di compiti sequen-

ziali. Ecco l'analisi dettagliata della funzione, strutturata per una chiara comprensione:

- **Controllo dell'abilitazione della routine**:
  - Viene effettuato un controllo iniziale sulla variabile `enableRoutine`. Se questa è impostata su `false`, la funzione termina immediatamente, evitando l'esecuzione delle operazioni previste se la routine è disabilitata. Questo meccanismo permette di agevolare le operazioni di debug tramite i comandi da seriale (`handleSerialCommand(c)`).
- **Esecuzione temporizzata della routine**:
  - La funzione avanza solo quando il timer `routineTimer.delay(1000)` indica il trascorrere di un intervallo di 1000 millisecondi (1 secondo), assicurando l'esecuzione delle operazioni a cadenze regolari. Questa strategia previene l'esecuzione eccessiva di letture dai sensori, le quali potrebbero necessitare di tempi prolungati per fornire risultati attendibili. Ad esempio, il modulo GNSS [21] può richiedere diversi minuti per stabilire un fix satellitare accurato; pertanto, leggere i dati a un ritmo non adeguatamente controllato si tradurrebbe in un consumo energetico inutile senza ottenere dati utili.

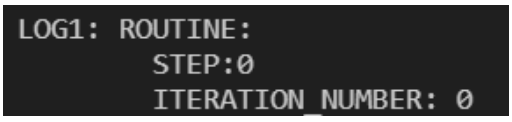

Figura 3.25: Log routine

  - Viene registrato un log che indica l'inizio della routine, mostrando il valore corrente di `ROUTINE_STEP` e `ROUTINE_ITERATION_NUMBER`, fornendo una chiara indicazione di quante routine sono state eseguite con successo e dello step attualmente in elaborazione.

– **Esecuzione delle operazioni specifiche della routine**:

* La funzione valuta il valore di `ROUTINE_STEP` per determinare quale operazione specifica eseguire:

Listing 3.12: Logica di esecuzione della routine

```
1               if(routineTimer.delay(1000)) {
2                   log("ROUTINE:\n\tSTEP:" + String(
                        ROUTINE_STEP) + "\n\
                        tITERATION_NUMBER: " + String(
                        ROUTINE_ITERATION_NUMBER), 1);
3                   if(ROUTINE_STEP == 0) {
4                     GNSS_routine();
5                   }
6                   if(ROUTINE_STEP == 1) {
7                     LoadCell_routine();
8                   }
9                   if(ROUTINE_STEP == 2) {
10                    SendMessage_routine();
11                  }
12                  if(ROUTINE_STEP == 3) {
13                    log("Routine compleate\n\tDeep
                          sleeping for 1 min...", 1);
14                    ROUTINE_STEP = 0;
15                    ROUTINE_ITERATION_NUMBER++;
16                    deep_sleep(3600);
17                  }
18              }
```

· Se `ROUTINE_STEP` è 0, viene eseguita la funzione `GNSS_routine()`, responsabile dell'acquisizione dei dati di geolocalizzazione.

· Se `ROUTINE_STEP` è 1, viene invocata la funzione `LoadCell_routine()`, per la raccolta dei dati dalla cella di carico.

· Se `ROUTINE_STEP` è 2, si procede con l'esecuzione di `SendMessage_routine()`, dedicata all'invio dei dati tramite comunicazione LoRa.

– **Conclusione e passaggio alla fase di deep sleep**:

∗ Se `ROUTINE_STEP` è 3, ciò indica il completamento di tutte le operazioni previste nella routine. Viene quindi registrato un log che annuncia il termine della routine e l'entrata del dispositivo in modalità `deep sleep` per un tempo di un'ora (`deep_sleep(60)`), ottimizzando il consumo energetico.

Procediamo con un'analisi approfondita delle funzioni attivate nei diversi step della routine.

Nel primo step troviamo la funzione GNSS:

Listing 3.13: Funzione GNSS_routine

```
1   void GNSS_routine()
2   {
3     bool GNSFix = gnssHandler.getIsLocationFixed();
4     bool RTCFix = gnssHandler.RTCFix();
5
6     if(!GNSFix || !RTCFix)
7     {
8       gnssHandler.toggleGNSS(true);
9       bool GNSSModuleState = gnssHandler.
            getGNSSModuleState();
10      if(GNSSModuleState) {
11          gnssHandler.update();
12      }else log("RUN: GNSS OFF.", 1);
13    }else if(GNSFix || RTCFix){
14      log("GNSS COMPLETED!", 1);
15      gnssHandler.toggleGNSS(false);
16      gnssHandler.saveOnEeprom();
17      ROUTINE_STEP++;
18    }
19  }
```

La funzione `GNSS_routine()` si occupa di gestire le operazioni relative al sistema di posizionamento globale (GNSS). Di seguito sono dettagliati i passaggi principali:

– Il primo passo è quello di controllare lo stato del fix del GNSS (`GNSFix`) e dell'RTC (`RTCFix`) attraverso i metodi `getIsLocationFixed()` e

`RTCFix()`. Finchè non sono entrambi "fixati", si continua a leggere all'interno dell'if ad ogni ciclo.

- Registra la posizione attuale letta dalla memoria EEPROM, utilizzando `gnssEeprom.readLatitude()` per la latitudine e `gnssEeprom.readLongitude()` per la longitudine, e visualizza questi valori tramite la funzione `log`.
- Se la posizione GNSS o l'orario RTC non sono stati fixati (`!GNSFix || !RTCFix`), il sistema GNSS viene attivato (`gnssHandler.toggleGNSS(true)`).
  - Verifica lo stato del modulo GNSS con `gnssHandler.getGNSSModuleState()`. Se il modulo è attivo (`GNSSModuleState` è `true`), procede con con il tentativo dell'aggiornamento della posizione tramite `gnssHandler.update()`.
  - Nel processo di fix dei satelliti viene confrontata la posizione rimasta in memoria EEPROM con la posizione attualmente ottenuta e nel caso in cui la distanza tra le due posizioni sia superiore alla distanza di "sicurezza", allora viene abilitato il flag di "displacement_alarm" che verrà poi incluso nel messaggio che verrà prodotto per l'invio con LoRa.

- Nel caso in cui sia la posizione GNSS che l'orario RTC siano stati fixati con successo (`GNSFix || RTCFix`), procede come segue:
  - L'attesa prosegue fino a che il valore di **EPE** non si riduce al di sotto di una soglia predeterminata. L'**EPE** (Estimated Position Error) rappresenta una stima dell'errore sulla posizione calcolata dai sistemi di navigazione satellitare, come il GNSS. Questo valore, espresso in metri, fornisce un'indicazione di quanto la posizione reale possa discostarsi da quella indicata dal dispositivo. Un EPE più basso indica una maggiore precisione nella determinazione della posizione, mentre un valore più alto suggerisce una maggiore incertezza. La riduzione dell'**EPE** (Estimated Position Error) al di sotto di una specifica soglia

è essenziale per prevenire falsi allarmi di spostamento. Un valore di EPE elevato potrebbe, infatti, indicare un'incertezza significativa nella posizione rilevata, portando a segnalazioni imprecise di spostamento dell'oggetto monitorato. Stabilire un limite inferiore per l'EPE aiuta a garantire che solo le variazioni di posizione reali e significative vengano rilevate e segnalate come allarmi, aumentando l'affidabilità del sistema di monitoraggio nel riconoscere effettivi spostamenti. Questo approccio riduce il rischio di allarmi non necessari, migliorando l'efficacia del monitoraggio e la gestione delle risorse.

- Al raggiungimento del valore di soglia di **EPE** (Estimated Position Error), la posizione attuale viene salvata nella EEPROM. Questo passaggio, oltre a prevenire falsi allarmi di spostamento, come accennato poco fa, consente di minimizzare il numero di scritture sulla EEPROM. Salvando soltanto le posizioni effettivamente significativi, si riduce l'usura della memoria garantendo che vengano memorizzate unicamente le informazioni di reale importanza per il monitoraggio.
- Registra il completamento dell'operazione GNSS.
- Imposta lo stato del GNSS su `false` (`gnssHandler.setGNSSStatus(false)`) e spegne il modulo GNSS (`gnssHandler.toggleGNSS(false)`).
- Salva i dati GNSS nella memoria EEPROM tramite `gnssHandler.saveOnEeprom()`.
- Incrementa il passo della routine (`ROUTINE_STEP++`), preparando il sistema per l'esecuzione del passo successivo.

Nel secondo step troviamo la funzione **LoadCell**:

Listing 3.14: Funzione LoadCell_routine

```
1  void LoadCell_routine()
2  {
3    weight = loadCell.read_weight(100);
4    log("loadCell step: " + String(weight), 1);
5    ROUTINE_STEP++;
6  }
```

- **Descrizione della funzione** `LoadCell_routine()`:
  - La funzione inizia con la lettura dei dati di peso attraverso la cella di carico, utilizzando il metodo `read_weight(100)`. Questa operazione effettua una media di 100 letture per ottenere un valore medio di peso più stabile e preciso, necessari in un ambienre ondoso come il mare.
  - Il risultato della lettura, contenuto nella variabile `weight`, viene successivamente loggato attraverso la funzione `log()`, fornendo un feedback immediato sul valore di peso rilevato e soprattutto tenuto in memoria per poter essere poi utilizzato nel payload LoRa.
  - Infine, la funzione incrementa il valore della variabile `ROUTINE_STEP`, preparando il sistema per l'esecuzione dello step successivo della routine. Questa gestione sequenziale degli step assicura che ogni fase della routine sia eseguita in ordine e senza sovrapposizioni, garantendo l'integrità del processo di raccolta dati.

Nel terzo step troviamo la funzione **SendMessage**:

Listing 3.15: Funzione SendMessage_routine

```
1  void SendMessage_routine()
2  {
3    MSGPayload.sendMergedMsg();
4    ROUTINE_STEP++;
5  }
```

- **Descrizione della funzione** `SendMessage_routine()`:
  - La funzione inizia con il `merge` del messaggio contenente tutti i dati da inviare, con la firma del messaggio stesso
  - Come per gli altri punti, viene successivamente incrementato il valore della variabile `ROUTINE_STEP`, preparando il sistema per l'esecuzione dello step successivo della routine.

Questo è un esempio di payload:

```
094737303232b26f6d38000080bfcc100020ffffc0460b6de4423c73c33b1c2a
a34f33f7c09b80dd4baac6215dcf431f140a52b0f77e4fe5c19621494be802a6
  9242e6995bd26acc944dffc4ffebe978ab967c7ea8
```

racchiude in se l'aggregazione del messaggio dei dati con la sua firma, possiamo quindi definirne le due componenti principale:

- **Array dati**: Per una maggiore comprensione mostriamo la componente dei dati separata da spazi per metterne in risalto le varie componenti:

```
094737303232 5d756d38 000080bfcc100020 ffff c0
```

  - **composizione stringa dati**: Riportiamo la lettura dell'output generato dal CAT via seriale.

```
LOG1: SerialID:  [HEX]: 09 47 37 30 32 32
LOG1: TimeStamp: [HEX]: 03 7a 6d 38
LOG1: GPS Data:  [HEX]: 00 00 80 bf cc 10 00 20
LOG1: LoadCell:  [HEX]: ff ff
LOG1: SysStatus: [HEX]: c0
```

Sono state implementate tecniche di compressione del messaggio al fine di ottenere un payload dalle dimensioni ridotte al minimo. Tratteremo di questo più avanti.

- **Array firma**:

```
460b6de4423c73c33b1c2aa34f33f7c09b80dd4baac6215dcf431f140a52b0f
77e4fe5c19621494be802a69242e6995bd26acc944dffc4ffebe978ab967c7ea8
```

  – La componente della firma digitale si ottiene elaborando l'hash del messaggio tramite l'algoritmo ECDSA. Questo processo assicura che la firma sia univocamente legata al contenuto del messaggio, garantendo l'integrità e l'autenticità dei dati trasmessi.

– **Output del seriale in fase di elaborazione**:

```
ECDSA.
======
Private key: 3c1c76130d17d93b55ea4174d5c2e
          1215a44da89c7617ecb5db5054229142832
Public key x:89af3906ea92a441a7caab37ab66878fa17
          b3e3d05f8c2514622586339418d8d
Public key y:98a70b59d8ac4c3df7fdbdef6c101a9249c
          2cf3e0fe31d05afde145208add238

Public_key is valid!

Signed message: ed9a9bea53ab1639a1040a37a1f7dfeabf
             d0194f73eae23cdfa71eda9d9f87d17988
             9296fa1bb0d8f79dc6b287f6d9054fffff
             bfa187b1bcf9128a8816b8e0b0

ECDSA Processing performed in 4766 ms
Sign verify: SUCCESS

=====================================================
LOG1: msgToSend:
       [HEX]: 094737303232037a6d38000080bfcc100020ffffc0
     [HASH]: d0c63ee25689eb0073418f496b95069d20d37e7bce881b1af49b304002aeb2e9
LOG1:
msgSignedToSend:
       [HEX]: ed9a9bea53ab1639a1040a37a1f7dfeabfd0194f73
            eae23cdfa71eda9d9f87d179889296fa1bb0d8f79d
            c6b287f6d9054fffffbfa187b1bcf9128a8816b8e0b0
...
19:29:21.196 | LOG1: sendMSG)
```

```
19:29:21.196 | MSG Data signed : [HEX]: 094737303232f428c16500000000cc1000200000
19:29:21.198 | LOG1: SENDING MESSAGE:
19:29:21.199 | LOG1: PAYLOAD: '094737303232f428c16500000000cc1000200000c078dd8b
19:29:21.201 | LOG1: PAYLOAD_SIZE: '85'
...
```

Nel log si osserva il processo di generazione e convalida delle chiavi pubbliche a partire dalla chiave privata. Successivamente, viene firmato l'hash del messaggio per poi procedere con la generazione e verifica della validità della firma. Al termine di queste operazioni, si ottiene un array esadecimale che rappresenta il messaggio con i relativi dati e la sua firma digitale. L'ultimo passo consiste nell'unire questi elementi in un singolo array che li contiene entrambi. L'array risultante viene trasmesso attraverso LoRa.

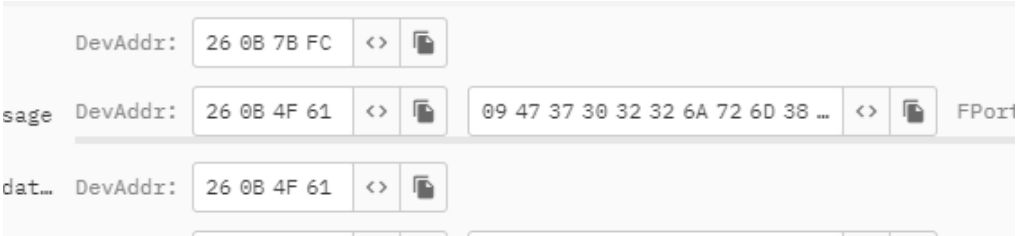

Figura 3.26: Messaggio LoRa ricevuto su TheThings Network

Se l'operazione di invio è stata completata con successo, il messaggio firmato sarà visibile sulla dashboard [[**?**]] di The Things Network.

Capitolo 4

Risultati

In questo capitolo verranno illustrati i risultati derivati dall'implementazione delle funzionalità discusse nel capitolo precedente, facendo riferimento ai test condotti per valutare il comportamento del MGT sotto diverse condizioni operative.

- **Test routine**
  - é stato effettuato un test in cui si è preso in considerazione i tempi necessari per compiere una routine dal momento dell'accensione del MGT, viene di seguito riportato l'output seriale
    - **Serial output**
      - **Output del seriale in fase di elaborazione**:

```
19:28:54.569 | ============|END LOOP|============
19:28:55.564 | LOG1: ROUTINE:
19:28:55.564 | STEP:0
19:28:55.564 | ITERATION_NUMBER: 0
19:28:55.565 | LOG1: Eeprom locVal => lat:42.42, lon: 14.15
19:28:55.566 | LOG1: getIsLocationFixed: 0 RTCFix:1
19:28:55.567 | LOG1: GNSS accuracy not yet good.
19:28:55.567 | Elapsed time: 84s
```

```
19:28:55.567 | LOG1: currentLocation:
19:28:55.567 | LAT:0.000000
19:28:55.567 | LON:0.000000
19:28:55.568 | LOG1: READ GNSS POS:
19:28:55.568 | GNSS still fixing...9 satellites found.
19:28:55.568 | isLocationFixed: 0
19:28:55.569 | ============|END LOOP|============
19:28:55.569 |
19:28:56.563 | LOG1: ROUTINE:
19:28:56.563 | STEP:0
19:28:56.563 | ITERATION_NUMBER: 0
19:28:56.565 | LOG1: Eeprom locVal => lat:42.42, lon: 14.15
19:28:56.565 | LOG1: getIsLocationFixed: 0 RTCFix:1
19:28:56.566 | LOG1: GNSS accuracy not yet good.
19:28:56.566 | Elapsed time: 85s
19:28:56.568 | LOG1: currentLocation:
19:28:56.568 | LAT:42.418103
19:28:56.568 | LON:14.153161
19:28:56.570 | LOG1: EPE: 8.92
19:28:56.601 | LOG1: GNSS Module info:
19:28:56.601 | CURRENT LOCATION: 42.4181025,14.1531596,22.790
            SATELLITES=5 DISPLACEMENT: 0.13 meters
19:28:56.601 | PREVIOUS LOCATION: 42.4181027,14.1531611,22.267
            SATELLITES=5
19:28:56.602 | DISPLACEMENT: 0.13 meters
19:28:56.602 | Connected satellites: 9
19:28:56.605 | LOG1: READ GNSS POS:
19:28:56.605 | 9 satellites found.
19:28:56.605 | isLocationFixed: 1
19:28:56.605 | GNSS data acquired in 85s.
19:28:56.605 | ============|END LOOP|============
```

Riportiamo dettagliatamente gli ultimi due cicli della routine: il ciclo immediatamente precedente alla ricezione del fix satellitare e il ciclo successivo, in cui la funzione GNSS_routine, avendo soddisfatto tutti i requisiti previsti, procede a passare il controllo allo step successivo della routine.

Il test è stato effettuato avviando il MGT all'esterno, con cielo sereno e in assenza di ostacoli (condizione migliore per la ricezione dei segnali satellitari), questo ci ha permesso di poter calcolare il tempo impiegato dal dispositivo a completare il primo step, che mediamente è quello che necessita di più tempo.

Come riportato nel log (**GNSS data acquired in 85s**), abbiamo ottenuto come risultato un tempo di attesa di **85 secondi**, ovvero **1min e 25secondi**, per effettuare il fix GNSS e per sincronizzare l'RTC.

Non riportiamo il tempo impiegato per la sincronizzazione dell'RTC, poiché questa avviene automaticamente non appena si raggiunge un numero sufficiente di satelliti, indipendentemente dal completamento del fix GNSS. Pertanto, la presenza di un fix GNSS garantisce di fatto che l'RTC sia già sincronizzato, andando di fatto a soddisfare immediatamente 2 dei 3 requisiti richiesti per completare questo step.

Il requisito dell'Errore di Posizione Previsto (**EPE**) è stato immediatamente soddisfatto nello stesso ciclo, registrando un valore di **8.92**, il che significa che, in un unico ciclo, sono stati raggiunti sia il fix GNSS che un valore di EPE accettabile.

Tale risultato ci permette di effettuare un confronto preciso tra la posizione precedente salvata in memoria (**PREVIOUS LOCATION**) e la posizione attuale (**CURRENT LOCATION**). Dai log emergono che la distanza calcolata tra le due posizioni (**DISPLACEMENT:**) è di soli **0.13 metri**, soddisfacendo così il criterio della massima differenza di distanza consentita e mantenendo di conseguenza il flag di allarme impostato su "**false**". È importante sottolineare che la distanza calcolata tra le due posizioni non considera l'altitudine.

Come abbiamo detto i requisiti per il **GNSS_routine** sono stati tutti rispettati, quindi dal ciclo successivo si parte col **secondo step** della routine

```
19:28:56.605 | ============|END LOOP|============
19:28:57.563 | ITERATION_NUMBER: 0
19:28:57.565 | LOG1: Eeprom locVal => lat:42.42, lon: 14.15
19:28:57.565 | LOG1: getIsLocationFixed: 1 RTCFix:1
19:28:57.572 | LOG1: GNSS COMPLETED!
19:28:57.585 | LOG1: Reading weight with 100 samples...
19:29:07.563 | LOG1: Calibration is advisable (command: 'x')
19:29:07.566 | LOG1: Weight: 0.00g
```

L'esecuzione dello step successivo evidenzia il completamento delle operazioni legate al GNSS, come indicato dal messaggio **"GNSS COMPLETED!"**, seguito immediatamente dall'avvio delle operazioni relative alla cella di carico, come evidenziato da **"Reading weight with 100 samples..."**.

Dall'analisi dei timestamp forniti dal software di lettura seriale, osserviamo che il passaggio da `19:28:57.585` a `19:29:07.566` richiede circa **10 secondi** (**9.98** secondi, per essere precisi), indicando che il tempo necessario per completare la misurazione del peso con la **cella di carico** è di circa **10 secondi**.

Questo step, non avendo specifici requisiti, viene eseguito senza ritardi significativi.

Completato anche questo step, passiamo al **terzo**. Considerando che in questo punto della routine vengono effettuati due compiti **cruciali** per lo scopo del progetto, li tratteremo separatamente, analizzando prima le operazioni riguardanti l'**ECDSA** e successivamente le operazioni di creazione, aggregazione ed invio del messaggio via **LoRa**.

Partiamo con l'**ECDSA**:

```
19:29:07.566 | Merging message ======================
19:29:08.808 | LOG1: Bat= No battery detected!
```

```
19:29:08.808 | LOG1: SerialID: [HEX]: 094737303232
19:29:08.808 | LOG1: TimeStamp: [HEX]: f428c165
19:29:08.808 | LOG1: GNSS Data: [HEX]: 000080bfcc100020
19:29:08.808 | LOG1: LoadCell: [HEX]: 0000
19:29:08.808 | LOG1: SysStatus: [HEX]: c0
19:29:08.812 | ECDSA
19:29:08.812 | Private key: ....
19:29:11.080 | Public key x: ....
19:29:11.081 | Public key y: ....
19:29:11.084 | Public_key is valid!
19:29:15.738 |
...         |...
19:29:15.742 | Signed message:
               78dd8b568a4347c6e81ce8c729c8265ae2f4b1f5145332e...c8cc
19:29:15.746 |
19:29:15.746 | Elapsed time: 4659 ms
19:29:21.176 | Sign verify: SUCCESS
```

Analizzando il log, scopriamo che il tempo necessario per l'esecuzione della **firma** e successiva **verifica**, richiede circa **4.6 secondi** (**Elapsed time: 4659 ms**).

Osserviamo ora la secondo parte:

```
19:29:21.177 | ====================================================
19:29:21.185 | LOG1: msgToSend:
19:29:21.185 | [HEX]: 094737303232f428c16500000000cc1000200000c0
19:29:21.185 | [HASH]: 40d60700a195e4ad3c4dbe36...c2a0
19:29:21.186 | LOG1:
19:29:21.186 | msgSignedToSend:
19:29:21.186 | [HEX]: 78dd8b568a4347c6e81ce8c7...6c8cc
19:29:21.187 | LOG1:
```

```
19:29:21.187 | combinedHexString:
19:29:21.187 | [HEX]: 094737303232f428c165000000...6c8cc
19:29:21.188 | LOG1:
19:29:21.188 | Merged message (85 bytes):  094737303232f428c...6c8cc
19:29:21.196 | LOG1: sendMSG)
19:29:21.196 | MSG Data signed : [HEX]: 094737303232f428c1...
19:29:21.198 | LOG1: SENDING MESSAGE:
19:29:21.199 | LOG1: PAYLOAD: '094737303232f428c165...'
19:29:21.201 | LOG1: PAYLOAD_SIZE: '85'
19:29:21.702 | LOG1: Routine compleate
```

Attraverso i timestamp forniti, si evidenzia che il tempo richiesto per la **composizione e la firma del messaggio** è estremamente ridotto. C'è da dire però che durante i test, abbiamo sperimentato lo scenario ottimale:

L'invio di un messaggio tramite **LoRa** richiede una **verifica preliminare** della disponibilità del canale di trasmissione, che non è garantita in ogni momento. Pertanto, si sottolinea che non è possibile stabilire a priori il tempo esatto che intercorre tra la **preparazione** del messaggio e il suo **invio** effettivo, a causa delle variabili intrinseche alla tecnologia **LoRa** che non sono prevedibili.

Passiamo infine all'ultimo **step** dove vendono effettuate le operazione per mettere il MGT in **deep sleep**.

```
19:29:21.702 | Deep sleeping for 1 min...
19:29:21.703 | LOG1: deep_sleeping...
```

Si precisa che il periodo di **deep sleep** di un solo minuto, è stato impostato solo per la fase di testing.

In conclusione dell'analisi, sommando i tempi delle varie routine (**GNSS= 85s**, **LoadCell= 10s**, **ECDSA= 4.6s**, **LoRa= 0.014s**), giungiamo a un tempo complessivo di circa **100 secondi**. Questa stima, sebbene sia influenzata dalle condizioni meteorologiche e dalla disponibilità della rete LoRa, ci permette di considerare con ragionevole sicurezza il lasso di tempo di circa 14.6 secondi per le operazioni tra la misurazione della cella di carico e la firma ECDSA come un dato affidabile.

È importante notare che il modulo GNSS offre la possibilità di collegare un'antenna esterna (configurabile tramite 'gnssHandler.configureGNSS(GNSS_ANTENNA_INTERNAL)' nel setup()), che potrebbe ridurre significativamente il tempo necessario per ottenere un fix satellitare, da noi osservato in 85 secondi.

L'analisi del tempo di esecuzione delle fasi di routine è cruciale poiché queste fasi corrispondono ai periodi di maggiore consumo energetico del MGT. Comprendere e ottimizzare la durata di queste operazioni permette di gestire in modo più efficiente la batteria del dispositivo, prolungandone l'autonomia. La minimizzazione del tempo necessario per completare ciascuna fase di routine è quindi fondamentale per ottimizzare il bilancio energetico del dispositivo e massimizzare la sua efficienza operativa nel monitoraggio dell'ambiente marino.

- **Compressione dati**

  Inizialmente, si era considerato l'uso di Ethereum-RPL per generare frame di dati, trasformando così la board in un client Ethereum. Tuttavia, lo spazio necessario per la firma occupa una porzione significativa dello spazio disponibile per il payload LoRa, motivo per cui sono state sviluppate specifiche tecniche di compressione dei dati.

  Sebbene fosse tecnicamente fattibile generare e trasmettere un frame con Ethereum-RPL, le richieste di elaborazione per generare tali firme eccedevano le capacità hardware della scheda. Questa limitazione ha portato alla ricerca di alternative più adatte, facendo ricadere la scelta sull'ECDSA. L'ECDSA, pur offrendo un livello di sicurezza comparabile, ha il vantaggio di richiedere meno spazio per la firma, il che si adatta meglio alle restrizioni di dimensione del payload LoRa.

  Nonostante l'ECDSA potesse probabilmente soddisfare i requisiti di dimensione del payload anche senza il bisogno di compressione dei dati, si è scelto di mantenere questa strategia per ottimizzare ulteriormente l'uso dello spazio e garantire la massima efficienza nella trasmissione dei dati.

  Come abbiamo visto dai precedenti log riguardanti la fase di firma mediante l'algoritmo ECDSA (li riportiamo per comodità anche qui sotto), possiamo osservare il modo in cui i dati sono stati compressi:

```
| LOG1: SerialID:  [HEX]: 09 47 37 30 32 32
| LOG1: TimeStamp: [HEX]: f4 28 c1 65
| LOG1: GNSS Data:  [HEX]: 00 00 80 bf cc 10 00 20
| LOG1: LoadCell:  [HEX]: 00 00
| LOG1: SysStatus: [HEX]: c0
```

  Come possiamo osservare dal log nuovamente riportato qui sopra, come detto in precedenza il messaggio contenente i dati i nostri interesse è così

suddiviso, le le operazioni di compressione dei dati sono state effettuate soltanto sulla voce GNSS Data e SysStatus.

- **GNSS Data**

Tabella 4.1: Conversione dei dati GNSS da float a int

Latitudine				**Longitudine**			
Valore	**Tipo**	**Byte**	**Formato**	**Valore**	**Tipo**	**Byte**	**Formato**
42.4181025	float	8	Originale	14.1531596	float	8	Originale
424181025	int	4	Compresso	141531596	int	4	Compresso

  –

  – Come illustrato nella figura, i dati **GNSS** relativi alla **latitudine** e **longitudine** sono di tipo **float**, che se trasformati direttamente, occuperebbero **8 byte** ciascuno, per un totale di **16 byte**. Per ottimizzare l'uso dello spazio nel payload LoRa, abbiamo deciso di convertire questi dati **float** in interi (**int**). Questo è stato realizzato moltiplicando i valori di latitudine e longitudine per un fattore che **elimina la virgola**, convertendo così i valori da **float** a **int**. Successivamente, trasformando l'intero in una stringa di byte, abbiamo ridotto l'occupazione **da 8 a 4 byte** per ciascuna coordinata, per un **totale di 8 byte** che contengono le informazioni complete di GNSS. Sarà poi compito della piattaforma The Things Network, attraverso calcoli matematici, ricostruire il dato originale di GNSS.

- **SysStatus**

  – La gestione dello stato del sistema, in particolare lo stato della batteria e il flag di *displacement_ alarm*, è ottimizzata attraverso l'uso della tecnica di byte packaging. Questo approccio consente una manipolazione diretta dei bit all'interno di un singolo byte, permettendo l'inserimento di più informazioni in un ridotto spazio di memoria.

Per la percentuale della batteria, che teoricamente richiederebbe un byte intero (da 0 a 100), si è optato per una compressione del dato in categorie approssimate di stato della batteria: 100%, 50%, 25%, e 10%. Questa semplificazione consente di rappresentare lo stato della batteria con soli due bit, riducendo significativamente l'uso dello spazio.

Il flag di *displacement_ alarm*, che indica se è stato rilevato uno spostamento significativo, è rappresentato da un singolo bit. Questo utilizzo parsimonioso del byte non solo conserva lo spazio prezioso del payload.

Tabella 4.2: Rappresentazione della percentuale della batteria in bit

Percentuale Batteria	**Bit**
100%	11
50%	10
25%	01
10%	00

Andando quindi ad analizzare i risultati ottenuti grazie alla compressione dei dati, osserviamo che il messaggio contenente i dati di monitoraggio utilizzano complessivamente 21 byte che sommati alla firma ECDSA portano ad un utilizzo complessivo di 85 byte su 256 disponibili nel payload.

Tabella 4.3: Confronto tra dati originali e compressi

Dato	**Dimensione Compressa (byte)**	**Dimensione Originale (byte)**
SerialID	6	6
TimeStamp	4	4
GNSS Data	8	16
LoadCell	2	2
SysStatus	1	3
ECDSA	64	64
TOTALE	85	95

Capitolo 5

Conclusione e sviluppi futuri

Nel corso di questa tesi, è stata esplorata la fase di progettazione e di sviluppo del Mussels Grow Tracker (MGT), un dispositivo IoT progettato per monitorare in tempo reale la crescita dei mitili, migliorando così le tecniche di gestione e ottimizzazione della produzione nell'ambito dell'acquacoltura.

Il principale obiettivo del progetto consisteva nella creazione di un dispositivo capace di monitore la crescita dei mitili per poi inviare il resoconto in Blockchian attraverso la comunicazione LoRaWAN e con l'ausilio di TTN.

Il MGT offre un mezzo efficiente per tracciare in sicurezza la crescita dei mitili, adottando tecniche di ECDSA e Blockchian per assicurare la validità dei dati riportati, permettendo agli allevatori di ottimizzare i cicli di produzione e migliorare la qualità del prodotto finale per il cliente.

Durante lo sviluppo, sono state adottate tecnologie chiave e metodologie di progettazione che hanno portato alla realizzazione di un prototipo funzionale, capace di operare in ambiente di acquacoltura cercando di soddisfare i requisiti richiesti.

La fase di testing, condotta attraverso simulazioni, ha dimostrato l'affidabilità e l'efficacia del MGT nel raccogliere e trasmettere dati critici. I test di carico hanno confermato la scalabilità e la robustezza del sistema, validando l'approccio adottato nello sviluppo del dispositivo e nel suo integratione con la piattaforma The Things Network.

I risultati ottenuti dal Mussels Grow Tracker sono promettenti e aprono nuove prospettive per l'industria dell'acquacoltura, offrendo uno strumento potente per il monitoraggio e la gestione delle risorse.

L'architettura modulare e scalabile del sistema consente future estensioni e miglioramenti.

5.1 Sviluppi futuri

I possibili sviluppi futuri del Mussels Grow Tracker (MGT) possono aprire nuove frontiere nell'ottimizzazione e nella gestione sostenibile dell'acquacoltura, sfruttando le avanzate tecnologie IoT e l'analisi dei dati. Di seguito sono elencate alcune direzioni promettenti per l'evoluzione futura del progetto:

- **Integrazione con Sensori ulteriori sensori**: L'implementazione di nuovi sensori, come quelli per la qualità dell'acqua (pH, ossigeno disciolto, salinità), può fornire dati più dettagliati sull'ambiente di crescita delle cozze, consentendo interventi più mirati per ottimizzare le condizioni di allevamento.

- **Analisi Predittiva e Machine Learning**: l'applicazione di tecniche di machine learning sui dati raccolti può permettere di sviluppare modelli predittivi sulla crescita delle cozze migliorando le strategie di intervento e prevenzione.

- **Interoperabilità e Standardizzazione**: lavorare verso una maggiore standardizzazione dei formati di dati e delle interfacce di comunicazione può facilitare l'integrazione del MGT con altre piattaforme IoT e software di gestione dell'acquacoltura, promuovendo un ecosistema più connesso e inter-operabile.

- **Sviluppo di Dashboard e Interfacce Utente Avanzate**: la creazione di dashboard può migliorare l'esperienza utente, fornendo strumenti di analisi e visualizzazione dati più potenti per la gestione e il monitoraggio delle colture dei mitili.

- **Sostenibilità e Alimentazione Energetica**: esplorare soluzioni per l'alimentazione energetica sostenibile del dispositivo, come l'uso di pannelli solari, può aumentare l'autonomia del MGT e ridurre l'impatto ambientale.
- **Controllo Remoto**: aggiungere la possibilità di poter effettuare determinate operazioni e compiti anche da remoto senza la necessità di operare sul campo.
- **Estensione a Nuove Specie e Ambienti**: adattare e testare il MGT per il monitoraggio di altre specie di interesse acquacolture o per l'uso in diversi ambienti.

  Questi sviluppi futuri potrebbero non solo migliorare la produttività e la sostenibilità dell'acquacoltura dei mitili, ma anche fornire strumenti preziosi per la ricerca scientifica e la conservazione degli ecosistemi marini.

Ringraziamenti

Desidero ringraziare la mia famiglia, che ha dimostrato una pazienza e un sostegno incondizionati lungo tutto il percorso che mi ha portato a raggiungere questo traguardo. Non è stato un cammino breve né semplice, ma la vostra presenza costante, il vostro incoraggiamento e la vostra fiducia nelle mie capacità mi hanno permesso di superare ogni ostacolo e di non perdere mai di vista l'obiettivo finale.

Elenco delle figure

Elenco delle tabelle

Bibliografia

[1] Sito ufficiale internet4things. https://www.internet4things.it/iot-library/internet-of-things-gli-ambiti-applicativi-in-italia/.

[2] Sito immagine lora. https://www.mokolora.com/lorawan-iot-is-thriving-over-other-lpwan/.

[3] Sito tlera corp. https://www.tindie.com/stores/tleracorp/.

[4] Sito ublox. https://www.mouser.it/ProductDetail/u-blox/CAM-M8Q-0?qs=vEM7xhTegWgOVMjXYnonQA%3D%3D.

[5] Sito immagine sx1276. https://ai-thinker.eu/product/ai-thinker-sx1276-soc-chip-868mhz-ultra-long-distance-wireless-spread-spectrum-lora-module-ra-01h/.

[6] Sito descrizione sx1276. https://www.semtech.com/products/wireless-rf/lora-connect/sx1276#inventory.

[7] Sito descrizione bma400. https://www.mouser.it/ProductDetail/Bosch-Sensortec/BMA400?qs=f9yNj16SXrKBoguHUc32eQ%3D%3D.

[8] Sito descrizione bme280. https://www.mouser.it/ProductDetail/Bosch-Sensortec/BME280?qs=2OnyuXx6vpj2fK9HX7qb3g%3D%3D.

[9] Sito immagine dc-dc step-up. https://futuranet.it/prodotto/convertitore-dc-dc-step-up-con-uscita-5-35-v/.

[10] Sito immagine level shifter. https://probots.co.in/4-channel-logic-level-converter-module.html.

[11] Sito immagine hx711. https://futuranet.it/prodotto/scheda-elettronica-per-cella-di-carico-hx711/.

[12] Sito immagine interruttore mosfet. https://www.amazon.it/HALJIA-Interruttore-elettronico-Regolazione-Controllo/dp/B08BC5B5CW.

[13] Sito immagine mpu6050. https://images.app.goo.gl/PiXJDz9Ng2vbVYPC7.

[14] Sito immagine ads1115. https://www.amazon.it/HALJIA-Interruttore-elettronico-Regolazione-Controllo/dp/B08BC5B5CW.

[15] Sito immagine circuito ycz-516c. https://lh3.googleusercontent.com/proxy/MAZk-W77fvMB0MnfWCbEO0n5b3GNys32uyzCJS2bnbqHpI3XiqGQC-ZNk4ZIdqG0CfHyVyKshbTRFS_g_BcoMMULSt1JrvcS9rE7b3GAzWCHCDFgZGTmHa8tVfSGzKDrgbQimxlMy T5lYsEs4yYq5YXnjDrPY5WP9rohcLXZtxyAFd5j.

[16] Sito immagine sentrius rg1xx lorawan gateway. https://www.lairdconnect.com/iot-devices/lorawan-iot-devices/sentrius-rg1xx-lorawan-gateway-wi-fi-ethernet-optional-lte-us-only.

[17] Sito immagine logo ttn. https://assets.cloud.thethings.network/brand-assets/.

[18] Fonte ecdsa. https://academy.bit2me.com/it/que-es-ecdsa-curva-eliptica/.

[19] Sito immagine logo arduino. https://brandslogos.com/a/arduino-logo-1/.

[20] Repository github. https://github.com/nsilveri/musselsgrowtracker.

[21] Sito descrizione gnss. https://www.strumentitopografici.it/c/gnss/.

[22] Fonte thething network. https://www.thethingsnetwork.org/.

[23] Fonte sha-256. https://academy.bit2me.com/it/sha256-algoritmo-bitcoin/.

[24] Sito immagine logo lora alliance. https://lora-alliance.org/.