Politecnico di Torino

Politecnico di Torino

Ingegneria Informatica(Computer Engineering)

A.a. 2024/2025

Sessione di laurea Luglio 2025

Configurazione e Sviluppo di un Sistema Embedded Linux per Automotive con Connettivita' Wireless

Relatore: Gianpiero Cabodi

Candidato: Samuele Ramello

Sommario

Questa tesi si concentra sullo studio, la configurazione e lo sviluppo di applicazioni pratiche per testare le funzionalita' di una scheda custom basata su architettura Arm64. Il progetto e' realizzato presso l'azienda Monet Srl, lavorando su una piattaforma progettata per l'automotive e basata su un SoM iMX8MP di Variscite, integrato su una carrier board con diverse periferiche, tra cui un modem Quectel con connettivita' LTE e GPS.

L'approccio adottato segue una strategia dal basso verso l'alto: si parte dall'analisi dell'hardware, passando per la configurazione di un'immagine Linux personalizzata tramite Yocto, fino allo sviluppo di applicazioni che sfruttano le funzionalita' di connettivita', come Wi-Fi e GPS. Inoltre, viene implementato un servizio di telemetria sicura su LTE tramite il protocollo MQTT con crittografia SSL. Il progetto si conclude con lo sviluppo di un'applicazione dedicata alla visualizzazione dei dati del veicolo.

Indice

Elenco delle figure

Capitolo 1

Introduzione

1.1 Contesto e motivazione del progetto

Il progetto e' stato svolto presso la azienda Monet SRL la quale produce piattaforme hardware e software per sistemi elettronici embedded. Questa tesi e' stata svolta utilizzando una loro scheda dedicata al settore automotive di recente produzione.

1.2 Obiettivi della tesi

Gli obiettivi di questo lavoro sono: testare il funzionamento della scheda, interfacciarsi con le varie periferiche, sviluppare una piattaforma che dia risalto alle potenzialita' della scheda.

1.3 Struttura del documento

Il seguente documento è strutturato seguendo circa il flusso di lavoro utilizzato durante la ricerca e sviluppo dell' intero progetto. Nel capitolo [2] vengono analizzate e confrontate le principali soluzioni per lo sviluppo embedded nel settore automotive. Nei capitoli [3] e [4] vengono analizzate le potenzialita' e periferiche della scheda e successivamente gli strumenti di generazione di una immagine linux ad hoc personalizzata per l'hardware. I capitoli [5] e [6] riguardano l'interfaccia verso le periferiche e la rete e lo sviluppo di programmi. Il capitolo [7] segue lo sviluppo di una piattaforma che racchiuda le funzionalita' della scheda e mostri graficamente dati utili di un veicolo.

Capitolo 2

Studio dello Stato dell'Arte delle soluzioni Sviluppo Software Embedded per l'Automotive

2.1 Tecnologie e strumenti utilizzati nel settore

Il settore automotive ha subito negli ultimi anni una forte trasformazione grazie all'integrazione di tecnologie embedded sempre più avanzate. I moderni veicoli sono equipaggiati con dozzine di centraline elettroniche (ECU - Electronic Control Unit) dedicate al controllo di sottosistemi quali motore, freni, infotainment, climatizzazione, e sistemi di guida autonoma o assistita (ADAS). Queste centraline sono principalmente composte da microcontrollori e microprocessori e sono fortemente interconnesse. Le connessioni tra esse seguono molteplici standard per garantire una comunicazione stabile ed affidabile. Sono presenti sul mercato molteplici strumenti che aiutano gli sviluppatori a interfacciarsi con una cosi grande varieta' di tecnologie, quali ide specializzati, framework, middleware software per l'astrazione dell'hardware e tanto altro. L'insieme di strumenti utilizzati per sviluppare e distribuire i programmi sulle ECU caratterizza una specifica toolchain di programazzione. Tali strumenti si combinano con approcci rigorosi di ingegneria del software, che includono requisiti di sicurezza funzionale secondo la norma ISO 26262.

2.2 Confronto bare-metal, RTOS, Linux

Nello sviluppo di sistemi embedded per l'automotive, la scelta del paradigma di esecuzione rappresenta una decisione cruciale che influenza direttamente le prestazioni, la manutenibilità e l'affidabilità del sistema finale. Questa decisione deve considerare attentamente i vincoli di tempo reale, la potenza computazionale disponibile, la memoria e la complessità totale del sistema.

Il paradigma **bare-metal** prevede l'esecuzione del software direttamente sull'hardware, senza l'intermediazione di un sistema operativo. Questo approccio si rivela particolarmente adatto per sistemi semplici che richiedono risposte in tempo reale stringenti. Le prestazioni offerte sono massime, con un consumo di risorse minimo e tempi di avvio estremamente ridotti. Tuttavia, la gestione di più task diventa rapidamente complessa, la scalabilità risulta limitata e la manutenzione del codice può rivelarsi difficoltosa nel lungo periodo. Nel contesto automotive, il paradigma bare-metal trova applicazione in sistemi critici come il controllo del motore, la gestione dei sensori e degli attuatori, dove la velocità e la prevedibilità di risposta sono prioritarie. Lo sviluppo di questi programmi e' estremamente personalizzato sul caso specifico.

I **sistemi operativi real-time (RTOS)** rappresentano una soluzione intermedia, offrendo meccanismi di scheduling deterministici, gestione efficiente dei task, delle interruzioni e delle risorse condivise. Un RTOS garantisce una buona scalabilità e semplifica notevolmente la gestione del multitasking mantenendo al contempo un elevato livello di determinismo. Questi vantaggi comportano una maggiore complessità rispetto all'approccio bare-metal e un consumo di memoria superiore. Gli RTOS trovano applicazione ideale nei sistemi di infotainment, nella gestione delle batterie (BMS), nei gateway CAN e nei sistemi ADAS di basso livello, dove è necessario bilanciare requisiti di tempo reale con una maggiore complessità funzionale. Esempi diffusi in ambito automotive includono FreeRTOS, VxWorks, QNX e AUTOSAR OS. Questi facilitano la portabilita' e scalabilita' degli applicativi su differenti architetture e sistemi hardware.

Per i sistemi più complessi e meno critici dal punto di vista del tempo reale, l'Embedded Linux rappresenta una scelta sempre più popolare. Distribuzioni specializzate come Yocto, Buildroot e AGL (Automotive Grade Linux) offrono soluzioni ottimizzate per l'ambiente automotive. Linux embedded si distingue per l'ampia disponibilità di librerie, strumenti di sviluppo e il supporto di una vasta community. Il multitasking avanzato e la gestione sofisticata delle risorse sono ulteriori punti di forza. Tuttavia, i tempi di avvio risultano significativamente più elevati e questo approccio non è generalmente adatto per applicazioni con

requisiti di hard real-time. I sistemi di infotainment, i cluster digitali della strumentazione, i sistemi di navigazione e telematica rappresentano candidati ideali per l'implementazione basata su Linux embedded.

È importante sottolineare che la scelta tra questi approcci raramente si configura come esclusiva nell'architettura complessiva di un veicolo moderno. Nella maggior parte dei casi, si adotta una strategia ibrida: all'interno dello stesso veicolo coesistono ECU basate su paradigma bare-metal, altre che utilizzano RTOS e altre ancora implementate su piattaforme Linux. Questa diversificazione consente di ottimizzare ciascun sottosistema per il proprio dominio funzionale specifico, bilanciando requisiti di prestazioni, affidabilità e complessità in modo mirato.

2.3 Tecniche di cross compilazione

La cross compilazione rappresenta un elemento fondamentale nello sviluppo di sistemi automotive moderni. Questa necessità deriva dalla natura stessa dei sistemi elettronici presenti sui veicoli, che raramente utilizzano architetture standard come quelle dei computer desktop o laptop (x86-64). I sistemi automobilistici prediligono invece architetture con instruction set meno complessi, appositamente progettati per soddisfare requisiti specifici come il controllo in tempo reale e la minimizzazione del consumo energetico e della dissipazione di calore.

In questo contesto, la cross compilazione consiste nel processo di compilazione del codice sorgente su una macchina host, tipicamente un PC con architettura x86, per generare eseguibili destinati a una macchina target con architettura differente, come ARM, PowerPC o RISC-V. Questo approccio si rende necessario poiché i dispositivi embedded nei veicoli generalmente non dispongono delle risorse computazionali o degli strumenti software necessari per eseguire la compilazione direttamente sull'hardware di destinazione.

Nel panorama dello sviluppo automotive, diversi strumenti si sono affermati per facilitare il processo di cross compilazione. Il GCC cross-compiler rappresenta una soluzione versatile che supporta numerose architetture target e può essere configurato tramite specifici file di toolchain. CMake è diventato uno standard de facto per progetti C/C++, offrendo un solido supporto per configurazioni cross-platform. Per sistemi Linux embedded, il Yocto Project fornisce un framework completo che include toolchain di cross compilazione, consentendo la creazione di sistemi personalizzati. Buildroot rappresenta un'alternativa più leggera a Yocto, particolarmente apprezzata per la sua capacità di costruire root filesystem e toolchain personalizzate in modo relativamente semplice.

Il processo di cross compilazione segue generalmente una sequenza ben definita.

Inizialmente, si procede con la configurazione della toolchain appropriata, installando il cross-compiler specifico per l'architettura target. Successivamente, si configura l'ambiente di build definendo variabili essenziali come CC, CXX, LD e AR. Una volta completata questa fase preparatoria, si procede alla compilazione del codice utilizzando make, CMake o altri sistemi di build. Il codice compilato viene quindi trasferito sul dispositivo target mediante diverse modalità: debug hardware (come JTAG), schede SD, connessioni di rete (TFTP/SSH) o interfacce USB. Infine, si eseguono operazioni di debugging e test utilizzando strumenti come GDB server, OpenOCD o soluzioni specifiche fornite dai produttori hardware.

Nonostante la sua importanza, la cross compilazione presenta diverse sfide che gli sviluppatori devono affrontare. Le incompatibilità tra le librerie presenti sull'host e quelle disponibili sul target possono causare problemi significativi durante la fase di linking. Il debugging remoto risulta intrinsecamente più complesso rispetto al debugging nativo, richiedendo configurazioni specifiche e strumenti dedicati. Inoltre, la portabilità tra diverse toolchain e versioni del compilatore può introdurre problematiche inattese che richiedono attenzione particolare. Nel caso di sviluppo per embedded linux, parte di questi problemi viene risolta tramite la configurazione e utilizzo di Yocto, come vedremo nel capitolo [4].

La gestione efficace del processo di cross compilazione riveste un'importanza cruciale per garantire una pipeline di sviluppo efficiente e replicabile nei progetti embedded automotive. Un approccio ben strutturato alla cross compilazione consente di ottimizzare i tempi di sviluppo, facilitare l'implementazione di aggiornamenti software e migliorare la qualità complessiva del prodotto finale, contribuendo significativamente al successo dei progetti automotive moderni.

2.3.1 Come (cross)compilare una libreria

Se la compilazione di un eseguibile senza dipendenze e' relativamente semplice, trattandosi di configurare solamente il compilatore e qualche flags, per una libreria diventa tutto piu' complesso. Ad esempio per compilare una libreria del protocollo mqtt, nello specifico la versione sviluppata dal team Eclipse paho-mqtt-c ci sono tre metodi:

- **Compilazione nativa**
  E' sufficiente scaricare localmente tramite package manager le librerie tramite:

```
$ apt install libssl3 libssl-dev libpaho-mqtt1.3 \
libpaho-mqtt-dev
```

Dopo di che' sara' sufficiente compilare con:

```
# Linker Flags
LDFLAGS = -lpaho-mqtt3cs -lssl -lcrypto
```

- **Cross compilazione da sorgente con CMake**
  La maggior parte dei progetti dispongono di una guida per la compilazione/-cross compilazione da codice sorgente con CMake.

```
#.../paho.mqtt.c/cmake/toolchain.linux-arm64.cmake
# path to compiler and utilities
# specify the cross compiler
SET(CMAKE_C_COMPILER aarch64-linux-gnu-gcc)

# Name of the target platform
SET(CMAKE_SYSTEM_NAME Linux)
SET(CMAKE_SYSTEM_PROCESSOR arm)

# Version of the system
SET(CMAKE_SYSTEM_VERSION 1)
------
# build with cmake, working but \
# install on default dir /usr/local
cmake -GNinja \
    -DPAHO_BUILD_DOCUMENTATION=TRUE \
    -DPAHO_WITH_UNIX_SOCKETS=TRUE \
    -DPAHO_BUILD_STATIC=TRUE \
    -CMAKE_TOOLCHAIN_FILE= \
    .../paho.mqtt.c/cmake/toolchain.linux-arm64.cmake \
    .../paho.mqtt.c

cmake --build . --target install
```

Per compilare e' poi necessario includere la directory contenente le librerie. Questa procedura puo' risultare complessa e dispendiosa, specialmente in casi di dipendenze ricorsive. Prendendo sempre paho-mqtt-c, per compilare la versione con protocollo SSL/TLS e' necessario avere precedentemente le librerie openssl-dev gia' compilate per la medesima architettura.

Ad esempio:

```
### Compile first the openssl/libssl-dev library ###
BUILD FOR ARM64
-configure-
./Configure
--prefix=.../folder
--openssldir=.../folder/ssl
--cross-compile-prefix=aarch64-linux-gnu-

-build-
make CC="aarch64-linux-gnu-gcc" CFLAGS=" \
-I.../libssl-dev/include" LDFLAGS="- \
L.../libssl-dev/lib" -j2
make
make install # --> should install the compiled \
# files into \ .../folder set in --prefix

### Then the paho-mqtt-c ###
file .../paho.mqtt.c/cmake/toolchain.linux-arm64.cmake
# path to compiler and utilities
# specify the cross compiler
SET(CMAKE_C_COMPILER aarch64-linux-gnu-gcc)

# Name of the target platform
SET(CMAKE_SYSTEM_NAME Linux)
SET(CMAKE_SYSTEM_PROCESSOR arm)

# Version of the system
SET(CMAKE_SYSTEM_VERSION 1)

------
export OPENSSL_ROOT_DIR=".../libssl-dev"
------
# build with cmake, working but install \
# on default dir /usr/local
cmake -GNinja -DPAHO_WITH_SSL=TRUE \
    -DPAHO_BUILD_DOCUMENTATION=TRUE \
    -DPAHO_WITH_UNIX_SOCKETS=TRUE \
    -DPAHO_BUILD_STATIC=TRUE \
    -DOPENSSL_LIB_SEARCH_PATH=.../libssl-dev/lib \
```

```
        -DOPENSSL_INC_SEARCH_PATH=.../libssl-dev/include \
        -DCMAKE_TOOLCHAIN_FILE= \
        .../paho.mqtt.c/cmake/toolchain.linux-arm64.cmake \
        .../paho.mqtt.c

    cmake --build . --target install
```

Un altro svantaggio infine e' l'obbligo di compilazione statica dell'eseguibile, rendendo impossibile la condivisione delle librerie tra piu' appicativi.

- **Cross compilazione con Yocto**
  Yocto risolve la maggior parte di queste procedure manuali, offrendo un sistema di compilazione basato su script Make/CMake auto generati in base alla configurazione impostata. Ad esempio per cross compilare la libreria paho-mqtt-c con support SSL/TLS, una volta impostate le librerie nell'immagine, e' sufficiente generare l'sdk e configurare le variabili d'ambiente con un singolo script. Dopo di che' la compilazione avviene come quella nativa, con compilatore e sysroot automaticamente impostate. Questo approcio permette anche la compilazione dinamica.

Capitolo 3

Analisi dell'Hardware

3.1 Panoramica sulla piattaforma automotive

Il sistema in esame è basato su una **carrier board** progettata internamente dall'azienda Monet, con lo scopo di ospitare un **System-on-Module (SoM)**. Questa architettura modulare consente una notevole flessibilità nello sviluppo di applicazioni embedded in ambito automotive, grazie alla possibilità di integrare differenti periferiche e interfacce di comunicazione. La scheda è concepita per garantire l'interconnessione con altre ECU presenti nel veicolo, supportando protocolli e standard comuni nell'industria automobilistica.

3.2 Descrizione del SoM di Variscite basato su i.MX 8M Plus di NXP

Il modulo installato sulla carrier board è un **DART-MX8M-PLUS** [1], prodotto da **Variscite**, costruito attorno al potente processore **i.MX 8M Plus** di **NXP Semiconductors**. Questo SoC (System on Chip) rappresenta una soluzione di fascia medio-alta per sistemi embedded, particolarmente indicata per applicazioni edge AI, multimedia, industriali e automotive.

Il **NXP i.MX 8M Plus** è dotato di:

- **Quad-core Arm Cortex-A53** a 1.8 GHz per l'esecuzione di sistemi operativi complessi come Linux e Android.
- **Microcontrollore integrato Cortex-M7** a 800 MHz per la gestione in tempo reale e il controllo di basso livello, utile per attività critiche legate alla

sicurezza o alla comunicazione con periferiche.

- Un **Neural Processing Unit (NPU)** dedicato, con potenza di calcolo fino a 2.3 TOPS (Tera Operations Per Second), ideale per eseguire inferenze AI direttamente sul dispositivo, senza necessità di connettività cloud.
- **Memoria LPDDR4 da 4 GB** ad alta larghezza di banda e memoria eMMC per lo storage locale del sistema operativo e dei dati persistenti.
- **Slot microSD** per espansione della memoria esterna.
- Connettività avanzata integrata: supporto a **Wi-Fi 6 (802.11ax), Bluetooth 5.4**, compresa la modalità BLE (Bluetooth Low Energy), utile per dispositivi a basso consumo energetico.
- Ampie possibilità di interfacciamento con l'esterno tramite **I2C, SPI, UART, GPIO, CAN, PCIe e USB**.

Queste caratteristiche rendono il VAR-SOM-MX8M-PLUS una soluzione estremamente versatile, adatta a gestire carichi computazionali anche complessi, come l'elaborazione video, il riconoscimento di oggetti tramite reti neurali o l'integrazione con sensori avanzati.

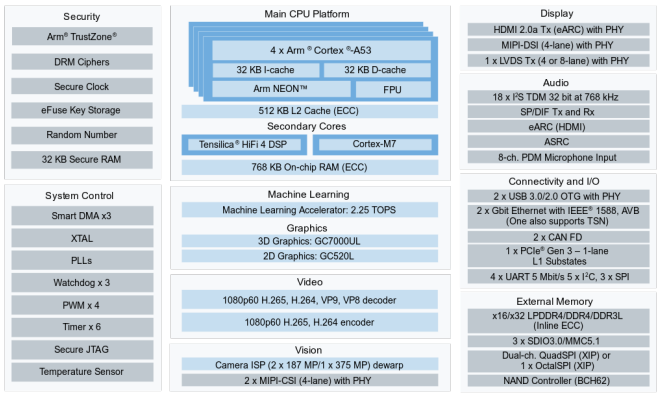

Figura 3.1: Diagramma a blocchi del SoC i.MX 8M Plus di NXP

3.3 Analisi della carrier board e delle periferiche integrate

La carrier board progettata da Monet rappresenta l'interfaccia principale tra il SoM e l'ambiente esterno. La scheda è stata realizzata con particolare attenzione alla

robustezza e all'adattabilità in ambito veicolare, e supporta numerose funzionalità fondamentali:

- **Doppia interfaccia Ethernet automotive**, per connettività ad alta velocità tra ECU.
- **Due bus CAN (Controller Area Network)**, utilizzati comunemente in ambito automotive per la comunicazione tra centraline.
- **Interfaccia SPI** verso un microcontrollore esterno a 32 bit, prodotto da **STMicroelectronics**, che svolge funzioni di supporto o controllo ausiliario.
- **Connettore mini-PCIe** con interfaccia USB, utilizzato per collegare un modem LTE/GNSS.
- **Due porte UART** dedicate al debug del sistema, utili per attività di sviluppo e diagnostica.
- Gestione dell'alimentazione tramite ingresso a **12-15V**, con regolazione interna per garantire stabilità anche in condizioni di tensione fluttuante tipiche di un ambiente automotive.
- **LED di stato multipli**, utilizzati per monitorare visivamente lo stato del sistema, l'alimentazione e la connettività.

3.4 Studio del modem Quectel EG25-G: Connettività LTE e GPS

Il sistema utilizza un **modulo di comunicazione Quectel EG25-G** [2], una soluzione **LTE Cat 4** progettata per applicazioni IoT e automotive che richiedono connettività mobile e geolocalizzazione affidabile.

Il **Quectel EG25-G** è un modulo globale che supporta:

- **LTE Cat 4** con velocità di download fino a **150 Mbps** e upload fino a **50 Mbps**.
- Compatibilità con reti **2G (GSM/GPRS/EDGE)** e **3G (UMTS/H-SPA+)**, garantendo la retrocompatibilità in aree dove la rete LTE non è disponibile.
- **Ricevitore GNSS multi-costellazione** integrato (GPS, GLONASS, BeiDou, Galileo), con supporto AGPS per un fix rapido e preciso della posizione.

- Interfacciamento tramite **mini-PCIe con connessione USB 2.0**, facilmente integrabile nella carrier board.
- Supporto a **comandi AT standard 3GPP**, facilitando lo sviluppo software su Linux/Android.
- **Certificazioni globali** (CE, FCC, RoHS, PTCRB, GCF) che lo rendono adatto per implementazioni internazionali.

Grazie a queste caratteristiche, il Quectel EG25-G consente alla piattaforma di essere sempre connessa alla rete, di ottenere informazioni di geolocalizzazione e di trasmettere dati in tempo reale, funzionalità fondamentali per sistemi telematici o piattaforme di diagnostica remota.

Capitolo 4

Configurazione dell'Immagine Linux Personalizzata

4.1 Introduzione a Yocto Project

Yocto Project [3] è un framework open source che fornisce modelli, strumenti e metodi per creare sistemi Linux personalizzati per dispositivi embedded, indipendentemente dall'architettura hardware. Fondato nel 2010 come progetto collaborativo della Linux Foundation, si è affermato come uno standard de facto nell'industria embedded.

Il progetto è basato sul sistema di compilazione OpenEmbedded, che utilizza il concetto di *ricette* (recipes) per descrivere come costruire e configurare pacchetti software. Queste ricette specificano dettagli come le dipendenze, le fonti del codice, le opzioni di compilazione e le configurazioni di installazione.

I componenti principali di Yocto Project includono:

- **BitBake**: Il motore di esecuzione delle attività che legge le ricette e le esegue per creare i componenti del sistema.
- **OpenEmbedded-Core**: Contiene i metadati fondamentali necessari per costruire un sistema Linux base.
- **Layer di metadati**: Collezioni organizzate di ricette e configurazioni che possono essere aggiunte al sistema di build.

- **Poky**: La distribuzione di riferimento di Yocto Project.

4.2 Vantaggi Yocto Project

Yocto Project offre numerosi vantaggi rispetto ad altre soluzioni come Buildroot o immagini standard:

- **Stratificazione del software**
  Yocto adotta un approccio modulare basato su layer, che permette di separare chiaramente i diversi livelli di personalizzazione (core, BSP, applicazioni specifiche). I layer possono essere riutilizzati tra progetti diversi, riducendo la duplicazione degli sforzi e migliorando la manutenibilità. Le personalizzazioni possono essere isolate in layer specifici, rendendo più semplice la gestione e l'aggiornamento del sistema.

- **Definizione precisa delle librerie e versioni**
  Yocto garantisce che ogni build produca risultati identici, specificando esattamente le versioni di ogni pacchetto e delle sue dipendenze. Gli sviluppatori possono definire con precisione quali versioni di librerie e componenti utilizzare, evitando problemi di compatibilità. Il sistema di build risolve automaticamente le dipendenze tra pacchetti, assicurando che tutte le librerie necessarie siano incluse con le versioni corrette.

- **Standardizzazione della toolchain di cross-compilazione**
  La toolchain generata include già un sysroot correttamente configurato, che contiene header e librerie target necessari per la cross-compilazione. Le librerie nella toolchain seguono esattamente la versione dell'immagine buildata, garantendo che il codice sviluppato funzioni allo stesso modo nell'ambiente target. È possibile generare SDK specifici per il progetto che includono esattamente le stesse versioni delle librerie presenti nell'immagine finale.

- **Vantaggi aggiuntivi**
  Altri vantaggi includono la tracciabilità completa di ogni componente, il supporto integrato per meccanismi di aggiornamento software (OTA), la configurazione flessibile dell'immagine e il supporto da parte di aziende leader nel settore e una comunità attiva di sviluppatori.

4.3 Creazione di un'immagine Linux customizzata

La creazione di un'immagine Linux personalizzata con Yocto Project richiede una comprensione dei concetti fondamentali e una corretta configurazione dell'ambiente di sviluppo. Nel caso specifico, utilizziamo la versione Mickledore di Yocto con kernel 6.1.36 fornita da Variscite per il SoM (System on Module) revisione 1.3, configurata con init/SysVinit [4].

4.3.1 Preparazione dell'ambiente di sviluppo

Il primo passo consiste nel download del BSP (Board Support Package) che contiene tutti i metadati specifici per l'hardware Variscite imx8mp-var-dart:

```
$ sudo dnf install repo
$ mkdir docker-yocto-build
$ cd docker-yocto-build
$ repo init -u
    https://github.com/varigit/variscite-bsp-platform.git \
    -b refs/tags/mx8mp-yocto-mickledore-6.1.36_2.1.0-v1.3 \
    -m default.xml
$ repo sync -j4
```

Per garantire un ambiente di build consistente, è possibile utilizzare Docker:

```
$ cd ~/var-host-docker-containers
$ ./run.sh -p -u 20.04 -w ~/docker-yocto-build
```

4.3.2 Configurazione e avvio del build

Una volta preparato l'ambiente, è necessario configurarlo per il build:

```
$ cd ~/var-fsl-yocto
$ MACHINE=imx8mp-var-dart DISTRO=fsl-imx-wayland
    . var-setup-release.sh build_xwayland
```

Per sessioni successive, è sufficiente eseguire:

```
$ cd ~/var-fsl-yocto
$ source setup-environment build_xwayland
```

La personalizzazione dell'immagine può avvenire a diversi livelli, partendo da immagini predefinite come `core-image-minimal`, creando layer personalizzati o modificando configurazioni esistenti.

Per avviare il processo di build di un'immagine minimale:

```
$ bitbake core-image-minimal
```

4.3.3 Creazione e utilizzo della toolchain SDK

Una volta completata la build dell'immagine, è possibile generare un SDK personalizzato [5]. Queso facilita di molto il lavoro dello sviluppatore configurando un cross compilatore con i parametri corretti per la scheda target. Sfrutta le caratteristiche definite in precedenza durante la creazione dell'immagine per impostare una directory (sysroot), che si sostituisce a quella di default del sistema host, dove sono presenti le medesime librerie descritte nei layer yocto precompilate per l'architettura target. Questo rimuove la necessita' di configurare manualmente strumenti come CMake.
Nella pratica:

```
$ bitbake -c populate_sdk core-image-minimal
```

Questo comando crea uno script di installazione per l'SDK nella directory `tmp/deploy/sdk/`.

Per installare l'SDK:

```
$ tmp/deploy/sdk/fsl-imx-xwayland-glibc-x86_64-core-image- \
minimal-armv8a-imx8mp-var-dart-toolchain-6.1-mickledore.sh
```

Prima di utilizzare l'SDK, è necessario configurare l'ambiente:

```
$ source /opt/fsl-imx-xwayland/6.1-mickledore/ \
environment-setup-armv8a-poky-linux
```

A questo punto la variabile d'ambiente CC come altre punteranno a un percorso specifico del cross compilatore.

Esempio di cross compilatore configurato con Yocto:

```
$CC -v
COLLECT_GCC=aarch64-poky-linux-gcc
.....
Target: aarch64-poky-linux
Configured with: work-shared/gcc-12.2.0-r0/gcc-12.2.0/configure \
--build=x86_64-linux --host=x86_64-pokysdk-linux \
--target=aarch64-poky-linux
.....
--with-build-sysroot=/workdir/build_xwayland/tmp/work/\
x86_64-nativesdk-pokysdk-linux/gcc-cross-canadian-aarch64/\
12.2.0-r0/recipe-sysroot
Thread model: posix
gcc version 12.2.0 (GCC)
```

4.4 Device tree e integrazione dei driver per le periferiche

Il Device Tree è una struttura dati ad albero utilizzata per descrivere in modo standardizzato e indipendente dal kernel l'hardware di un sistema embedded. Questa descrizione include dispositivi, controller, bus e altre componenti hardware, organizzate gerarchicamente in nodi e proprietà. E' quindi fondamentale per la corretta integrazione dei driver delle periferiche hardware e tra gli altri vantaggi facilita la portabilita' tra immagini differenti e permette di separare la descrizione dell'hardware dal codice del kernel, evitando di dover ricompilare il kernel per ogni variazione hardware.

4.4.1 Panoramica del Device Tree

Il Device Tree è scritto in un linguaggio testuale (Device Tree Source, file .dts) che viene compilato in un formato binario (Device Tree Blob, file .dtb). Il bootloader (es. U-Boot) carica il DTB in memoria e passa il suo indirizzo al kernel durante l'avvio. Il kernel legge e si autoconfigura dinamicamente in base all'hardware.

4.4.2 Integrazione dei driver per le periferiche

L'integrazione dei driver nel sistema Yocto richiede diverse fasi:

1. **Selezione dei driver nel kernel**:

```
$ bitbake virtual/kernel -c menuconfig
```

2. **Creazione di un layer per driver personalizzati**:

```
$ cd ~/var-fsl-yocto
$ source setup-environment build_xwayland
$ bitbake-layers create-layer ../meta-custom-drivers
$ bitbake-layers add-layer ../meta-custom-drivers
```

3. **Definizione delle ricette per i driver**:

```
$ mkdir -p ../meta-custom-drivers/recipes-kernel/mydriver
$ cp mydriver.c ../meta-custom-drivers/recipes-kernel/ \
mydriver/
$ touch ../meta-custom-drivers/ \
recipes-kernel/mydriver/mydriver_1.0.bb
```

4. **Inclusione dei driver nell'immagine finale**:

```
IMAGE_INSTALL:append = " mydriver"
```

(Aggiungere questa riga nel file 'local.conf' o nella ricetta dell'immagine)

Dopo il deployment dell'immagine sul dispositivo target, è possibile verificare l'integrazione corretta dei driver con comandi come `dmesg` e `lsmod`.

E' anche possibile inserire direttamente dei sorgenti personalizzati da compilare insieme all'immagine per ottenere dei driver custom.

La combinazione di una corretta configurazione del device tree e l'integrazione appropriata dei driver è fondamentale per garantire il funzionamento ottimale dell'hardware nel contesto di un'immagine Linux personalizzata creata con Yocto Project.

4.5 Servizi all'avvio e configurazioni aggiuntive

Yocto permette di inserire direttamente nell'immagine file di configurazione in directory specifiche e script/servizi da eseguire all'avvio del sistema.

4.5.1 Script e configurazioni

Nel caso della scheda in questione, viene utilizzato uno script per impostare correttamente le porte can in fase di boot.

```
#!/bin/bash
$ ip link set can0 type can bitrate 500000
$ ip link set can0 up
$ ip link set can1 type can bitrate 500000
$ ip link set can1 up
```

Inoltre vengono inseriti file di configurazione per il servizio NetworkManager. Questo permette di impostare quali interfacce di rete, tra wifi, module lte e altro, debbano essere gestite automaticamente e quali no. Un esempio:

```
/etc/NetworkManager/conf.d/99-unmanaged-devices.conf
[keyfile]
unmanaged-devices=interface-name:uap0; interface-name:usb0
```

Approfondimento nella sezione [5.4]

4.5.2 Lancio dell'applicativo

E' possibile aggiungere dei file sorgente e header all'immagine in modo da compilare tutto insieme. L'immagine avra' quindi gia' incluso un eseguibile pronto al lancio. Per eseguire l'applicazione principale all'avvio e' quindi necessario aggiungere uno script o servizio che verra' lanciato all'avvio della scheda, in seguito alla completa inizializzazione del sistema operativo. La versione dell'immagine mickledore utilizza un gestore di servizi "vecchio stampo". Essenzialmente i servizi o demoni sono script bash che vengono richiamati in ordine basandosi sul nome e il percorso del file. Questo rende piu' semplice la scrittura e la gestione di piccoli script, ma risulta meno sofisticato di un gestore moderno quale systemd.

Esempio:

```
#!/bin/sh
case "$1" in
  start)
    /etc/customApp/main &
    echo $! > /var/run/customapp.pid
    ;;
  stop)
    kill $(cat /var/run/customapp.pid)
    rm /var/run/customapp.pid
    ;;
  restart)
    $0 stop
    $0 start
    ;;
  *)
    echo "Uso: $0 {start|stop|restart}"
    exit 1
esac
```

Salvato in /etc/init.d/customapp puo' essere configurato per essere lanciato all'avvio e accessibile tramite:

```
$ service start customapp
```

Capitolo 5

Configurazione periferiche tramite utilities e comandi Linux

5.1 Interfaccia con il modem e comandi AT

Il modem Quectel si interfaccia con il SoC tramite usb. espone più porte seriali virtuali tramite il driver *option* del kernel. Linux riconosce automaticamente due nuovi dispositivi, una di configurazione /dev/ttyUSB2 e la porta dati /dev/ttyUSB1. Comunicando tramite seriale/uart, e' possibile configurare il modem tramite comandi AT. La sintassi e' la seguente e puo' essere di 4 tipi:
AT + COMMAND SUFFIX DATA SUFFIX

- read ?
- set =
- execute None
- test =?

Una volta configurato a piacere, dalla porta in lettura verrano inviati i dati richiesti. Il modem mantiene lo stato delle impostazioni al ravvio.

5.2 Configurazione del modulo GPS

Il modem dispone di una serie di personalizzazioni e funzionalita' avanzate per il gps. Per avviare il gps:

```
$ AT+QGPS = 4 -> turn on gps 1 Stand-alone 2 MS-based
3 MS-assisted 4 Speed-optimal
$ AT+QGPSCFG="autogps",1 // enable GNSS to run automatically
```

I dati gps vengono inviati tramite delle stringhe standard definite da NMEA (National Marine Electronics Association). Per abilitare l'output su /dev/ttyUSB1:

```
$ AT+QGPSCFG="nmeasrc",1 // enable nmea sentences
```

Il modem Quectel EG25-G permette la connessione in contemporanea a piu' sistemi di geolocalizzazione, detta constellazione. Supporta tutti e 4 i principali: GPS(americano/globale), GLONASS, GALILEO(europeo), BEIDOU(cinese). Per abilitarli contemporaneamente:

```
$ AT+QGPSCFG="gnssconfig" // configure gnss
constellations GLONASS ON/BDS ON/Galileo ON
```

Sono disponibili ulteriori configurazioni, quali la configurazione granulare del tipo di dati da ricevere, differenti in base alle stringhe NMEA. Ad esempio:

- RMC Recommended minimum specific GPS/Transit data
- GGA Global Positioning System Fix Data
- GSV Satellites in view

Un esempio di output:

```
$GPVTG,347.5,T,347.9,M,0.0,N,0.0,K,A*2F
$GPRMC,114447.00,A,4504.161523,N,00740.133656,E,0.0 \
,347.5,211124,0.4,W,A,V*5A
$GPGSA,A,3,05,13,14,15,22,24,30,,,,,,1.6,1.4,0.9,1*27
```

E' intoltre possibile impostare la frequenza di aggiornamento dei dati 1-10 Hz.

5.3 Configurazione del modulo LTE

La gestione della connessione 3G/4G e' piu' complessa rispetto al GPS. La velocita' e quantita' di dati scambiati sono spesso notevolmente maggiori, e con una minore latenza richiesta. Per questospesso non e' sufficiente una porta seriale, ma e' necessario implementare un protocollo di comunicazione piu' avanzato.

Esistono 3 modalita' di interfacciamento:

- **Point to Point Protocol**
  La soluzione piu' semplice e di vasta compatibilta' e la modalita' PPP(Point to Point Protocol), tramite il quale e' possibile stabilire una connessione IP su una linea seriale. Utile quando la velocita' di trasferimento non e' una priorita'.

- **Ethernet Control Model**
  Presenta il modem come un'interfaccia di rete ehternet-usb. Per attivare questa modalità, inviare il comando AT:

  ```
  $ AT+QCFG="usbnet",1
  ```

  Dopo il ravvio e' sufficiente utilizzare gli strumenti standard di rete linux. La velocita' di trasferimento dei dati e' quella di un USB.

- **Qualcomm MSM Interface**
  Utilizza il driver qmi_wwan e l'interfaccia wwanX. Richiede l'installazione di strumenti aggiuntivi come libqmi e ModemManager per la gestione della connessione. Permette una configurazione nei dettagli delle impostazioni di connessione cellulare.

In questo caso e' stata scelta la modalita' ECM(Ethernet Control Model)

```
$ AT+CGDCONT=1,"IPV4V6","mobile.vodafone.it" //Set apn
$ AT+QCFG="usbnet",1 //Set modem to ECM, \
config to usb-ethernet interface
$ AT+CGDCONT=1,"IP","UNINET" //Define PDP context.
$ AT+CGACT=1,1 //Activated PDP.
$ AT+CFUN=1,1  //activate full functionality, reboot
```

L'interfaccia e' poi gestita in automatico da NetworkManager.

5.4 Gestione Wi-Fi e access point mode tramite Linux

Buona parte dei comandi che seguono sono stati presi dalla guida di Variscite [6]. Linux dispone di una serie di strumenti per la gestione del Wi-Fi, quali:

- **iw** per la gestione dei device wireless
- **ip** per la configurazione di interfacce di rete
- **dnsmasq**, un semplice server DHCP/DNS
- **hostapd**, servizio software per la creazione di access point
- **wpa_supplicant** per connettersi a access point WPA
- **NetworkManager** per gestire i sopra citati automaticamente tramite un unico tool(nmcli)

5.4.1 Station mode

Nell'immagine della scheda e' stato inserito un file di configurazione per ottenere da subito l'accesso alla rete locale.

```
$ /etc/NetworkManager/CustomAP.nmconnection
[connection]
id=CustomAP
uuid=a281ba03-c2fa-41b8-b9ca-6bb4839e7ec7
type=wifi
interface-name=wlp3s0
timestamp=1716829704

[wifi]
mode=infrastructure
ssid=CustomAP

[wifi-security]
key-mgmt=wpa-psk
psk=CustomPasswd

[ipv4]
```

```
dns=1.1.1.1;1.0.0.1;
ignore-auto-dns=true
method=auto

[proxy]
```

Per connettersi ad una nuova rete non nota a priori e' possibile sfruttare il tool **nmcli**

```
# connect to ap
$ nmcli dev wifi connect "ssid"|XX:XX:XX:XX:XX:XX \
password "mypassword" ifname INTERFACE
# scan for new ap
$ nmcli dev wifi rescan
```

E' possibile impostare i DNS direttamente tramite nmcli, anche per connessione specifica.

```
/etc/NetworkManager/conf.d/file.conf
[main]
dns=none
then set nameserver=8.8.8.8 in /etc/resolv.conf
or
/etc/NetworkManager/conf.d/file.conf
[main]
dns=dnsmasq --> should launch a dnsmasq \
internally managed by NetworkManager
```

In alternativa utilizzando **dnsmasq** si settano i DNS comuni per tutte le connessioni/interfacce di rete.

```
/etc/dnsmasq.d/file.conf --> set: server=8.8.8.8\n server=8.8.4.4
```

5.4.2 Access Point

Il dispositivo wifi puo' servire a sua volta come access point. La configurazione piu' semplice e automatica avviene tramite NetworkManager. Tuttavia per mantenere contemporaneamente entrambe le funzionalita', station mode e access point, sono necessari alcuni passaggi aggiuntivi. E' necessario quindi andare a definire manualmente i singoli strumenti di rete.

Per prima cosa occorre creare una interfaccia di rete virtuale, a cui viene assegnato un indirizzo ip di base

```
$ iw dev wlan0 interface add uap0 type __ap
$ ip addr add 192.168.2.1/24 dev uap0
```

Dnsmasq viene usato come server DHCP

```
# /etc/dnsmasq.d/dhcp_AP.conf
interface=lo,uap0,usb0       # Use the interface uap0 for DHCP
no-dhcp-interface=eth0      # Disable DHCP on eth0
no-dhcp-interface=wlan0     # Disable DHCP on wlan0
#no-dhcp-interface=usb0      # Disable DHCP on usb0

# Configure ipv4 address range
# Set the DHCP range and lease time
dhcp-range=uap0,192.168.2.10,192.168.2.20,12h

# Default gateway
dhcp-option=uap0,3,192.168.2.1

# Enable Router Advertisement (RA)
enable-ra
```

In seguito si sfrutta hostapd per la creazione del access point vero e proprio

```
# /etc/hostapd.conf
interface=uap0
driver=nl80211
ssid=TELMA_AP_2G
hw_mode=g
channel=6
wpa=2
wpa_passphrase=11223344
auth_algs=1
rsn_pairwise=CCMP
```

Infine, per accendere l' access point, occore lanciare l'eseguibile binario o avviare il servizion tramite *service* or *systemctl*

```
$ hostapd /etc/hostapd.conf
$ service hostapd start
$ systemctl enable|start hostapd
```

5.5 Comunicare con bus CAN con SocketCAN

Il protocollo **CAN** è ampiamente utilizzato nei sistemi embedded per la comunicazione tra dispositivi a bassa latenza. In Linux, l'interfacciamento al bus CAN è gestito tramite il sottosistema **SocketCAN** [7], che espone le interfacce CAN come socket di tipo `AF_CAN`.

5.5.1 Moduli del kernel

Per abilitare il supporto CAN, è necessario caricare i seguenti moduli:

```
$ modprobe can
$ modprobe can-raw
$ modprobe can-gw
$ modprobe vcan  # per interfacce virtuali
```

5.5.2 Configurazione interfacce CAN

Interfaccia virtuale (VCAN): utile per test e simulazione senza hardware.

```
$ ip link add dev vcan0 type vcan
$ ip link set up vcan0
```

È possibile creare più interfacce virtuali, ad esempio `vcan1`, per simulare comunicazioni multiple.

Interfaccia fisica: nel caso di dispositivi CAN reali, si configura specificando il bit rate.

```
$ ip link set can0 up type can bitrate 500000
```

Per test in assenza di rete, è possibile abilitare il loopback:

```
$ ip link set can0 type can loopback on
```

5.5.3 Gateway tra interfacce CAN

Il pacchetto `can-utils` include lo strumento `cangw` per la definizione di gateway CAN-to-CAN:

```
$ cangw -A -s vcan0 -d vcan1 -e
$ cangw -A -s vcan1 -d vcan0 -e
```

5.5.4 Invio e ricezione dati

`can-utils` fornisce strumenti per testare la comunicazione direttamente da linea di comando:

- Invio di un messaggio CAN:

  ```
  $ cansend vcan0 123#DEADBEEF
  ```

- Ascolto passivo su un'interfaccia:

  ```
  $ candump vcan0
  ```

Questi strumenti consentono il debug e la validazione delle comunicazioni CAN in ambienti embedded Linux, anche in fase di sviluppo senza accesso a hardware reale.

5.5.5 Interfaccia con codice C

Grazie all'astrazione del bus come socket e' possibile interagire con il protocollo CAN nel linguaggio c sfruttando le API dei socket standard di Linux.

Occorre importare le librerie

```
#include <sys/socket.h>
#include <linux/can.h>
```

La libreria *can.h* definisce il tipo:

```
struct can_frame {
    canid_t can_id;
    __u8 len8_dlc;
    __u8 data[CAN_MAX_DLEN];
};
```

Dopo aver aperto e connesso il socket e' sufficiente utilizzare le chiamate di sistema read/recv e write per leggere e scrivere messaggi CAN.

```
// Socket
struct sockaddr_can addr;
struct ifreq ifr;
struct can_frame frame;

s = socket(PF_CAN, SOCK_RAW, CAN_RAW);
strcpy(ifr.ifr_name, "vcan0");
ioctl(s, SIOCGIFINDEX, &ifr);

addr.can_family = AF_CAN;
addr.can_ifindex = ifr.ifr_ifindex;
bind(s, (struct sockaddr *)&addr, sizeof(addr));


// read & write
read(sockfd, frame, sizeof(struct can_frame));
write(sockfd, &frame, sizeof(struct can_frame));
```

Capitolo 6

Sviluppo delle singole Applicazioni in C

6.1 Accesso ai dati GPS

I dati ricevuti dal modem GPS sono trasmessi in formato standardizzato NMEA 0183 attraverso interfaccia seriale. Per consentire l'accesso concorrente da parte di più servizi del sistema, è necessario implementare un meccanismo di parsing, elaborazione e condivisione dei dati in tempo reale.

6.1.1 Architettura del sistema di acquisizione

Il sistema di acquisizione GPS è implementato come processo dedicato che gestisce la comunicazione seriale, il parsing dei messaggi NMEA e la condivisione dei dati attraverso file system.

6.1.2 Configurazione dell'interfaccia seriale

La comunicazione con il ricevitore GPS avviene attraverso porta seriale USB (`/dev/ttyUSB1`) configurata con i seguenti parametri:

```
#define SERIAL_PORT "/dev/ttyUSB1"
#define BAUD_RATE B115200

struct termios tty;
// Configurazione 8N1 (8 bit, no parity, 1 stop bit)
tty.c_cflag &= ~PARENB;  // No parity
```

```
tty.c_cflag &= ~CSTOPB;  // 1 stop bit
tty.c_cflag |= CS8;      // 8 bits
cfsetospeed(&tty, BAUD_RATE);
cfsetispeed(&tty, BAUD_RATE);
```

Listing 6.1: Configurazione porta seriale

La configurazione utilizza le API POSIX `termios` per impostare modalità raw, disabilitando elaborazioni software di controllo di flusso e caratteri speciali.

6.1.3 Strutture dati per la rappresentazione GPS

I dati GPS sono organizzati in strutture C ottimizzate per l'accesso in memoria:

```
struct coord {
    float degrees;
    float minutes;
    char direction;    // N, S, E, W
    char padding[3];   // Allineamento a 4 byte
};

struct gps_frame {
    struct coord latitude;
    struct coord longitude;
    struct time time;
    struct date date;
    float speed;
    float course;
    float altitude;
    u_int32_t satellites_in_view;
    u_int64_t last_update;
};
```

Listing 6.2: Strutture dati GPS

Le coordinate sono rappresentate nel formato nativo NMEA (gradi e minuti decimali) per preservare la precisione originale dei dati.

6.1.4 Acquisizione da seriale

L'acquisizione dati e' un ciclo infinito che legge carattere per carattere l'output da seriale. Estrae la stringa compresa tra il carattere di inizio '`$`' e di terminazione '`\n`'. In seguito richiama la funzione di parsing

```
void *serial_read_thread(void *arg) {
    int fd = *((int *)arg);
    char buffer[BUFFER_SIZE];

    while (1) {
        ssize_t bytes_read = read(fd, buffer, sizeof(buffer)
   -1);
        if (bytes_read > 0) {
            // Parsing carattere per carattere
            // Rilevamento inizio/fine messaggio NMEA
            parse_nmea_sentence(sentence);
        }
    }
}
```

Listing 6.3: Thread di lettura seriale

6.1.5 Parsing dei messaggi NMEA

Il sistema implementa parser dedicati per i principali tipi di messaggi NMEA(RMC, GGA, GSV, GNS). Ad esempio:

RMC (Recommended Minimum Course):

```
void parse_rmc(const char *data) {
    char status, lat_dir, lon_dir;
    char time_str[15], date_str[7];
    float speed, course;

    sscanf(data, "%[^,],%c,%[^,],%c,%[^,],%c,%f,%f,%[^,]",
           time_str, &status, lat_str, &lat_dir,
           lon_str, &lon_dir, &speed, &course, date_str);

    if (status == 'A') { // Valid fix
        // Aggiornamento struttura current_frame
    }
}
```

Listing 6.4: Parsing messaggio RMC

6.1.6 Condivisione dati tramite file system

I dati GPS vengono resi disponibili ad altri processi attraverso scrittura su file binario. Per garantire consistenza e atomicità nelle operazioni di aggiornamento, vengono implementate due strategie:

Rename atomico:

```
void update_file() {
    FILE *file = fopen(GPS_FILENAME_TMP, "w");
    fwrite(&current_frame, sizeof(struct gps_frame), 1, file
   );
    fclose(file);

    rename(GPS_FILENAME_TMP, GPS_FILENAME); // Operazione
   atomica
}
```

Listing 6.5: Aggiornamento atomico

File locking: Utilizza le system call `flock()` per controllo dell'accesso esclusivo durante scrittura:

```
int fd = fileno(file);
flock(fd, LOCK_EX);         // Lock esclusivo
fwrite(&current_frame, sizeof(struct gps_frame), 1, file);
flock(fd, LOCK_UN);         // Rilascio lock
```

Listing 6.6: File locking

Shared memory(memoria condivisa) Un terzo approcio prevede l'utilizzo di una zona di memoria condivisa, controllata da un semaforo. Permette un accesso piu' rapido e una velocita' di trasferimento dati maggiore, ma inutile in questa situazione.

6.1.7 Sincronizzazione dell'orologio di sistema

Il sistema implementa sincronizzazione automatica dell'orologio di sistema basata sul tempo GPS ricevuto:

```
void set_system_date() {
    char datetime[100];
    snprintf(datetime, sizeof(datetime),
             "%04d-%02d-%02d %02d:%02d:%02d",
             current_frame.date.year, current_frame.date.
   month,
```

```
            current_frame.date.day, current_frame.time.hour
    ,
            current_frame.time.minute, current_frame.time.
    second);

     char command[200];
     snprintf(command, sizeof(command), "date --set=\"%s\"",
    datetime);
     system(command);
}
```

Listing 6.7: Impostazione data sistema

Questa funzionalità garantisce che il sistema mantenga l'orario preciso anche in ambienti privi di connettività di rete, utilizzando il riferimento temporale satellitare.

6.2 Gps su CAN bus

GPS.c definisce due funzioni di lettura e scritture di dati GPS su CAN bus. I dati GPS vengono ridimensioinati per entrare in un messaggio can da 8 bytes.
Sono stati tentati due approci:

Le struct bit fields, che permettono di definire strutture con risoluzione al bit. Purtroppo queste funzionano e sono efficaci solamente sulla stessa macchina e nello stesso eseguibile. Differentei compilatori e altro possono influenzare il metodo di salvataggio dei dati.

Operazioni bit_wise, sono meno immediate dei bit fields ma operando direttamente sui bit si ottiene la stessa funzionalita' garantendo intercompatibilita' con qualsiasi sistema, fondamentale considerando che il CAN bus si interfaccia con svariate architetture e sistemi.

6.2.1 Struttura dei messaggi

Tre tipi di messaggio sono definiti:

Position: contiene coordinate e altitudine

Values: contiene velocita' e direzione

Time_info: contiene data, ora e numero di satelliti visibili

I primi 4 bit del primo byte definiscono il comando/tipo di pacchetto:

```
int command = data[0] & 0x0F; // take the lower 4bits
switch (command) {
case POSITION_BIT:
    ready = 1;
    read_position_bytes(data, gps);
    break;
```

I successivi bytes i dati. Ad esempio per Values:

```
// Speed with 0.1 precision
frame->speed = ((values_data >> 4) & 0xFFFF) / 10.0f;
// Course with 0.1 precision
frame->course = ((values_data >> 20) & 0xFFFF) / 10.0f;
```

6.3 CAN su TCP/IP fullduplex

Questa applicazione implementa un gateway bidirezionale che permette il controllo remoto di uno o più bus CAN attraverso connessioni TCP/IP. La soluzione consente l'accesso ai dispositivi CAN da sistemi remoti utilizzando socket Unix standard, facilitando l'integrazione in architetture distribuite e sistemi di monitoraggio.

6.3.1 Architettura del gateway CAN-TCP

Il sistema adotta un'architettura multi-threaded che gestisce simultaneamente il traffico bidirezionale tra interfaccia CAN locale e connessione TCP remote. La connessione client viene servita da una coppia di thread dedicati per garantire throughput ottimale.

6.3.2 Configurazione delle interfacce di rete

Il gateway utilizza due tipi di interfacce di comunicazione:

Interfaccia CAN:

```
#define CAN_INTERFACE "can0"

int can_sock;
```

```
if ((can_sock = initCAN(interface)) < 1) {
    exit(EXIT_FAILURE);
}
```

Listing 6.8: Inizializzazione socket CAN

L'interfaccia CAN viene configurata utilizzando socket SocketCAN, standard Linux per l'accesso ai bus CAN tramite API sockets.

Server TCP:

```
#define PORT 33333
struct socket_t socket_d;

if (initSocket(port, &socket_d) != 0) {
    perror("Error init socket");
    exit(EXIT_FAILURE);
}
```

Listing 6.9: Inizializzazione server TCP

Il server TCP accetta connessioni su porta configurabile (default 33333) e gestisce una connessione client alla volta.

6.3.3 Threading e gestione delle connessioni

Per ogni connessione client vengono creati due thread specializzati:

```
void startConnection() {
    new_conn = newConnection(&socket_d);

    int *args = malloc(2 * sizeof(int));
    args[0] = can_sock;    // Socket CAN
    args[1] = new_conn;    // Socket TCP client

    pthread_create(&tr, NULL, can_to_tcp, (void *)args);
    pthread_create(&tw, NULL, tcp_to_can, (void *)args);
}
```

Listing 6.10: Creazione thread per connessione

Thread CAN-to-TCP (`can_to_tcp`): Monitora il bus CAN e inoltra i frame ricevuti al client TCP:

```
void *can_to_tcp(void *vargs) {
    struct can_frame frame;
    fd_set readfds;
```

```
    while (global_stop) {
        FD_ZERO(&readfds);
        FD_SET(can_sock, &readfds);

        int ret = select(can_sock + 1, &readfds, NULL, NULL,
    &timeout);
        if (ret > 0) {
            nbytes = readCAN(can_sock, &frame);
            socketSendCanRaw(tcp_sock, &frame, sizeof(struct
    can_frame));
        }
    }
}
```

Listing 6.11: Forward CAN verso TCP

Thread TCP-to-CAN (`tcp_to_can`): Riceve comandi dal client TCP e li trasmette sul bus CAN:

```
void *tcp_to_can(void *vargs) {
    struct can_socket_t csd;

    while (global_stop) {
        nbytes = socketReadRaw(tcp_sock, &csd, sizeof(struct
    can_socket_t));
        if (nbytes > 0) {
            writeCAN(can_sock, csd.can_id, csd.data);
        }
    }
}
```

Listing 6.12: Forward TCP verso CAN

6.3.4 Strutture dati per protocollo CAN

Il trasferimento dati utilizza strutture ottimizzate per il protocollo CAN:

```
struct can_frame {
    canid_t can_id;     // CAN ID (32-bit)
    __u8 can_dlc;       // Data Length Code
    __u8 data[8];       // Payload (max 8 byte)
};

struct can_socket_t {
    canid_t can_id;
    __u8 data[8];
```

```
};
```

Listing 6.13: Struttura frame CAN

Le strutture mantengono compatibilità binaria con il kernel SocketCAN, eliminando overhead di conversione.

6.3.5 Gestione del ciclo di vita delle connessioni

Il sistema implementa gestione del ciclo di vita delle connessioni. A seguito della chiusura del socket da parte del client, i thread vengono fermati e il server si prepara ad accettare una nuova connessione:

```
while (1) {
    pthread_join(tw, NULL);     // Attendi terminazione
   thread TCP
    pthread_cancel(tr);         // Cancella thread CAN
    pthread_join(tr, NULL);

    global_stop = 1;            // Reset stato per nuova
   connessione
    startConnection();          // Accetta nuova connessione
    sleep(2);
}
```

Listing 6.14: Gestione disconnessioni

6.3.6 Multi connessione

Una seconda versione potenziata dello stesso programma permette di controllare piu' connessioni contemporaneamente, invece di dover creare una nuova istanza del programma per ognuna. In questa variante un thread "connector" viene generato per ogni connessione. Questo si occupa di lanciare la coppia di thread can-tcp/tcp-can. La differenza sta nel thread di lettura del bus CAN. Al posto che aprire una connessione su SocketCAN per ciascuna coppia client/server, questo thread aspetta su una condition variable condivisa tra i thread di lettura. Un flusso di esecuzione parallelo si occupa di leggere dal CAN, scrivere il messaggio su un buffer condiviso e attivare la condition variable. La gestione di tutto questo si e' rivelata complessa ma funzionante. Probabilmente le stesse prestazioni possono essere ottenute con una chiamata di sistema select e un ciclo "classico" senza generare piu' thread.

6.3.7 Interfaccia utente Python-QT

Una semplice interfaccia utente in python con framework QT e' stato sviluppata per ricevere e inviare messaggi sul CAN bus.
All'avvio in base all impostazioni di default si collega ad un determinato ip/porta e permette di visualizzare i messaggi in arrivo. Questo si rivela particolarmnete utile sfruttando la modalita' access point della scheda.

6.4 MQTT con crittografia SSL/TLS

6.4.1 Perché MQTT?

MQTT (Message Queuing Telemetry Transport) è un protocollo di messaggistica leggero, basato su un'architettura publish/subscribe, pensato per scenari in cui la larghezza di banda è limitata, la rete è poco affidabile o i dispositivi sono dotati di risorse limitate — caratteristiche comuni nel contesto dell'embedded Linux e dell'IoT. La sua semplicità lo rende ideale per implementazioni rapide ed efficienti su dispositivi embedded, garantendo allo stesso tempo affidabilità nella comunicazione.

6.4.2 Libreria Paho-MQTT-C

Per l'implementazione lato client si è scelta la libreria `Paho-MQTT-C`, sviluppata all'interno del progetto Eclipse Paho. Rispetto allo sviluppo di una soluzione custom, l'utilizzo di una libreria consolidata come Paho garantisce:

- maggiore affidabilità e sicurezza, grazie al supporto attivo della community;
- conformità allo standard MQTT 3.1.1 e 5.0;
- supporto nativo per SSL/TLS, essenziale per la crittografia dei dati in transito;
- facilità di integrazione su sistemi embedded con toolchain C.

La scelta di non implementare un client MQTT da zero è motivata dalla necessità di evitare problematiche di sicurezza e stabilità già ampiamente risolte da soluzioni open-source consolidate.

MQTTAsync

Come fattore aggiuntivo Paho include piu' varianti, tra cui una completamente **asincrona tramite thread Posix**. Questa garantisce maggiore efficienza e velocita'

ma aggiunge complessita' il che renderebbe difficile garantire il funzionamento con una soluzione fai-da-te.

6.4.3 Generazione dei certificati con OpenSSL

Per garantire la sicurezza delle comunicazioni, MQTT è stato configurato con crittografia SSL/TLS utilizzando certificati X.509[8]. I certificati sono stati generati tramite il toolkit `openssl` [9], seguendo una struttura a tre livelli:

- **Root CA**: autorità di certificazione principale.
- **Signing CA (intermedia)**: utilizzata per firmare i certificati server e client.
- **Certificati Server e Client**: identificano univocamente le entità nel sistema.

1. Creazione della Root CA

```
openssl genrsa -out rootCA.key 2048
openssl req -x509 -new -nodes -key rootCA.key -sha256 -days 1825 \
  -out rootCA.crt -subj "/C=GB/ST=Wiltshire/L=Salisbury/ \
  O=PahoProject/OU=Testing/CN=Root CA"
```

2. Creazione della Signing CA

```
openssl genrsa -out signingCA.key 2048
openssl req -new -key signingCA.key -out signingCA.csr \
  -subj "/C=GB/ST=Wiltshire/O=Paho \
  Project/OU=Testing/CN=Signing CA"
openssl x509 -req -in signingCA.csr -CA rootCA.crt \
  -CAkey rootCA.key -CAcreateserial -out signingCA.crt -days 1825 \
  -sha256 -extfile <(echo "basicConstraints=CA:TRUE")
```

3. Creazione del certificato Server Si definisce un file di configurazione `server.cnf` per specificare i campi alternativi (SAN) e le estensioni:

```
openssl genrsa -out server.key 2048
openssl req -new -key server.key -out server.csr -config server.cnf
openssl x509 -req -in server.csr -CA signingCA.crt \
  -CAkey signingCA.key -CAcreateserial -out server.crt \
  -days 1825 -sha256 -extfile server.cnf -extensions v3_req
```

I certificati delle CA vengono concatenati per creare un file unico:

```
cat signingCA.crt rootCA.crt > allCA.crt
```

4. Creazione del certificato Client

```
openssl genrsa -out client.key 2048
openssl req -new -key client.key -out client.csr \
  -subj "/C=GB/ST=Wiltshire/L=Salisbury/O=Paho \
  Project/OU=Production/CN=test client"
openssl x509 -req -in client.csr -CA signingCA.crt \
-CAkey signingCA.key \
  -CAcreateserial -out client.crt -days 1825 -sha256 \
  -extfile <(echo "basicConstraints=CA:FALSE")
```

5. Unione dei certificati e chiavi Il client utilizza un file `.pem` contenente sia certificato che chiave privata:

```
cat client.crt client.key > client.pem
```

L'utilizzo della crittografia SSL/TLS in combinazione con MQTT e certificati generati tramite OpenSSL garantisce un elevato livello di sicurezza nella comunicazione tra dispositivi embedded. La struttura a più livelli delle autorità di certificazione assicura flessibilità e scalabilità del sistema, permettendo di revocare o rigenerare i certificati in modo centralizzato e controllato.

6.5 GPS su MQTT

In questa sezione viene descritto lo sviluppo di un'applicazione che integra i dati GPS in un sistema di pubblicazione MQTT sicuro, basato su SSL/TLS. L'applicazione, sviluppata in C per ambienti Linux embedded, utilizza la libreria `Paho MQTT C` per la gestione del protocollo MQTT in modalità sicura e prevede il recupero dei dati GPS da più sorgenti (file condiviso, memoria condivisa, oppure direttamente da un'interfaccia seriale), con successivo parsing e pubblicazione.

6.5.1 Architettura del programma

Il flusso principale del programma è strutturato nel modo seguente:

1. Inizializzazione della configurazione MQTT tramite file `/etc/mqtt_gps.conf`.
2. Inizializzazione del client MQTT con certificati SSL/TLS.
3. Lettura ciclica dei dati GPS.
4. Pubblicazione dei dati sul topic MQTT assegnato, personalizzato per ogni dispositivo.
5. Gestione dell'interruzione (SIGINT) per una chiusura ordinata della connessione.

6.5.2 Configurazione e connessione MQTT

Dopo aver caricato i parametri di configurazione (certificati, indirizzo del broker, client ID, ecc.), il client viene inizializzato e connesso:

```
if ((rc = mqtt_create(&client, config.client_id, config.
   address_ssl)) != 0)
{
    printf("Failed to create the client\n");
    rc = EXIT_FAILURE;
    goto exit;
}
if ((rc = mqtt_connect(client, config.ca_root_cert, config.
   client_pem)) != 0)
{
    printf("Failed to connect\n");
    rc = EXIT_FAILURE;
    goto destroy_exit;
}
```

Listing 6.15: Connessione sicura al broker MQTT

6.5.3 Ciclo principale e pubblicazione dei dati

Il programma esegue un ciclo periodico in cui legge i dati GPS da file, li converte in stringa e li invia al broker MQTT. L'intervallo tra due invii può essere personalizzato tramite parametro di linea di comando.

```
while (go)
{
    sleep(interval_s);
    ret = readGPS_file(&gpsData);
    if (ret != 0)
    {
        printf("Error in read gps\n");
        continue;
    }
    sprint_gps_frame(message, gpsData);
    if (mqtt_publish(client, topic, message, strlen(message)
   ,
                     config.client_id, config.qos, config.
   timeout) != 0)
    {
        printf("Error publish\n");
    }
}
```

Listing 6.16: Loop principale con lettura GPS e pubblicazione

6.5.4 GO implementation

Come tentativo di riscrivere parte del codice di questo progetto in go, e' stata creata una variante del publisher di dati GPS su MQTT. Go e' un linguaggio moderno che puo' rivelarsi adatto e potente per sistemi embedded. Dispone di nuovi paradigmi(es. goroutines) e di potenti astrazioni gia' nella libreria standard e viene (cross)compilato in codice nativo binario. Inoltre facilita' la gestione di moduli e librerie esterne e il loro versionamento, del tutto assente in C.
Questa implementazione del GPS pubblica direttamente i dati in formato JSON, utilizzando un approcio ibrido disponibile in questo linguaggio che permette di definire struct pari al C con corrispettivo chiave/valore in JSON.

```
type GpsFrame struct {
    Latitude          Coord    `json:"latitude"`
    Longitude         Coord    `json:"longitude"`
    Time              Time     `json:"time"`
    Date              Date     `json:"date"`
    Speed             float32  `json:"speed"`
    Course            float32  `json:"course"`
    Altitude          float32  `json:"altitude"`
    SatellitesInView  uint32      `json:"satellites_in_view"`
}
```

6.6 CAN su MQTT fullduplex

In questa sezione viene illustrato il funzionamento di un'applicazione embedded per Linux, il cui scopo è quello di gestire uno scambio di messaggi bidirezionale (pub/sub) tra un'interfaccia `CAN bus` e un broker `MQTT` mediante protocollo sicuro `SSL/TLS`. Il sistema rappresenta un'evoluzione del precedente modulo `publish gps/can`, che era limitato alla sola pubblicazione dei messaggi in modalità bloccante, non asincrona.

Questa versione si basa sull'utilizzo della libreria `paho-mqtt-c` nella variante `asynch`, con supporto multithreading. Tale approccio consente una maggiore reattività, specialmente nella ricezione dei messaggi MQTT da inoltrare sul bus CAN.

L'applicazione stabilisce una connessione sicura con il broker MQTT utilizzando certificati `X.509`, si sottoscrive a un topic di controllo e configura le relative *callback*. In parallelo, un ciclo principale attende messaggi dal bus CAN (tramite `SocketCAN`), che vengono convertiti e pubblicati verso il broker MQTT.

Durante il funzionamento, quando un nuovo messaggio viene ricevuto sul bus CAN, questo viene convertito in stringa e pubblicato via MQTT:

```
canToString(message, &canData);
if (mqtt_publish_withCanId(&mqtt, message, strlen(message),
   canData.can_id) != 0)
{
    printf("Error publish\n");
}
```

Listing 6.17: Pubblicazione messaggio CAN come stringa

Viceversa, al ricevimento di un messaggio da MQTT, il sistema attiva la `callback messageArrived`, estrae il `can_id` dal topic, decodifica il payload e invia il frame sul bus CAN:

```
int messageArrived(void* context, char* topicName, int
   topicLen, MQTTAsync_message* message)
{
    int canId = extractCanId(topicName, message->payloadlen)
   ;
    if (canId > 0 && message->payloadlen <= 8){
        struct can_frame frame;
        frame.can_id = canId;
        frame.can_dlc = message->payloadlen;
        memcpy(frame.data, message->payload, message->
   payloadlen);
```

```
        writeFrameCAN(can_sock, &frame);
    }
}
```

Listing 6.18: Callback per ricezione messaggi MQTT e invio su CAN

6.6.1 Conversione Can_id–Topic

Una caratteristica chiave del sistema è la mappatura tra gli identificativi CAN (`can_id`) e i `topic` MQTT. Questo consente di semplificare il payload dei messaggi, evitando di dover includere il `can_id` nel contenuto, poiché può essere derivato direttamente dal nome del `topic`.

La convenzione adottata è la seguente:

$$\texttt{topic/} < \textit{vehicle\_id} > \texttt{/CAN/} < \textit{can\_id} >$$

Dove `can_id` è espresso in formato esadecimale. L'estrazione avviene nel seguente modo:

```
int extractCanId(const char *topic_src, int topic_src_len) {
    char *last_slash = strrchr(topic_src, '/');
    if (last_slash == NULL) {
        return -1;
    }
    char *can_id_str = last_slash + 1;
    int can_id = 0;
    if (sscanf(can_id_str, "%x", &can_id) != 1) {
        return -1;
    }
    return can_id;
}
```

Listing 6.19: Estrazione del CAN ID dal topic

Attualmente, il payload dei messaggi pubblicati include sia il contenuto dati che l'identificativo CAN. Tuttavia, grazie alla codifica del `can_id` nel nome del topic, è possibile ridurre l'overhead dei messaggi pubblicati, trasmettendo solamente il payload.

6.7 Client di pubblicazione CAN

Durante lo sviluppo sono emerse le ovvie limitazioni e difficolta' della fusione tra CAN bus e MQTT. CAN e' un protocollo estremamente piu' semplice e efficace per comunicazioni real time tra schede interconnesse. La latenza e overhead imposte da MQTT portano ad un rapido degrado delle prestazioni. Per garantire il corretto flusso di dati ad alta velocita' e larghezza di banda sono state utilizzate alcune tecniche di programmazione.

6.7.1 Confronto tra CAN e MQTT

Il protocollo **CAN** è progettato per comunicazioni real-time affidabili su una rete locale, con messaggi brevi (fino a 8 byte per frame standard) e trasmissione deterministica. Dall'altro lato, **MQTT** è orientato a sistemi distribuiti su larga scala motlo piu' complessi.

La fusione tra i due protocolli presenta criticità:

- **Overhead e latenza**: la necessità di impacchettare messaggi CAN in batch e inviarli tramite MQTT comporta una perdita di reattività.
- **Compatibilità semantica**: CAN trasmette dati binari a basso livello, mentre MQTT si basa su stringhe o payload più complessi.
- **Sincronizzazione**: il flusso continuo e asincrono di messaggi CAN va gestito con cura per non perdere dati nella transizione verso MQTT.

6.7.2 Batching e double buffering

Per ridurre l'impatto prestazionale dell'integrazione, sono state adottate tecniche di batching e buffering:

- Raccolta di messaggi in batch di dimensione massima predefinita (`BATCH_SIZE = 512`).
- Buffer circolare doppio (*double-buffering*) per evitare blocchi durante l'invio asincrono.
- Invio forzato del batch su **timeout** per garantire latenza determinata (`BATCH_TIMEOUT_MS = 100`).
- Utilizzo di un thread dedicato all'invio MQTT, con sincronizzazione tramite mutex e condition variable.

6.7.3 Struttura del codice

Il codice sviluppato implementa un bridge tra CAN bus e MQTT. All'avvio, vengono inizializzati il socket CAN, la configurazione MQTT e le strutture di batching.

```
typedef struct {
    char buffer[MAX_BATCH_BUFFER];
    int count;
    int buffer_len;
    struct timeval last_message_time;
} MessageBatch;
```

Listing 6.20: Definizione della struttura di batch

La struttura `MessageBatch` funge da accumulatore per i messaggi convertiti da CAN a stringa. I messaggi vengono concatenati in `buffer`, con un contatore per il numero di messaggi e un marcatore temporale per valutare il timeout.

Durante il ciclo principale:

1. Viene letto un messaggio dal bus CAN con `readCAN()`.
2. Il messaggio viene convertito in stringa tramite `canToString()`.
3. Se il batch corrente è pieno o è scaduto il timeout, viene attivato il thread MQTT per l'invio.
4. Si passa al buffer di backup mentre l'altro viene svuotato in background.

Questo permette il lavoro parallelo di acquisizione dei dati dal bus e invio sulla rete tramite MQTT.

```
if (should_flush_batch(working_batch)) {
    pthread_mutex_lock(&lock);
    flushing_batch = working_batch;
    working_batch = (working_batch == &batch1) ? &batch2 : &
   batch1;
    thread_wait = 1;
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&lock);
}
```

Listing 6.21: Condizione di flush asincrono

La funzione `should_flush_batch()` determina se un batch deve essere inviato in base al numero di messaggi accumulati o al timeout.

```
void* mqtt_thread(void* arg) {
    while (1) {
        pthread_mutex_lock(&lock);
        while (thread_wait == 0) {
            pthread_cond_wait(&cond, &lock);
        }
        flush_batch(flushing_batch, client, config.topic,
   config.client_id, config.qos, config.timeout);
        reset_batch(flushing_batch);
        thread_wait = 0;
        pthread_mutex_unlock(&lock);
    }
    return NULL;
}
```

Listing 6.22: Logica di invio batch da thread secondario

Il thread MQTT attende l'attivazione tramite una `condition variable`, quindi pubblica i messaggi tramite `mqtt_publish()`, e infine resetta il batch svuotato. Questo permette la continua esecuzione del flusso principale senza interruzioni dovute a operazioni I/O sulla rete.

Di seguito viene riportata un'immagine che mostra le performance del sistema. Il test dimostra un throughput > 1000 messagi/secondo.

- Sulla destra la scheda: in alto uno script python che genera e invia messaggi su SocketCAN, in basso il programma c di buffering e invio dei dati can.
- Sulla sinistra il server con il programma c di ricezione dei dati e scrittura su file.

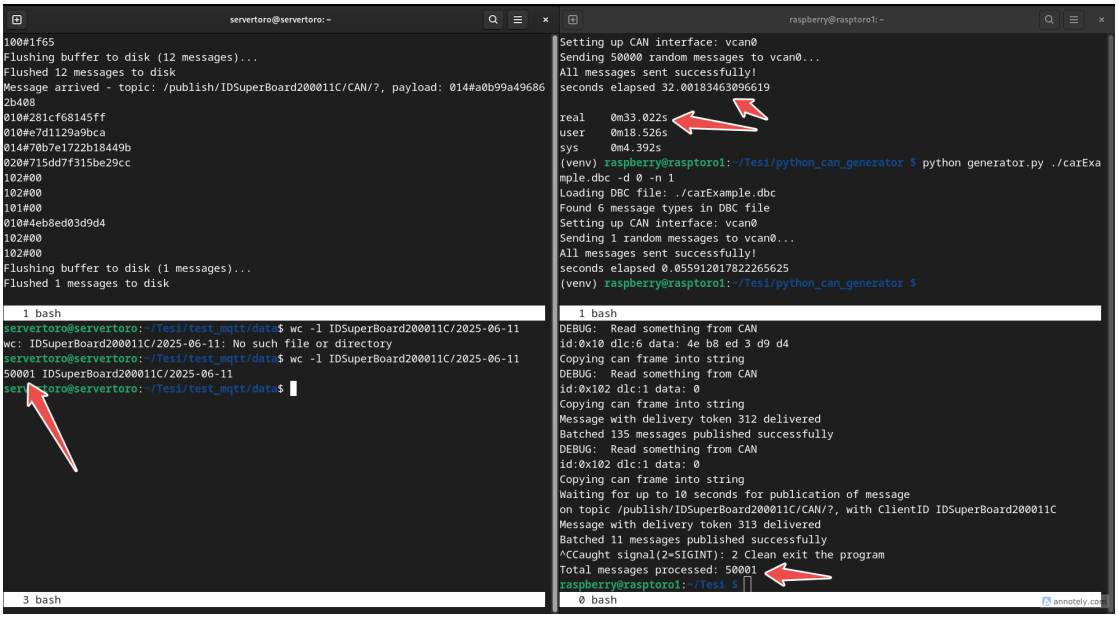

Figura 6.1: Test di throughput CAN-to-MQTT

Capitolo 7

Sviluppo del sistema di visualizzazione

In seguito all'analisi dell'hardware, alla personalizzazione del sistema operativo, e all'esplorazione delle funzionalità software, questa sezione illustra l'integrazione finale del tutto. L'obiettivo è la realizzazione di un sistema completo in grado di rendere disponibili all'utente finale i dati acquisiti dalla scheda installata a bordo di un veicolo.

CAN bus database(DBC)

Al fine di permettere la visualizzazione di dati analizzati e non di soltanto dati privi di valore e' necessario introdurre il concetto di database DBC (DataBase CAN) [10]. Questo è un formato standard utilizzato per descrivere la struttura dei messaggi e dei segnali trasmessi sulla rete CAN. All'interno sono codificate una serie di informazioni: gli identificatori dei messaggi, i segnali contenuti, le unità di misura, i fattori di scala e gli offset. Il DBC permette l'interpretazione coerente dei dati grezzi ricevuti dal bus CAN, rendendo possibile la decodifica automatica delle informazioni in valori significativi per l'applicazione. Esistono numerose librerie pubbliche che permettono l'analisi programmatica dei dati. In questo caso e' stata utilizzata **cantools** [11], la quale rappresenta lo standard per le comunicazioni con bus CAN su python.

7.1 Stack tecnologico

Lo sviluppo si è articolato in due principali varianti tecnologiche.

- **Variante 1 - Soluzione custom:**
  - Client di pubblicazione in C
  - Server in C per la ricezione (subscriber)
  - Backend in Python per parsing e API REST
  - Interfaccia web scritta in React e TypeScript
- **Variante 2 - Soluzione basata su software open source:**
  - Client di pubblicazione in C
  - Subscriber, parser e API integrati in Python
  - Database temporale InfluxDB (in esecuzione via Docker)
  - Sistema di visualizzazione Grafana (in esecuzione via Docker)

7.2 Interfaccia web custom

7.2.1 Storing dei dati

Il subscriber in C opera lato server e si sottoscrive ai topic MQTT del tipo `topic/ID-<16bytes>-C` per dati CAN e `topic/ID-<16bytes>-G` per dati GPS. I messaggi ricevuti vengono salvati in una struttura a file come segue:

```
data/
 ID-<16bytes>-C/
    YYYY-MM-DD.log  (dati CAN)
 ID-<16bytes>-G/
     YYYY-MM-DD.log  (dati GPS)
```

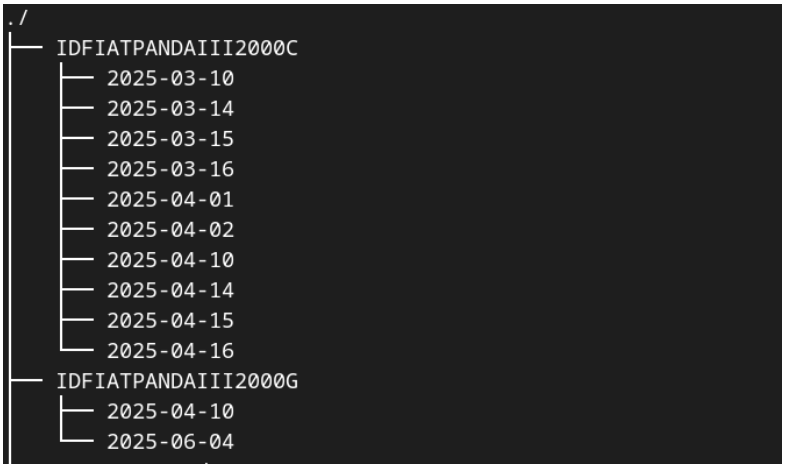

Figura 7.1: Esempio struttura dati

Il formato dei file è:

- **CAN:** `hh:mm:ss canid canpayload(UTF-8 <=8byte)`
- **GPS:** `hh:mm:ss latdd latmm latss NS londd lonmm lonss EW speed course altitude satellites`

Il programma e' stato adattato al sistema client per ricevere i messaggi in batch che includono piu' dati in una singola trasmissione MQTT. Questa soluzione sfrutta l'efficienza del linguaggio C e l'accesso diretto alle system call per massimizzare le prestazioni in fase di scrittura. Rimane comunque piu' orientata al debug, mantenendo i dati come stringa per essere facilmente analizzabili.

7.2.2 Backend Python API

Il backend Python legge i file generati dal subscriber e offre un'interfaccia REST per l'accesso ai dati. I file vengono parsati dinamicamente:

- Il GPS viene decodificato direttamente dai formati stringa.
- I dati CAN vengono interpretati tramite un file `.dbc`, senza alcuna logica hardcoded, ma sfruttando l'analisi strutturata dei segnali (`SG_`) e dei messaggi (`BO_`).

Per decodificare i messaggi CAN e' stata utilizzata la libreria `cantools` di python che include la comunicazione con SocketCAN e le funzioni di parsing dei dati.

```
import cantools
id_, data_hex = msg.split('#')
    if id_ and data_hex:
        try:
            id_ = int(id_, 16)
            data = self.dbc.decode_message(id_, bytes.fromhex(
   data_hex))
            rounded_data = {key: round(value, 2) for key, value in
    data.items()}
            name = self.dbc.get_message_by_frame_id(id_).name
```

Listing 7.1: Python cantools

Il servizio espone REST API per ottenere i dati filtrati per veicolo e intervallo temporale. I dati CAN vengono restituiti in formato JSON, rispettando la struttura gerarchica del file `dbc`, raggruppando i segnali per oggetto (`BO_`).

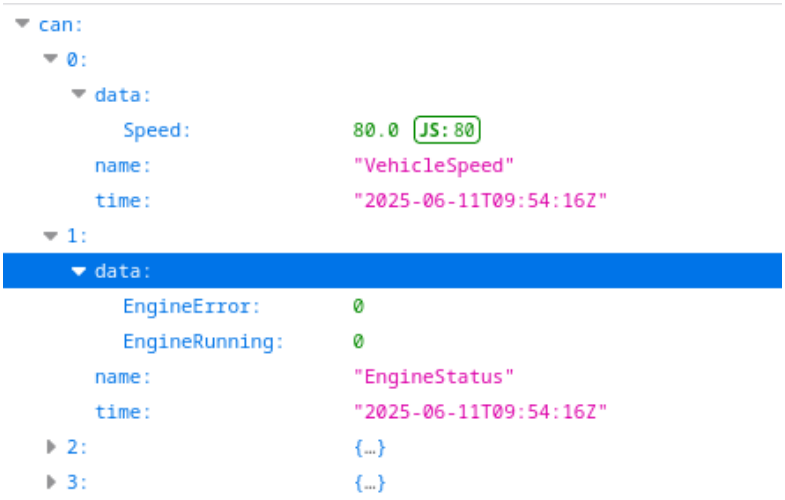

Figura 7.2: Esempio api python json can

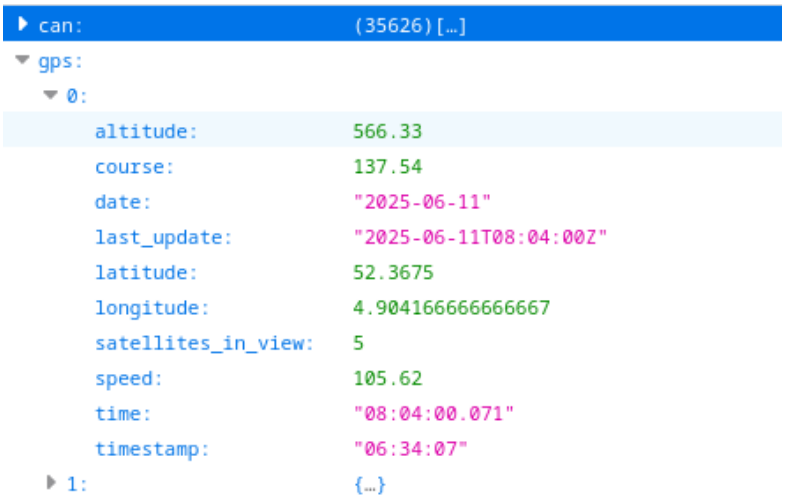

Figura 7.3: Esempio api python json gps

7.2.3 Web app per gestire i veicoli

L'interfaccia utente è realizzata in React con TypeScript. Essa consente:

- Visualizzazione della lista dei veicoli monitorati.
- Selezione di un veicolo per accedere a:
  - Mappa OpenStreetMap che traccia il percorso utilizzando le coordinate GPS ricevute.
  - Dashboard CAN: i segnali decodificati vengono mostrati in "card" dinamiche, raggruppate secondo gli oggetti (`BO_`) presenti nel file `dbc`.
  - Download diretto di tutti i dati CAN(gia' interpretati) e gps come json

La dashboard e' aggiornata agli ultimi dati ricevuti dai veicoli, e permette di scaricare di default i dati dei 30 giorni passati.

L'interfaccia è completamente dinamica e si adatta automaticamente a nuove configurazioni di `dbc`, rendendo il sistema scalabile e indipendente dalla struttura e quantita' dei dati.

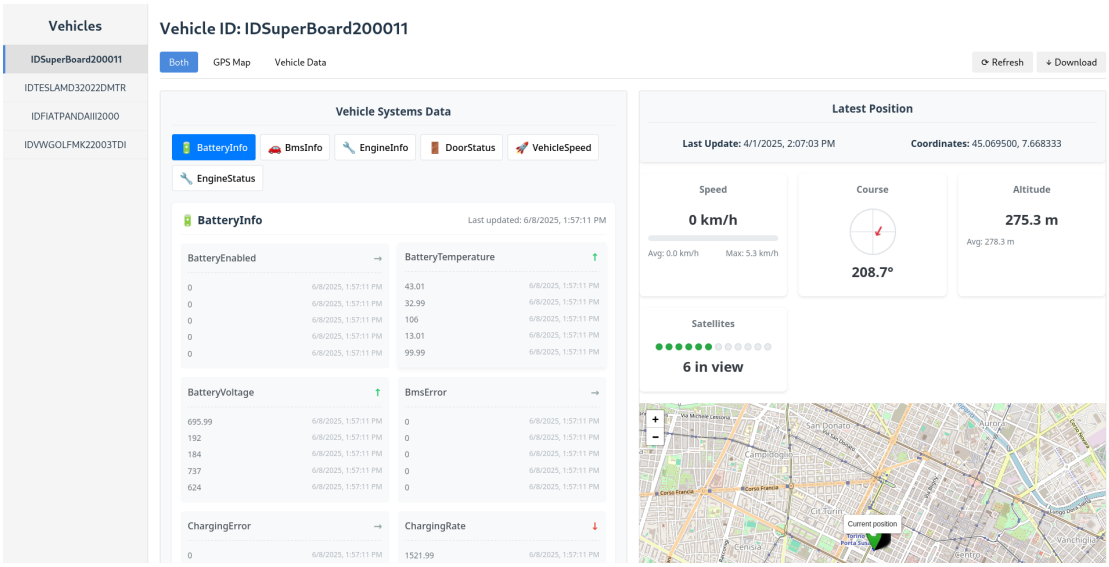

Figura 7.4: Interfaccia web in react

7.2.4 Limiti di un approccio custom

L'approccio custom, sebbene flessibile e performante, presenta alcune criticità:

- Complessità crescente nella gestione del sistema
- Assenza di indicizzazione efficiente dei dati
- Difficoltà nell'implementazione di filtri interattivi e visualizzazioni avanzate
- Maggiore esposizione a instabilità software e potenziali falle di sicurezza

7.3 Soluzioni OSS

Per mitigare le limitazioni sopra citate, è stata realizzata una seconda versione del sistema basata interamente su software open source, largamente adottato e testato, in grado di offrire stabilità, sicurezza e funzionalità avanzate.

7.3.1 Python dbc parser API

La logica del backend Python è stata riadattata per operare in tempo reale. In questa implementazione:

- La ricezione MQTT avviene tramite la libreria `paho-mqtt` in Python
- I messaggi ricevuti vengono immediatamente parsati e bufferizzati
- I buffer vengono periodicamente inviati a InfluxDB tramite la relativa API Python
- I timestamp vengono applicati alla ricezione (CAN) o preservati (GPS)
- L'ID veicolo viene estratto dal topic MQTT per una corretta indicizzazione su Influx

L'intero processo è dinamico e scalabile: i bucket Influx vengono creati automaticamente all'arrivo di nuovi veicoli, sfruttando le API ufficiali.

La struttura dei dati e' salvata seguendo il seguente schema:

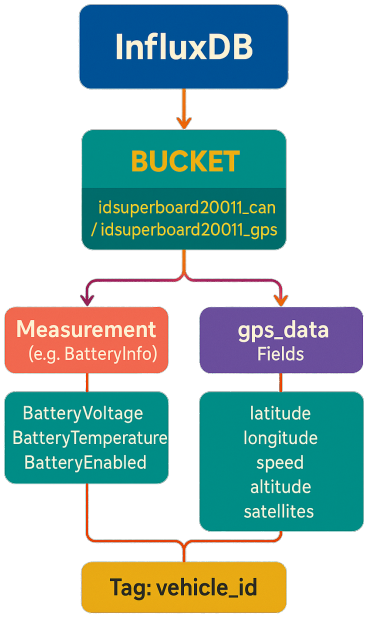

Figura 7.5: InfluxDB data schema

La gestione dei thread è così strutturata:

- **Callback asincrona MQTT:** Inserisce i messaggi in una coda gestita dal sistema operativo
- **Worker thread:** Estrae i messaggi dalla coda, applica il parsing e li inserisce in un buffer condiviso
- **Flush thread:** Svuota il buffer a intervalli regolari o a riempimento completato

Un lock consente di gestire l'accesso concorrente al buffer condiviso. Quando il buffer si riempe vengono bloccati per un instante il worker e flush thread, copiato al volo il buffer e resettato l'originale. Questo meccanisco permette un interruzione minima del flusso di lavoro, senza dover aspettare le api REST verso influx.

7.3.2 InfluxDB e Grafana

InfluxDB [12] è un database temporale progettato per gestire elevati volumi di dati a bassa latenza, come quelli generati da sensori. Le sue caratteristiche principali includono:

- Scrittura e query ad alte prestazioni
- Gestione automatica di bucket, retention e compressione
- Struttura dati basata su: **bucket**, **measurement**, **field**, **tag**

Grafana [13] è una piattaforma open source per la visualizzazione di dati. Le sue caratteristiche chiave includono:

- Interfaccia grafica avanzata per la creazione di dashboard
- Integrazione diretta con InfluxDB
- Supporto a filtri temporali, aggiornamento automatico, e interazioni dinamiche

Questa combinazione offre una soluzione scalabile e robusta per la visualizzazione dei dati in tempo quasi reale. Grafana permette una personalizzazione molto avanzata della visualizzazione del sistema.

Di seguito vengono riportati due esempi di dashboard molto semplici:

Figura 7.6: Dashboard grafana 1

Figura 7.7: Dashboard grafana 2

Docker

La implementazione dei precedenti servizi puo' essere inglobata all'interno di container Docker [14]. Questo offre numerosi vantaggi, tra i quali: semplicita' di utilizzo, scalabilita', portabilita' e senza una significativa riduzione delle prestazioni. I due servizi vengono compilati e avviati tramite un docker compose file, che descrive la procedura di avvio, il contenuto dei container, le connessioni di rete tra essi e permette di personalizzarne alcuni aspetti, ad esempio password, account, o piu' nello specifico impostare la risoluzione minima temporale in grafana da 5 secondi a 1 secondo.

Capitolo 8

Conclusioni e Sviluppi Futuri

8.1 Sintesi dei risultati ottenuti

Questo lavoro di tesi ha raggiunto con successo gli obiettivi prefissati, dimostrando la fattibilità e le potenzialità di un sistema embedded Linux per applicazioni automotive con connettività wireless. Partendo da una scheda custom basata su un SoM iMX8MP di Variscite, è stata configurata un'immagine Linux personalizzata tramite Yocto, gettando le basi per lo sviluppo di applicativi software dedicati.

Sono state testate e validate le principali periferiche della scheda, con particolare attenzione al modem Quectel per la connettività LTE e GPS e alle interfacce CAN, tramite api e strumenti di Linux. Inoltre, è stato implementato un servizio di telemetria sicura tramite il protocollo MQTT con crittografia SSL, garantendo un canale di comunicazione affidabile per la trasmissione dei dati del veicolo.

Il culmine di questo progetto è rappresentato dallo sviluppo di una piattaforma completa per la visualizzazione dei dati raccolti. Questa piattaforma, accessibile tramite un'interfaccia web, permette di monitorare in tempo reale la posizione GPS del veicolo su una mappa e di analizzare i dati provenienti dal bus CAN, decodificati attraverso un database DBC.

8.2 Limiti del lavoro svolto

Nonostante i risultati raggiunti, il progetto presenta alcune limitazioni:

- La gestione dei dati avviene sotto forma di stringhe per facilitare il debug, ma ciò implica un maggior consumo di banda e un carico computazionale superiore rispetto alla gestione in formato binario.
- Non sono stati eseguiti test in un contesto veicolare reale, limitando la validazione in ambienti simulati.
- L'analisi prestazionale e la robustezza del sistema non sono state approfondite in scenari di carico estremo o condizioni ambientali sfavorevoli.
- Alcuni componenti del sistema, come il backend Python e l'interfaccia web, sono ancora in fase prototipale.

8.3 Possibili sviluppi e miglioramenti futuri

Numerosi sviluppi futuri sono stati identificati per migliorare e potenziare la piattaforma:

- Interpretazione dei messaggi CAN direttamente on edge, spostando parte della computazione direttamente sulla scheda
- Riscrittura del backend(in Go) per migliorare le prestazioni, la gestione della concorrenza, affidabilita' e la scalabilità.
- Supporto al download dei dati in formato compresso (es. MF4) dal sistema embedded, per facilitare l'analisi offline.
- Migrazione della gestione dei dati da stringhe a rappresentazione binaria, per ridurre la larghezza di banda e il carico computazionale.
- Aggiunta di test automatizzati per garantire la stabilità e l'affidabilità dell'intero sistema.

Oltre al lavoro svolto, rimangono ancora da esplorare numerose parti della scheda e SoC utilizzato. Un esempio interessante sarebbe sfruttare la capacità AI del SoC i.MX8MP, in particolare dell'NPU, per l'analisi in tempo reale dei dati CAN a bordo veicolo.

Per concludere quindi, questa tesi rappresenta un esempio concreto di utilizzo della scheda, ma costituisce anche un punto di partenza per esplorare molteplici possibilità applicative.

Bibliografia

[1] Variscite. *DART-iMX8MP*. Accessed: 2025-06-11. n.d. URL: https://www.variscite.it/product/system-on-module-som/cortex-a53-krait/dart-mx8m-plus-nxp-i-mx-8m-plus/ (cit. a p. 9).

[2] Quectel. *Quectel EG225-G*. Accessed: 2025-06-11. n.d. URL: https://www.quectel.com/product/lte-eg25-g/ (cit. a p. 11).

[3] Yocto Project. *Yocto wiki*. Accessed: 2025-06-11. n.d. URL: https://wiki.yoctoproject.org/wiki/Main_Page (cit. a p. 13).

[4] Variscite. *Yocto docker*. Accessed: 2025-06-11. n.d. URL: https://variwiki.com/index.php?title=Docker_Build_Environment&release=mx8mp-yocto-kirkstone-5.15.71_2.2.0-v1.3 (cit. a p. 15).

[5] Variscite. *Yocto sdk*. Accessed: 2025-06-11. n.d. URL: https://variwiki.com/index.php?title=Yocto_Toolchain_installation&release=mx8mp-yocto-kirkstone-5.15.71_2.2.0-v1.3 (cit. a p. 16).

[6] Variscite. *Configuring WiFi access point with hostapd*. Accessed: 2025-06-11. n.d. URL: https://variwiki.com/index.php?title=Wifi_NetworkManager&release=mx8mp-yocto-kirkstone-5.15.71_2.2.0-v1.3 (cit. a p. 24).

[7] Linux foundation. *SocketCAN*. Accessed: 2025-06-11. n.d. URL: https://docs.kernel.org/networking/can.html (cit. a p. 27).

[8] ssl.com. *SSL x509*. Accessed: 2025-06-11. n.d. URL: https://www.ssl.com/it/faq/che-cos%27%C3%A8-un-certificato-x-509/ (cit. a p. 40).

[9] Openssl. *Openssl*. Accessed: 2025-06-11. n.d. URL: https://docs.openssl.org/3.4/man1/openssl/ (cit. a p. 40).

[10] CSS Electronics. *Can DBC file format*. Accessed: 2025-06-11. n.d. URL: https://www.csselectronics.com/pages/can-dbc-file-database-intro (cit. a p. 50).

[11] Python. *cantools*. Accessed: 2025-06-11. n.d. URL: https://cantools.readthedocs.io/en/latest/ (cit. a p. 50).

[12] Influx. *InfluxDB*. Accessed: 2025-06-11. n.d. URL: `https://docs.influxdata.com/influxdb3/core/` (cit. a p. 58).

[13] Grafana. *Grafana*. Accessed: 2025-06-11. n.d. URL: `https://grafana.com/docs/grafana/latest/?pg=oss-graf&plcmt=hero-btn-2` (cit. a p. 58).

[14] Docker. *Docker*. Accessed: 2025-06-11. n.d. URL: `https://docs.docker.com/reference/` (cit. a p. 60).