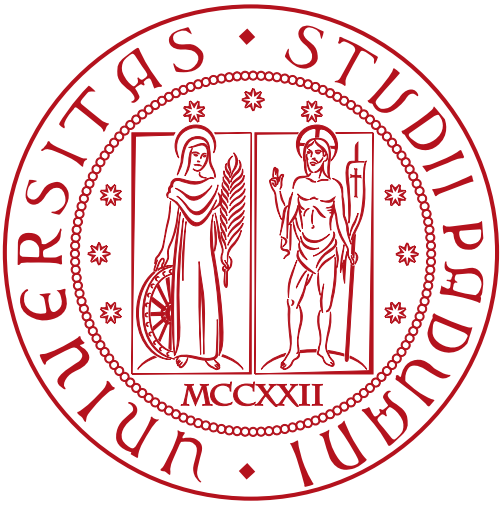
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Modernizzazione delle Applicazioni dei Sistemi di Controllo

Tesi di laurea

20 Settembre 2024

Relatore

Prof. Tullio Vardanega

Laureando

Matteo Rango

Matricola 2008066

ANNO ACCADEMICO 2023-2024

Sommario

Attualmente, il *software* dei sistemi di controllo è ancora prevalentemente sviluppato in linguaggio *C*. Tuttavia, il codice che ne deriva è spesso difficile da leggere e quindi anche da *mantenere*. Vista anche la sua natura estremamente versatile, facilita l'introduzione di errori di sicurezza. Durante il tirocinio, ho riprogettato e sviluppato, in *Rust*, un *driver* per il modulo dedicato alle conversioni da segnali analogici a digitali, ad approssimazioni successive (*SAR*), del microcontrollore *Infineon Aurix TC375*. Il lavoro ha evidenziato come è possibile, grazie a *Rust*, rimuovere un'intera classe di errori e garantire sicurezza e stabilità al sistema.

In questo documento, espongo il lavoro che ho svolto con la seguente suddivisione: nel primo capitolo descrivo l'azienda all'interno della quale sono stato inserito, fornendo una panoramica degli strumenti e dei processi adottati. Nel secondo capitolo espongo lo scopo del tirocinio, i prodotti attesi, il metodo di lavoro e gli obiettivi personali. Nel terzo capitolo descrivo nel dettaglio le attività svolte e metodi di approccio a problemi per me nuovi, facendo anche un esempio completo; alla fine del capitolo presento i risultati ottenuti e le conclusioni tratte. Per finire, nel quarto capitolo, espongo una retrospettiva finale in termini di conoscenze acquisite e di soddifacimento delle parti coinvolte.

Oltre alla struttura appena descritta, ho adottato le seguenti convenzioni tipografiche:

- gli acronimi, le abbreviazioni e i termini di uso non comune menzionati, vengono definiti nel Glossario, situato alla fine del presente documento;
- per la prima occorrenza dei termini riportati nel glossario viene utilizzata la seguente nomenclatura: parola$_G$;
- i termini in lingua straniera o facenti parte del gergo tecnico sono evidenziati con il carattere *corsivo*;
- dove non diversamente specificato, l'uso del "noi" si riferisce a me e ai miei tutor aziendali.

Non c'è limite al peggio, negli algoritmi come nella vita.

— Paolo Baldan

Ringraziamenti

Innanzitutto, vorrei esprimere la mia più profonda gratitudine al Prof. Tullio Vardanega, relatore della mia tesi, per il supporto fornitomi durante la stesura e per la pazienza di rispondere ad ogni mio pensiero inquieto.

Desidero ringraziare con affetto i miei genitori, per il sostegno durante tutti gli anni di studio. Mamma, alla fine, tutti i capelli bianchi che ti ho fatto venire, hanno dato i loro frutti.

Un grazie, dal profondo del cuore, a Sofia, per i momenti condivisi assieme. Con te affianco, posso raggiungere qualsiasi vetta.

Padova, Settembre 2024 *Matteo Rango*

Indice

Elenco delle Figure

Elenco delle Tabelle

Capitolo 1.

Il contesto aziendale

In questo capitolo presento una panoramica generale dell'azienda in cui ho svolto il tirocinio e qual è il rapporto tra quest'ultima e le nuove tecnologie.

1.1. *Bluewind* S.R.L.

L'azienda *Bluewind*, con sede a Castelfranco Veneto (TV), offre servizi di ingegneria nel mondo dei sistemi *embedded*, principalmente attiva nei settori: automobilistico, medico e industriale.

Il punto di forza dell'azienda è quello di poter contare su *team* multifunzionali, che permettono di coprire a trecentosessanta gradi l'intero processo di creazione di un prodotto:

- analisi;
- progettazione;
- certificazioni di conformità;
- implementazione.

Essere in grado di operare in tutte le fasi di creazione, è permesso anche grazie alla forte componente di formazione che viene sempre messa in primo piano quando necessaria. Durante il tirocinio ho potuto sperimentarlo in prima persona grazie ai *tutor* e ai colleghi, che mi hanno affiancato durante tutto il percorso, offrendomi le loro conoscenze e fornendomi degli interessanti spunti di riflessione. Oltre a questo ho anche partecipato a due sessioni aziendali, della durata di un giorno, in cui abbiamo trattato rispettivamente il tema Agile$_G$, e come migliorare i

processi interni, e il tema della ristrutturazione e della definizione dei ruoli in azienda. Questo dimostra come *Bluewind* investa molto non solo nelle nuove tecnologie, ma anche nel miglioramento continuo del proprio ambiente di lavoro.

La principale attività dell'azienda, prevede di portare un prodotto *software*, già esistente o meno, in una condizione tale per cui possa essere certificato secondo uno specifico *standard*. Altra parte dei ricavi arriva dalla rivendita di prodotti e licenze sempre in ambito *embedded*, per programmi e librerie *software* di proprietà delle aziende di cui *Bluewind* è *partner*. La clientela risulta di conseguenza molto varia: dai clienti finali che hanno bisogno di una soluzione *software* completa, ad altre aziende informatiche che necessitano di attuare sistemi di sicurezza, spesso all'interno di programmi o sistemi che hanno già sviluppato.

1.2. Organizzazione interna

All'interno dell'azienda vi sono due reparti principali:

- Research and Development (R&D)$_G$, ossia il reparto di ricerca e sviluppo, in cui ero inserito;
- *Marketing* and *Sales*, ossia il reparto che si occupa della vendita di prodotti e della parte pubblicitaria;

Il reparto *R&D* è quello in cui si svolgono tutte le attività di produzione, ed è suddiviso in *team*, ognuno dei quali lavora ad un progetto, con persone che possono essere presenti in più *team* diversi. Questi sono guidati da un product owner (PO)$_G$, persona che si occupa di interfacciarsi con il cliente e di assicurarsi che il lavoro venga consegnato nei modi e tempi previsti. Ogni *team* include persone con diversi ruoli, tra cui:

- analisti *software*;
- analisti della sicurezza;
- progettisti;
- sviluppatori.

A capo di tutto il reparto c'è una persona che ha il ruolo di *Head of R&D*, che si occupa del buon funzionamento del reparto stesso.

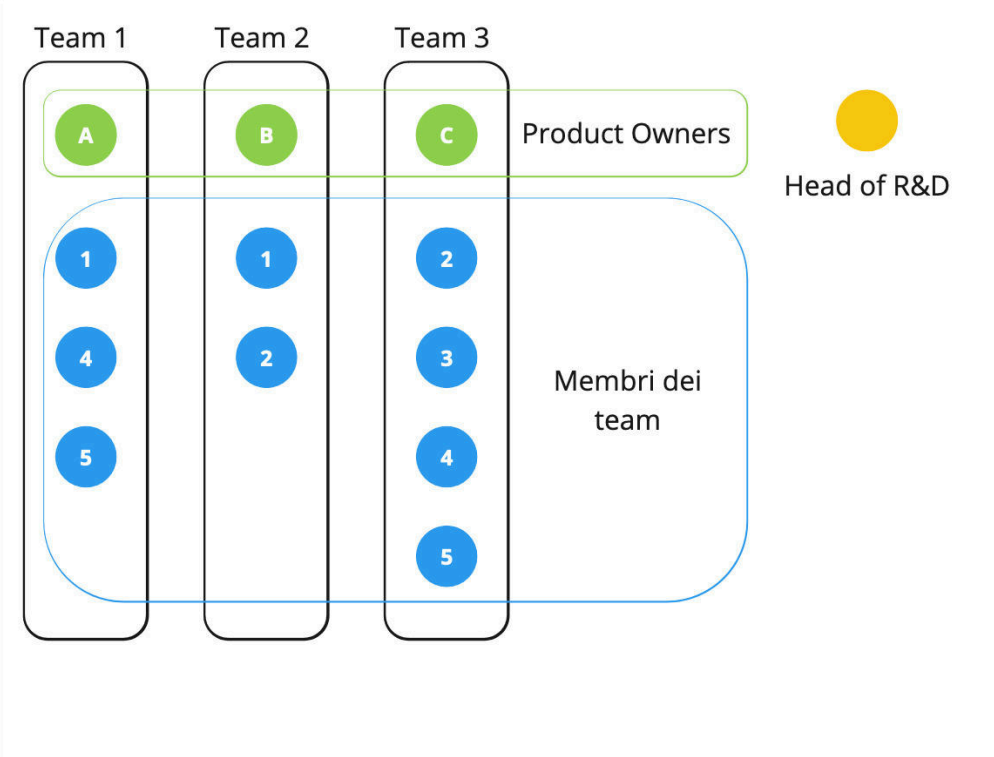


Figura 1.1: rappresentazione grafica di esempio della suddivisione del reparto *R&D* in Bluewind.

1.3. Il metodo *agile*

Tutti i processi trovano fondamento nella metodologia *agile*, una serie di principi elencati nel *"Manifesto Agile for Software Development"* [4], che mirano a garantire la minimizzazione del rischio. Essa utilizza un approccio centrato sulla consegna iterativa e incrementale di un prodotto funzionante, che si avvicina sempre di più al prodotto finale. Oltre a questo, anche la stretta collaborazione con il cliente e una forte componente di adattamento e apertura al cambiamento, garantiscono questa minimizzazione.

La metodologia *agile*, benché nasca all'interno del mondo dello sviluppo *software*, si può adattare bene a qualsiasi tipo di ruolo ed è proprio

quello che accade in *Bluewind*. Per quello che ho potuto osservare, non tutti all'interno dell'azienda utilizzano questa metodologia, ma sicuramente ne sono influenzati.

Quello che ho potuto osservare più da vicino è come questi principi prendano vita all'interno di un framework$_G$, cioè una serie di metodologie di supporto, utilizzate per lo sviluppo *software*, che descriverò in dettaglio nella sezione "Metodo di lavoro".

1.4. Tecnologie e strumenti

1.4.1. *Infineon Technologies Aurix TC375*

I microcontrollori della ditta *Infineon Technologies* sono dei prodotti ad altissime prestazioni e altamente affidabili. Infatti sono diventati lo *standard* in diversi settori tra cui quello automobilistico, quello spaziale, quello industriale e quello della sicurezza.

Figura 1.2: foto della scheda sulla quale ho sviluppato il progetto di tirocinio presa dal sito web [1] ufficiale.

La scheda di sviluppo che ho utilizzato era una *Aurix TC375 Lite Kit*, prodotta dalla stessa azienda, che offre un ambiente di sviluppo *hardware* pre-configurato, pensato per dimostrare tutte le funzionalità del microcontrollore saldato sulla scheda, che è proprio l'omonimo *Aurix TC375*.

Di quest'ultimo, ho studiato in particolare il modulo Enhanced Versatile Analog-to-Digital Converter $(EVADC)_G$, ossia un orchestratore di diversi analog-to-digital converter $(ADC)_G$, componenti elettronici che sono in grado di convertire dei segnali analogici in segnali digitali, attraverso un sistema chiamato successive approximation register $(SAR)_G$. Ciò significa che prendono un segnale in *Volt* e lo trasformano in un numero che può essere utilizzato dal microcontrollore. Le caratteristiche principali sono:

- la gamma, in inglese *range*, voltaggio, minimo e massimo, che possono accettare come valore in ingresso da convertire;
- la risoluzione, il numero di *bit* del valore numerico convertito, questo determina anche il più piccolo incremento che può venire riconosciuto.

Il funzionamento di un *ADC* di tipo *SAR* è molto semplice: il circuito parte con l'acquisizione del segnale da convertire e lo memorizza, grazie ad un circuito chiamato *sample and hold*; successivamente genera un segnale analogico a partire da un numero, il cui valore è noto (nella prima iterazione corrisponde alla metà del *range*, il che equivale ad impostare ad uno il bit più significativo e tenere a zero tutti gli altri). I due segnali vengono comparati, controllando se il valore in ingresso è maggiore o uguale all'approssimazione fatta fino a quel momento, il risultato è inserito in un registro che farà da base per l'iterazione successiva. Figura 1.3 mostra il funzionamento logico del componente, in particolare l'albero binario che porta alla soluzione finale di approssimazione.

1.4.2. Linguaggio *Rust*

Rust è un linguaggio di programmazione che gode di un sistema di tipi molto ricco, è estremamente perfomante e ha una gestione della memoria che elimina un'intera classe di errori legati a quest'ultima, che si riflettono anche sulla sicurezza. Uno degli obiettivi del progetto, era proprio quello di utilizzare *Rust* come linguaggio principale, non solo per i vantaggi che offre in termini di affidabilità, ma soprattutto per studiare i suoi limiti. Quando parlo di limiti non intendo quelli prestazionali ma, piuttosto, quelli relativi alla facilità di adozione e integrazione nei sistemi esistenti.

1.4.3. Compilatori *HighTec*

HighTec vende sotto licenza un compilatore per *Rust* certificato ISO 26262$_G$ Automotive Safety Integrity Level D (ASIL D)$_G$ [5], il che vuol

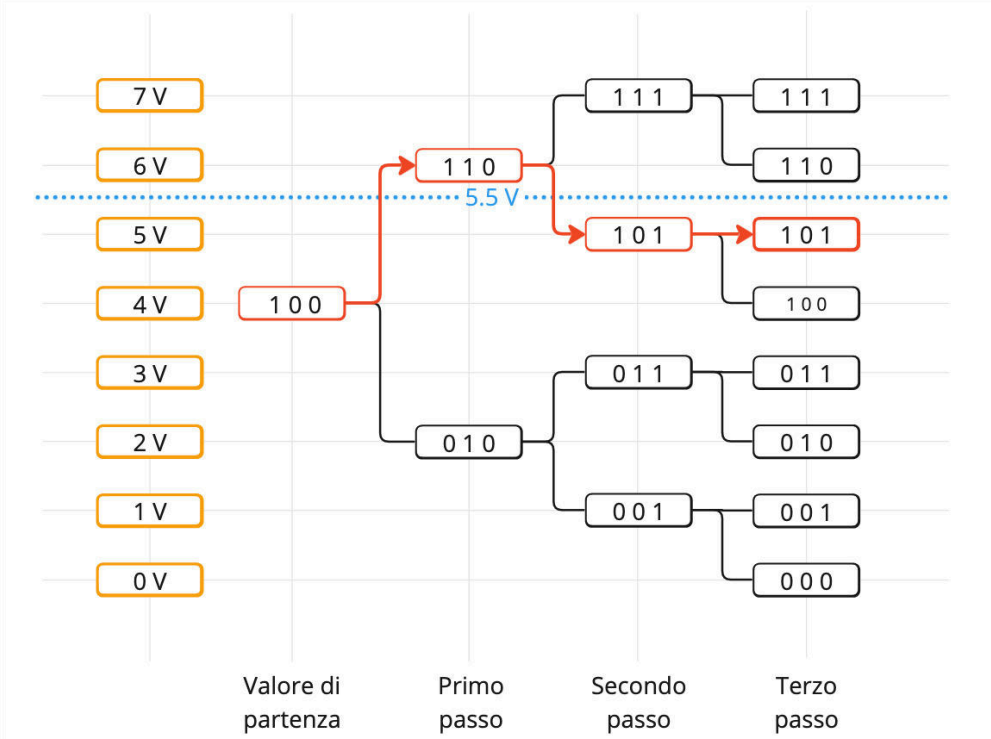


Figura 1.3: esempio di come avviene l'approssimazione digitale del valore analogico in ingresso di 5.5V, immaginando un ADC con range da 0V a 7V ed una risoluzione di 3 bit.

dire che può essere usato per compilare codice di sistemi di controllo per il settore automobilistico. Il compilatore in questione compila il codice anche per la piattaforma *Aurix* e, visti gli interessi di *Bluewind* sia per il settore, sia per il microcontrollore, lo abbiamo scelto come strumento, in modo da studiarne ed analizzarne l'utilizzo.

1.4.4. *PLS UDE*

Per scrivere il codice all'interno dell'*hardware*, e farne il *debug*, abbiamo scelto uno strumento già utilizzato in azienda anche per il linguaggio *C*, l'*Universal Debug Engine*, comunemente chiamato *UDE*. Di questo strumento, vista la possibilità di fare il *debug* nei microcontrollori *Aurix*, ne erano conosciute a fondo caratteristiche e funzionalità ma, era stato provato PoCo per lo sviluppo *Rust*. Questo lo ha reso un interessante caso di studio per vedere come, passando da un linguaggio all'altro, si sarebbe comportato, rispetto anche alle promesse fatte dalla ditta che lo produce.

1.4.5. Sistemi operativi

Siccome il sistema operativo utilizzato nei computer aziendali è *Ubuntu*, e la maggior parte degli strumenti di sviluppo funziona solo su *Windows*, utilizzavo una macchina virtuale. Essa funzionava grazie al programma *VirtualBox*, che attraverso alcune estensioni, mi permetteva di provare il codice direttamente sulla scheda di prova collegata al sistema.

1.4.6. Analisi e progettazione

Per la scrittura dei requisiti invece non ho utilizzato uno strumento specifico, ma ho usato *StarUML* per fare delle rappresentazioni grafiche legate alla progettazione del *software*:

- diagramma dei casi d'uso;
- diagramma delle classi;
- diagramma per la macchina a stati finiti;
- digramma di sequenza.

1.4.7. Altri strumenti

Nella *routine* quotidiana, come issue tracking system (ITS)$_G$, utilizzavamo *Gitlab*, unito al suo sistema di controllo di versione per il *software*. Infine abbiamo usato *Telegram* e *Zoom* come canali di comunicazione, rispettivamente per messaggi rapidi e video-conferenze.

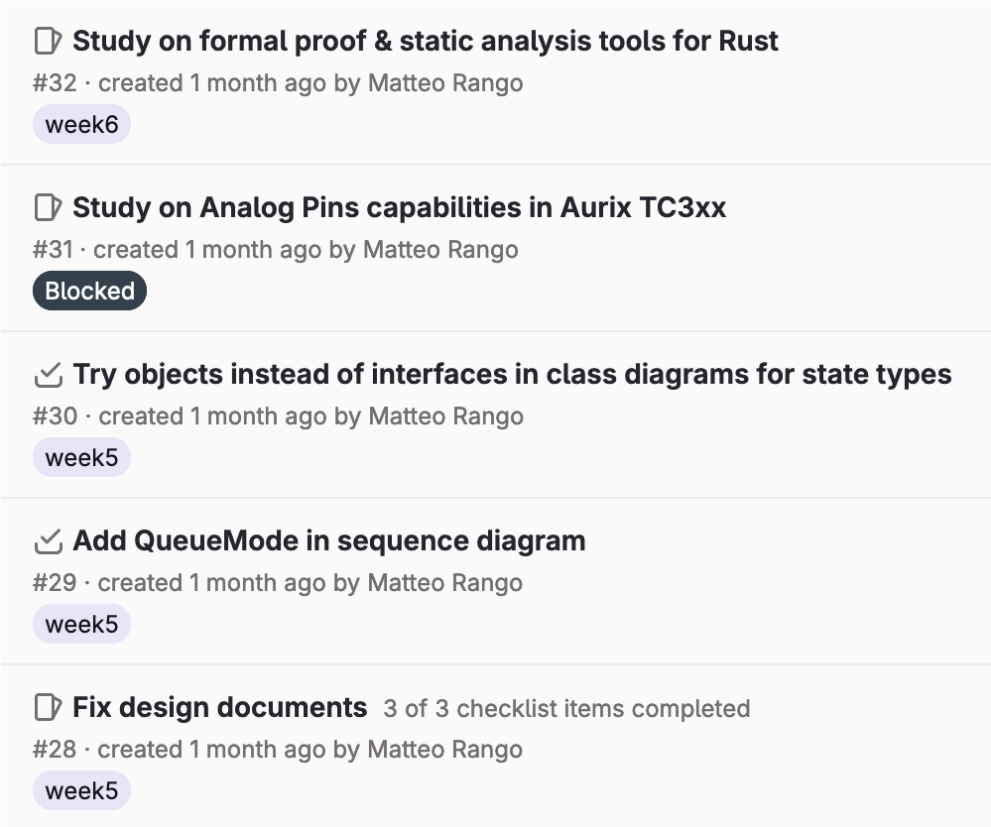


Figura 1.4: cattura a schermo di una porzione delle attività svolte, tracciate nell'*ITS*.

1.5. Rapporto con l'innovazione

Bluewind ha un rapporto speciale con l'innovazione, facendo di essa il proprio cavallo di battaglia.

L'azienda impiega risorse ed energie nello studio volto all'integrazione di nuove tecnologie, al servizio dei processi interni e dei prodotti sviluppati, per soddisfare al meglio le proprie esigenze e quelle dei clienti.

Proprio a questi è spesso difficile spiegare l'importanza di mantenersi aggiornati rispetto all'evoluzione tecnologica, che mai come in questi anni avanza velocemente. Affinché tutto questo sia possibile, l'azienda si avvale sia di risorse interne, come i dipendenti, sia di risorse esterne, come consulenti e tirocinanti universitari.

In particolar modo, i tirocini universitari che si svolgono, hanno anche come scopo quello di condurre ricerche su temi in voga, oppure sviluppare progetti utilizzando nuovi strumenti e tecnologie, per le quali, ad esempio, non si conosce l'impatto futuro perché troppo giovani.

Questo non è solo un modo di procedere che l'azienda applica, ma una vera e propria cultura aziendale, orientata alla sperimentazione e, quando possibile, all'adozione, di nuove tecnologie.

La collaborazione con partner esterni tra cui, esperti del settore, aziende leader e università permette l'integrazione di diverse prospettive e competenze tecniche all'avanguardia, creando un continuo ciclo di rinnovamento. Basti pensare alle diverse figure che collaborano con Bluewind: dalle persone che aiutano a migliorare i processi aziendali di produzione, ai consulenti per il miglioramento dell'ambiente di lavoro, o ancora alle aziende partner, con le quali avvengono innumerevoli scambi di informazioni e conoscenze. Inoltre, anche le università giocano un ruolo fondamentale, grazie ai confronti con professori universitari nei temi più svariati.

10.

Capitolo 2.

Scopo del tirocinio

In questo capitolo espongo le idee che hanno messo le basi per la formalizzazione della proposta di stage, assieme alla risposta al fine di realizzarle.

2.1. Rapporto azienda - *stage*

Bluewind sfrutta a pieno i tirocini universitari, soprattutto quelli che prevedono un piano di lavoro definito e che portano alla creazione di un progetto o allo studio mirato e consapevole di una nuova tecnologia.

Il triplice scopo dell'azienda, ha alla base mettere in discussione i processi aziendali di sviluppo di un progetto. I colleghi sono i primi con cui è possibile confrontarsi e discutere su quali possano essere le metodologie più adatte per lo specifico tirocinio. La discussione ha la finalità di trovare una strada da poter replicare in futuro, avendo già fatto un'analisi sul perché essa sia la più indicata. Oltre a ciò, il confronto con i colleghi migliora entrambe le parti in gioco, creando una visione più ampia, che spesso porta a conclusioni diverse da quelle pensate inizialmente.

Il secondo scopo è legato al progetto stesso, che sia sviluppo o ricerca. Infatti l'azienda è molto attiva quando si parla di studio e implementazione di nuove tecnologie, mettendosi in prima linea per conoscerle e provarle. Tutto ciò che può portare all'azienda un vantaggio competitivo vale la pena di essere, come minimo, conosciuto. Se si tratta di progetti di sviluppo di un prodotto, il vantaggio è ovviamente quello di poterlo

utilizzare, così come viene fornito, o usandolo come base di partenza per ulteriori sviluppi. Lo stesso ha valore anche per i progetti di ricerca.

Il terzo scopo, ma non per importanza, è sicuramente legato al fattore assunzioni. L'azienda ha la possibilità, per qualche mese, di vedere e studiare le capacità dei tirocinanti, decidendo poi se vale la pena fare loro una proposta di assunzione. Oltre a questo, gli studenti avranno già fatto, al termine del loro *stage*, un po' di formazione riguardo gli strumenti adottati da *Bluewind* e quindi l'integrazione potrà essere più veloce.

2.2. Panoramica del progetto

2.2.1. Scopo

Il codice attualmente presente nei sistemi in produzione è ancora prevalentemente scritto in *C*. Nonostante le aziende che operano nel settore automobilistico, attingano ai più moderni *standard* per la creazione di *software*, a volte non è sufficiente a garantire che esso sia sicuro e affidabile. Inoltre, vista la natura stessa del linguaggio *C*, il codice che ne deriva può risultare difficile da leggere e *mantenere*. Lo scopo del mio tirocinio era l'implementazione di un *driver* in *Rust*, in Figura 2.1 chiamato *Rust Peripheral Driver*, per la gestione della periferica *EVADC* su microcontrollori per applicazioni automotive$_{G}$, in particolare l'*Infineon Aurix TC375*.

A seguito o in parallelo alla parte implementativa avevamo pensato ad una fase di analisi su due temi rilevanti per il dominio della sicurezza funzionale$_{G}$ [6], ossia l'insieme di tutte le misure automatiche che si adottano al fine di aumentare l'affidabilità e la sicurezza del sistema:

- un'indagine generale sui vantaggi e/o svantaggi di utilizzo di *Rust* per librerie di basso livello, tenendo conto anche della sua ripida curva di apprendimento;

- un'indagine sulla possibilità, ed eventuali modalità preferibili di progettazione, del *driver* per evitare errori di configurazione della periferica in modo statico, cioè a tempo di compilazione, con la possibilità di tracciamento delle regole di configurazione rispetto ai manuali tecnici del microcontrollore.

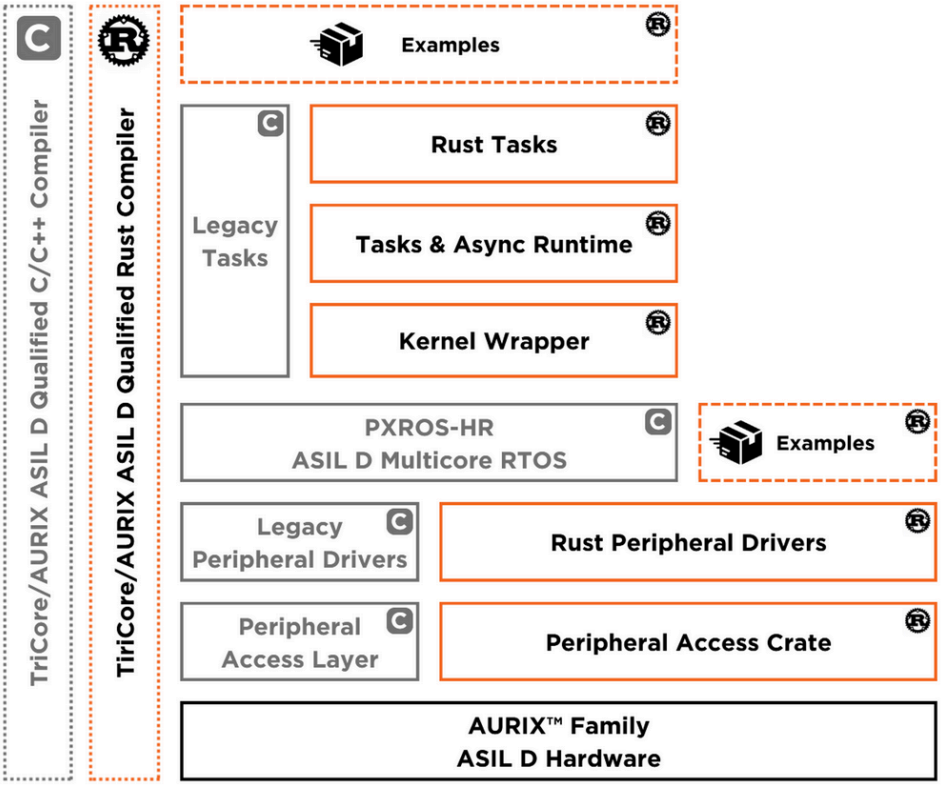


Figura 2.1: rappresentazione dello stato attuale dell'ecosistema di sviluppo *Rust* nel mondo *Aurix*, bordo arancione, paragonato al corrispondente ecosistema *C*, bordo grigio. [2]

2.2.2. Obiettivi e attese

Gli obiettivi principali erano tre:

1. L'implementazione di un *driver* in *Rust* per la gestione della periferica *EVADC* su microcontrollori della famiglia *Infineon Aurix TC3xx* per applicazioni *automotive*. Lo scopo di tale obiettivo era ovviamente avere un prodotto funzionante da poter utilizzare alla necessità. Implementare il *driver* ci avrebbe permesso anche di incrementare la bontà dell'architettura e la sua usabilità.
2. Indagine sulla possibilità, ed eventuali modalità preferibili di progettazione, del *driver* per evitare errori di configurazione della periferica in modo statico, cioè a tempo di compilazione, con la possibilità di tracciamento delle regole di configurazione rispetto ai manuali tecnici del microcontrollore. L'idea di questo obiettivo, nasce dalle precedenti esperienze nella programmazione a basso livello, dove errori di configurazione erano semplici di inserire. Volevamo creare un'interfaccia sicura dal punto di vista funzionale, pur garantendo un'architettura ricca di funzionalità.
3. Esperimento di integrazione di uno strumento di verifica formale$_{\text{G}}$ tramite il quale è possibile dichiarare e verificare proprietà logiche direttamente sul codice sorgente *Rust*, come raccomandato dallo standard *ISO 26262* per alti livelli di integrità della *sicurezza funzionale*. La verifica formale ci avrebbe permesso di controllare, a tempo di compilazione, tutte quelle proprietà *hardware* che il compilatore, attraverso l'architettura, non sarebbe stato in grado di verificare.

Converter Group	Input Channels	Converter Cluster	Common Service Req. Group	Associated Standard Reference Pins
Primary Groups				
G0	CH0 … CH7	Primary	C0	V_{AREF2}, V_{AGND2}
G1	CH0 … CH7	Primary	C1	V_{AREF2}, V_{AGND2}
G2	CH0 … CH7	Primary	C0	V_{AREF2}, V_{AGND2}
G3	CH0 … CH7	Primary	C1	V_{AREF2}, V_{AGND2}
Secondary Groups				
G8	CH0 … CH15	Secondary	C0	V_{AREF2}, V_{AGND2}
G9	CH0 … CH15	Secondary	C1	V_{AREF2}, V_{AGND2}
G10	CH0 … CH15	Secondary	C0	V_{AREF2}, V_{AGND2}
G11	CH0 … CH15	Secondary	C1	V_{AREF2}, V_{AGND2}
Fast Compare Channels				
FC0	CH0	FastCompare	C0	V_{AREF2}, V_{AGND2}
FC1	CH0	FastCompare	C1	V_{AREF2}, V_{AGND2}
FC2	CH0	FastCompare	C0	V_{AREF2}, V_{AGND2}
FC3	CH0	FastCompare	C1	V_{AREF2}, V_{AGND2}

Figura 2.2: tabella 260 del manuale utente [3] della famiglia di microcontrollori *Infineon Aurix TC37X* che rappresenta la configurazione del modulo *EVADC*.

A questi, durante il corso del progetto e in accordo con i miei tutor, ne ho aggiunti altri; all'inizio non erano stati pensati, ma sono risultati interessanti al fine di approfondire meglio i temi trattati:

- indagine sull'utilizzo di strumenti per la programmazione ed il *debug* del codice, pensati per il linguaggio *C*, usando *Rust*
- indagine e confronto su strumenti che facilitano lo sviluppo in *Rust* rispetto a quelli già conosciuti per *C*.

Nell'insieme, gli obiettivi portano alla luce come *Bluewind* stia puntando, così come molti altri, ad implementare questa tecnologia, *Rust*, all'interno dei suoi progetti. Infatti, crede che essa possa radicalmente cambiare il settore e ammodernarlo, portando vantaggi ai clienti finali, così come ai team di sviluppo.

I principali prodotti attesi, derivanti dagli obiettivi erano:

- una crate$_G$, in pratica una libreria *software Rust*, con l'implementazione del *driver*;

- documentazione architetturale e di dettaglio sul *driver* sviluppato;
- un *report* sulle indagini e gli esperimenti citati sopra.

2.2.3. Metodo di lavoro

Al fine di avere un quadro chiaro degli obiettivi, dei prodotti attesi e delle tempistiche entro le quali avrei svolto il lavoro, ho stilato assieme ai miei tutor un piano di lavoro che poi abbiamo sottoposto, per approvazione, al professore responsabile dei tirocini. Il documento specificava con particolare attenzione una scaletta di attività, divise su base settimanale, che si può riassumere nella seguente tabella:

Nr. Settimana	**Descrizione Attività**
1	Introduzione al microcontrollore. Selezione delle funzionalità da implementare.
2	Modellazione transizione tra stati.
3	Proposta di progettazione.
4	Implementazione del *driver*
5	
6	Test e confronto con librerie in *C* esistenti
7	Esperimento di integrazione di uno strumento di verifica formale
8	Collaudo finale e retrospettiva del progetto

Tabella 2.1: rappresentazione del lavoro organizzato in settimane all'interno del piano di lavoro.

Oltre al collaudo finale, ogni settimana prevedeva una giornata di analisi e retrospettiva legata al singolo periodo, per valutare il raggiungimento degli obiettivi prefissati.

Per la natura stessa del progetto e, per mettere alla prova le metodoglogie aziendali, abbiamo adottato un metodo di lavoro, costruito sulla base del *framework* Scrum$_G$. Esso prevede che il lavoro sia suddiviso in iterazioni chiamate *sprint*, piccole sezioni di tempo, solitamente della durata di una o due settimane, che prevedono:

- all'inizio dello *sprint*, una pianificazione del lavoro da svolgere, in un evento chiamato *Sprint Planning*;
- durante lo *sprint*, lo svolgimento dei compiti assegnati
- giornalmente, un aggiornamento del lavoro svolto, durante i cosiddetti *Sprint Daily*;
- al termine dello *sprint*, un aggiornamento, solitamente al cliente, del lavoro svolto, al fine di raccogliere riscontri e/o consigli, durante la *Sprint Review*
- infine, durante la *Sprint Retrospective*, una discussione su quali sono stati i problemi affrontati e quali potrebbero essere le migliorie che si possono adottare per incrementare l'efficienza ed efficacia.

Durante il corso del progetto, le iterazioni da noi stabilite erano della durata di una settimana, durante la quale svolgevamo solamente un sottoinsieme degli eventi dettati dal *framework*:

- la pianificazione è stata fatta all'inizio del percorso e ritoccata solamente a causa di alcune incogruenze tra gli impegni personali. Infatti l'idea di cosa andava fatto, e con quale priorità, era chiara fin dall'inizio, rendendo superflua la necessità di una pianificazione settimanale.
- gli aggiornamenti giornalieri invece, si sono svolti regolarmente;
- anche le revisioni di avanzamento a fine *sprint* le abbiamo svolte con regolarità, spesso unendole ad un breve confronto sul lavoro da svolgere per l'iterazione successiva.

In particolar modo gli eventi giornalieri, ci hanno permesso di rimanere sempre allineati sul lavoro in corso d'opera. Di conseguenza è stato impossibile uscire dai tempi previsti, merito anche della corretta stima temporale iniziale. Il tirocinio ha confermato come, l'uso di un *framework* specializzato, possa fare la differenza, non solo quando viene utilizzato da *team* con un discreto numero di membri, ma anche quando i *team* sono molto ridotti, come nel nostro caso.

2.2.4. Vincoli documentali

Attraverso l'uso del formato Markdown$_G$, ho sviluppato tutti i documenti tecnici che abbiamo ritenuto necessari al fine di supportare l'implementazione e i processi. La documentazione di progetto che ho scritto, l'ho sviluppata seguendo le linee guida aziendali, per due ragioni principali:

- conformare i miei documenti, almeno nella forma, a quelli della ditta, in modo tale da renderne più semplice la fruizione da parte dei colleghi;
- provare l'efficacia delle linee guida stesse attraverso la loro implementazione.

I documenti di progetto che ho scritto sono:

- *Technical Specification*: la specifica tecnica, documento nel quale ho elencato e spiegato le scelte progettuali e architetturali fatte. Esso contiene anche il tracciamento dei requisiti, ossia il loro stato rispetto al progetto.
- *Software Requirement Specification*, la specifica dei requisiti *software*, documento nel quale ho inserito una descrizione delle interfacce *hardware* e *software* esterne, i vincoli progettuali, le dipendenze e la descrizione dei requisiti di alto livello$_G$, dei quali se ne mostra una parte nella Tabella 2.2.
- *Low Level Requirements*: requisiti di basso livello$_G$, documento specifico per i requisiti di basso livello in cui, oltre a descriverli, ho creato una mappatura tra essi e quelli di alto livello.

ID	*Title*	*Type*	*Description*
FUN-01	*Initialization*	*Functional*	*Drvier shall enable the user to initialize the components with a configuration provided by the user itself.*
FUN-02	*Single conversion*	*Functional*	*Driver shall enable the user to perform a single conversion on a single input source.*
FUN-03	*Single continuous conversion*	*Functional*	*Driver shall enable the user to perform a cyclic serie of conversions on a single input source.*
…	…	…	…
FUN-05	*Multiple continuous conversions*	*Functional*	*Driver shall enable the user to perform a cyclic serie of conversions on multiple input sources, sequentially*
FUN-06	*Software conversion trigger*	*Functional*	*Driver shall enable the user to trigger a conversion, by software*
…	…	…	…

Tabella 2.2: parte della tabella dei requisiti sviluppata all'interno del documento *Software Requirements Specification (SRS)*.

2.3. Obiettivi personali

L'obiettivo primario che mi ero fissato era uscire dalla mia zona di *comfort*, andando ad esplorare un mondo che fino a quel momento avevo visto solamente a livello amatoriale. Questo avrebbe permesso di provare a me stesso che sono in grado di adattarmi velocemente a situazioni nuove, pur trovandomi, inizialmente, spaesato. Praticamente:

- imparare a leggere e capire il manuale utente di un microcontrollore, comprendendo il funzionamento dei diversi componenti e la funzione dei registri delle periferiche;
- imparare a leggere uno schema elettrico, capendo le connessioni tra i componenti esterni e quelli interni;

- imparare a scrivere *software* di basso livello, lavorando senza un sistema operativo e con strutture vicine al linguaggio macchina;

erano parte del bagaglio di conoscenze che volevo guadagnare in questa esperienza.

Il secondo obiettivo era legato all'apprendimento vero e proprio. Volevo conoscere e toccare con mano cosa significa scrivere *software* a basso livello, quali sono le sfide e i problemi che si affrontano ogni giorno. Oltre al fatto che le mie nozioni di elettronica erano molto limitate, e questa esperienza mi avrebbe permesso di ampliarle notevolmente. Anche il fatto che *Bluewind* lavori principalmente nel settore della *sicurezza funzionale*, mi ha garantito l'apprendimento di metodi e tecnologie atte allo sviluppo di *software* certificabile. Nello specifico, volevo compiere i seguenti punti:

- imparare quali strumenti e metodi si usano per scrivere codice conforme a determinati *standard*;
- ampliare le mie conoscenze di programmazione in *Rust*, in particolare capire quali fossero le strutture dati più adatte per scrivere codice di basso livello;
- arricchire il bagaglio di conoscenze legato ai metodi per lo sviluppo *software*, in ambiente professionale, principalmente per quanto concerne il metodo *agile* e il *framework Scrum*.

Terza e ultima considerazione riguarda la carriera lavorativa. Volevo raggiungere gli obiettivi citati sopra per aprirmi, nel caso mi fosse interessato, una strada verso questo settore. Perciò l'obiettivo era trovare lavoro, nell'azienda stessa o in altre, che si occupassero di sistemi di controllo e sicurezza funzionale, in settori dove i margini di fallimento sono minimi.

Capitolo 3.

Svolgimento del progetto

Nel presente capitolo descrivo quali sono state le attività e i frutti del lavoro. Nel capitolo mostrerò anche un esempio, di una sezione del progetto, di tutte le attività svolte, al fine di dare concretezza al contesto.

3.1. Attività e nuovi approcci

3.1.1. Analisi

Ho affrontato l'analisi del problema con un approccio ibrido. Nella prima fase, nonché quella di maggiore importanza, ho analizzato con cura il codice implementato dalla casa madre del microcontrollore in *C*. Farlo, mi ha permesso di capirne il funzionamento e, assieme alla lettura del manuale utente, di comprenderne a fondo le caratteristiche, specialmente quelle elettroniche del componente *EVADC*. Particolari dubbi mi sono sorti nel corso dello studio del manuale, legati proprio al lato elettronico, per il quale avevo una conoscenza basica, legata solamente alla mia esperienza da autodidatta.

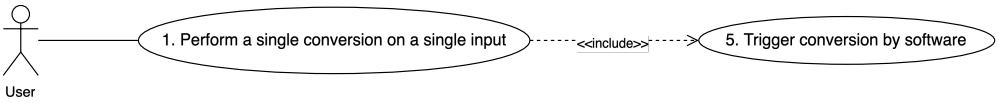


Figura 3.1: diagramma del caso d'uso numero 1, abbinato all'inclusione del numero 5.

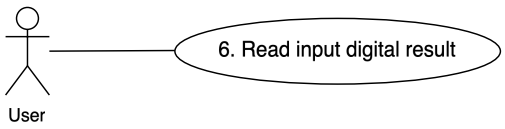


Figura 3.2: diagramma del caso d'uso numero 6.

In questo processo ho iniziato a scrivere i documenti tecnici *Software Requirements Specification* e *Low Level Requirements*. Nella specifica dei requisiti, oltre ai requisiti stessi, era presente anche il corrispondente diagramma dei casi d'uso. I requisiti raccolti, frutto dei casi d'uso, andavano a coprire un insieme di funzionalità considerato minimo per l'utente medio di una libreria di questo tipo. Per trovare l'insieme minimo, ci siamo confrontati con un collega, che aveva avuto diverse esperienze di utilizzo del modulo *EVADC* da parte dei clienti, che ci ha dato una visione sulle caratteristiche più usate. In tutto ho prodotto quindici requisiti funzionali e cinque requisiti di vincolo, per un totale di venti. Di seguito riporto una sezione della tabella dei requisiti, modificata per includere tutti i dati, che erano divisi nei vari documenti:

Identificativo	Titolo	Tipo	Descrizione	Stato	Modulo che lo soddisfa	Requisito collegato
FUN1	Inizializzazione	Funzionale	Il driver deve permettere all'utente di inizializzare i componenti con una configurazione.	Soddisfatto	`EvadcModule::<Unitialized>::initialize(...)`	Non applicabile
VIN3	Unicità hardware	Vincolo	Il driver deve garantire l'unicità di configurazione dei seguenti moduli: modulo, gruppi, canali e code.	Soddisfatto	`<Structure>Parts e Ownership`	Non applicabile
RBL1	Richiesta conversione	Funzionale (basso livello)	Il driver deve permettere all'utente di aggiungere una richiesta di conversione alla coda `i` del gruppo `x`, impostando i registri: `QxQINRi.REQCHNR = <channel id>` e `QxQINRi.RF = <refill mode>`.	Soddisfatto	`EvadcQueue::<Initialized>::add_channel(...)`	FUN1, FUN2, FUN3, FUN4, FUN5

Figura 3.3: rappresentazione dei dati raccolti, in forma tabellare, per una parte dei requisiti.

Durante l'analisi ho anche sviluppato il primo Proof of Concept $(\text{PoC})_{\text{G}}$, cioè un'implementazione del *software* basilare, priva di una documentazione dettagliata, atta a fornire riscontri rapidi in merito alle funzionalità che essa fornisce. I risultati del *PoC* sono stati:

- la conferma delle ultime nozioni acquisite, in merito all'implementazione originale del *software* di controllo della periferica *EVADC*;
- l'ottenimento di importanti informazioni tecniche di funzionamento del microcontrollore che altrimenti non avrei scoperto fino alla fase di implementazione, cioè dopo la fase progettuale. Per fare un esempio, alcune di queste informazioni erano legate alla relazione tra registri di configurazione e di sicurezza. Bisognava infatti modificare opportunamente il valore di questi ultimi per poter configurare la periferica con successo.

3.1.2. Progettazione

La progettazione mi ha portato alla creazione di diversi diagrammi, tutti originariamente scritti in notazione Unified Modeling Language (UML)$_G$, lo standard per la creazione di diagrammi tecnici, e riportati di seguito sotto forma di immagini, con le opportune considerazioni. Essi sono stati poi inseriti all'interno del documento *Technical Specification*.

Macchina a stati finiti

Il primo tra tutti è stato il diagramma a stati finiti. Gli stati rappresentano delle situazioni in cui il sistema si può trovare, ognuna con il proprio contesto, collegate le une con le altre attraverso delle regole, in forma di passaggi da uno stato all'altro. Inizialmente, pensavo di riuscire a garantire la correttezza di tutti i passaggi tra gli stati a tempo di compilazione ma, come si può vedere anche dalla Figura 3.4, questo non è stato possibile, e sono dovuto ricorrere ad una duplice gestione: sia a tempo di compilazione, sia a tempo di esecuzione. La principale differenza tra le due è che la correttezza della prima viene direttamente garantita dal compilatore, ancor prima di eseguire il programma, il che porta numerosi vantaggi in termini di affidabilità e di sicurezza. La seconda invece viene garantita solo tramite il codice, che andrà opportunamente verificato, ma per il quale non avremo mai la certezza matematica della mancanza di errori.

Per capire di avere la necessità di una gestione divisa tra compilazione e esecuzione, non mi sono state sufficienti le fasi di analisi e progettazione. A loro ho dovuto affiancare anche la costruzione di un secondo *PoC*, che comprendeva solamente la gestione degli stati, senza alcuna funzionalità pratica. Ciò mi ha permesso di concentrare i miei sforzi sulla creazione dell'architettura, piuttosto che sui problemi legati al funzionamento del microcontrollore.

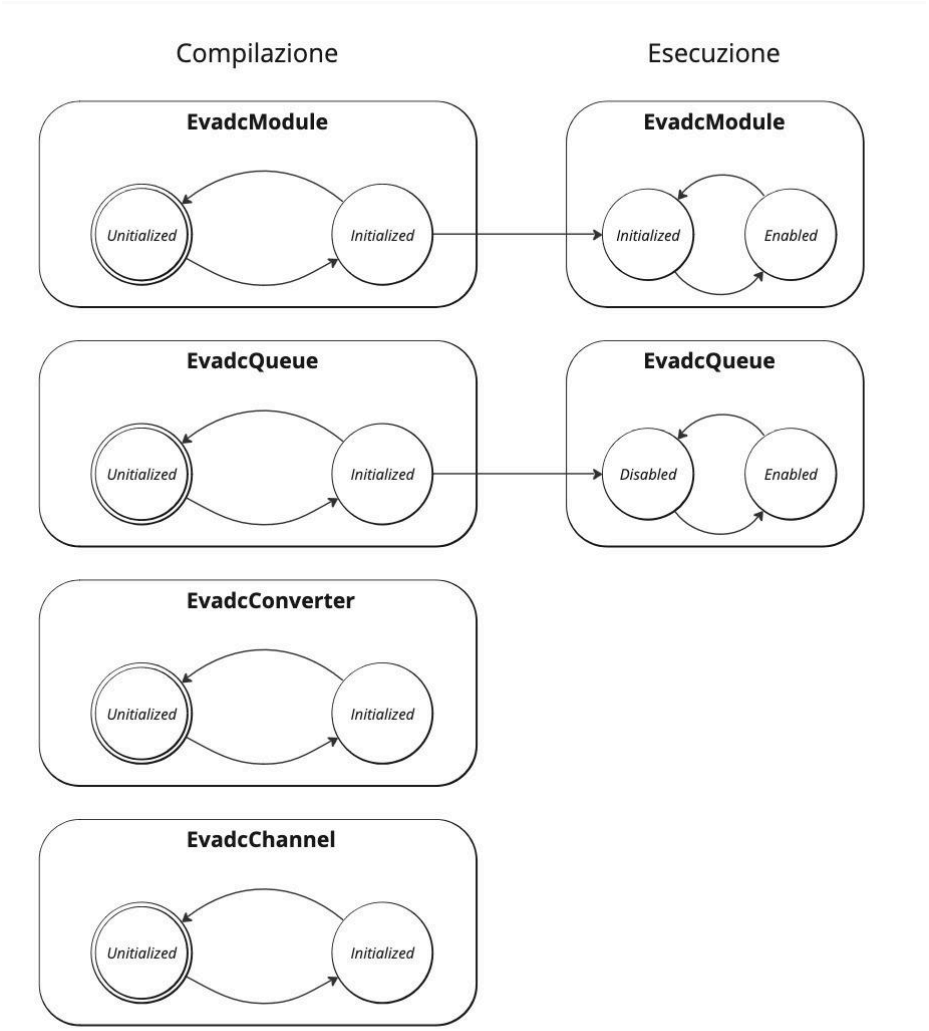


Figura 3.4: diagramma a stati finiti che rappresenta gli stati e le loro relazioni per ogni componente del sistema, in grassetto.

Diagramma delle classi

Il diagramma delle classi rappresenta la struttura architetturale e come gli elementi in essa siano legati tra di loro. Il suo scopo è anche quello di descrivere, ad alto livello, quali modelli progettuali si vogliono adottare, senza specificarne in dettaglio l'implementazione, che potrebbe cambiare da linguaggio a linguaggio.

Utilizzando la notazione *UML* più recente, e le mie conoscenze pregresse, è stato comunque difficile descrivere alcune parti dell'architettura, spiegate di seguito, a causa della scelta "obbligata" dell'uso di *Rust*. Infatti il linguaggio scelto offre un paradigma di programmazione che si discosta dai canoni classici e di conseguenza risulta più difficile da rappresentare. Dovendo scegliere, ho preferito chiarezza e comprensibilità piuttosto di correttezza della sintassi *UML*.

Il primo esempio riguarda l'uso delle tuple$_G$. In *Rust* una *tuple* è una collezione di valori, anche di diverso tipo. Ogni *tuple* è un valore a se stante e il suo tipo è formato dall'insieme dei tipi dei suoi membri:

```
// [...]

// Il tipo è (ModuleParts, ModuleMode).
struct ModuleSections(ModuleParts, ModuleMode);

impl EvadcModule<Initialized> {
    pub fn split(self) -> ModuleSections {
        return ModuleSections(
            ModuleParts::new(),
            ModuleMode::Disabled(EvadcModule {
                _marker: PhantomData,
            }),
        );
    }
}
```

Codice 3.1: esempio di rappresentazione e utilizzo di una *tuple* in *Rust*

I tipi composti, come in questo caso, non sono facilmente rappresentabili in notazione *UML* e quindi sono ricorso alla creazione di una classe apposita, avente membri chiamati con i numeri. Per definire poi *tuple* di diverso tipo, ho usato la notazione tra parentesi angolate, utilizzando lettere diverse per singolo tipo.

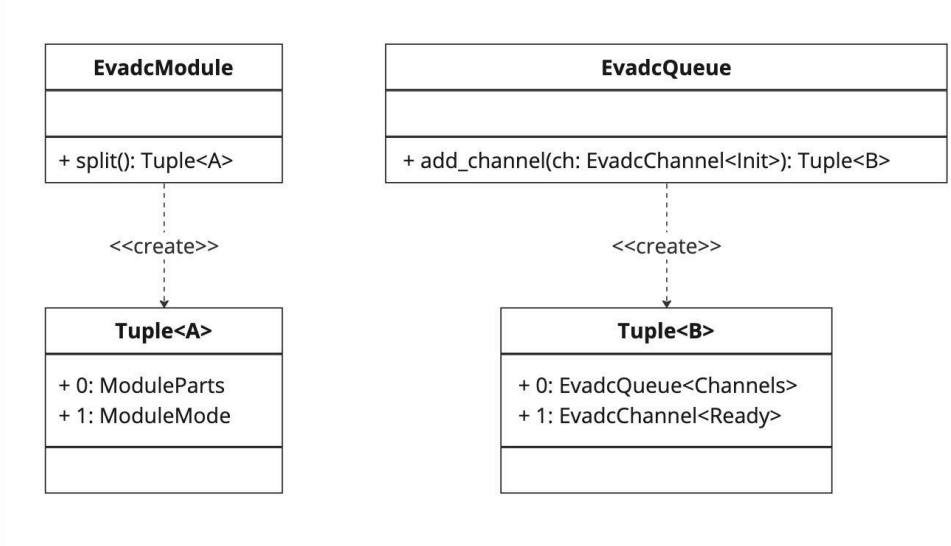


Figura 3.5: quattro classi nel diagramma omonimo che mostrano la convenzione adottata per le *tuple*.

Altro esempio in cui ho preferito rendere esplicito un concetto di *Rust* è stato per l'ownership$_G$. Questa caratteristica impone una serie di regole su come sono gestiti i valori:

1. ogni valore deve avere un proprietario;
2. ci può essere un solo proprietario per volta;
3. quando il proprietario esce dal contesto, il valore viene liberato.

Per fare un esempio pratico:

```rust
pub struct PrimaryClusterParts<G: ConverterGroup> {
    pub ch0: EvadcChannel<Unitialized, G>,
    pub ch1: EvadcChannel<Unitialized, G>,
    pub ch2: EvadcChannel<Unitialized, G>
}

// [...]

// g0_parts è di tipo PrimaryClusterParts, che contiente tre canali
let ch0 = g0_parts.ch0.initialize(&config);

// Un errore di battitura come questo, in cui proviamo a reinizializzare il ch0, ci
// da un errore di compilazione, spiegandoci che il ch0 è già stato "preso" nella
// riga sopra
let ch1 = g0_parts.ch0.initialize(&config); // ERRORE DI COMPILAZIONE
```

Codice 3.2: esempio del concetto di *ownership* in *Rust*.

Quindi i valori dei membri si potrebbero definire come dei singleton_G per ogni istanza di classe; volevo far trasparire anche dal diagramma che i membri di alcune particolari classi potessero essere unici in tutto il programma. Per esempio, ho usato la proprietà per garantire che un componente elettronico non potesse essere utilizzato due volte per fare la stessa operazione, il che avrebbe portato a errori logici difficili da scovare. Di conseguenza anche nella rappresentazione in *UML* ho dovuto creare un tipo specifico per rappresentare questa caratteristica e l'ho chiamato `InsanceSingleton<T>` dove `T` è il tipo del membro effettivo.

Ci tengo a sottolineare come questa scelta sia stata presa in favore di uno degli scopi principali del progetto: garantire a tempo di compilazione una correttezza dal punto di vista logico, rispetto al funzionamento del microcontrollore. Non sarebbe stato possibile avere la stessa garanzia utilizzando un modello *singleton* oppure lasciando il controllo a metodi e funzioni.

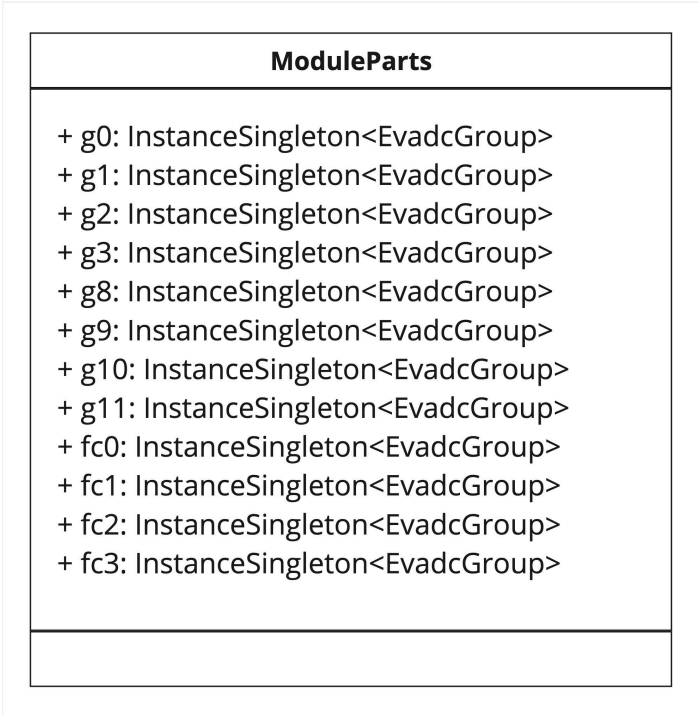


Figura 3.6: una classe del diagramma omonimo contenente membri che rappresentano il concetto di *ownership* in *Rust*.

3.1.3. Implementazione

L'implementazione è stata l'attività che è durata meno tempo; considerando il lavoro svolto a priori, rientrava nella pianificazione effettuata a inizio tirocinio.

Durante questa attività, mentre sviluppavo, uno dei miei tutor si occupava di provare ad usare la libreria *software*, per dare un riscontro immediato sull'usabilità della stessa. Il lavoro congiunto mi ha permesso di migliorarne notevolmente semplicità e comprensibilità da parte dell'utente finale, oltre che a permettermi di trovare immediatamente errori logici di programmazione.

Per fare in modo di garantire una buona usabilità, ho creato un diagramma di sequenza, che mostrava la relazione temporale tra l'uso dei diversi componenti *software* dell'applicazione. In particolare, si faceva vedere come le chiamate ai metodi e alle funzioni potevano essere usate

in un particolare caso d'uso. Di seguto riporto una sezione di tale diagramma.

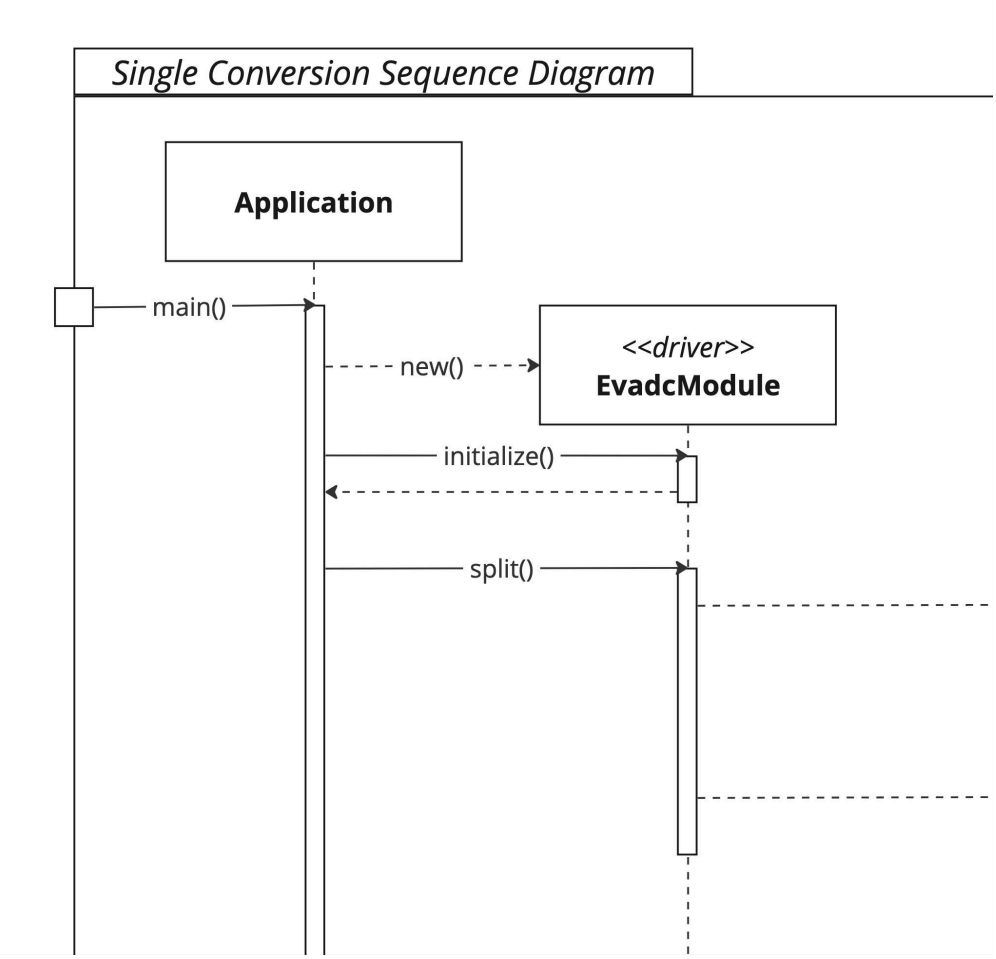


Figura 3.7: sezione iniziale del diagramma di sequenza creato durante il tirocinio.

Nonostante la creazione dei due *PoC*, durante la fase di implementazione, le sfide non sono mancate. In particolar modo voglio descrivere la più significativa, ossia quella legata al modello di progettazione chiamato typestate$_G$. Il *typestate pattern* si può collocare nei modelli progettuali comportamentali e serve a codificare le informazioni di stato, a tempo di esecuzione, di un oggetto, nel suo tipo a tempo di compilazione; proprio per questo è preferibile al *builder pattern*, dove i controlli sullo stato di avanzamento di costruzione di un oggetto sono fatti durante l'ese-

cuzione. Questo è utile soprattutto nel momento in cui si vogliono far rispettare determinati contratti, come nel nostro caso la configurazione, che doveva avvenire in un ordine specifico, assicurandosi di non saltare nessun passaggio. Questo è il caso per quasi tutti gli oggetti configurabili del *driver* che ho sviluppato.

Ci tengo a sottolineare che per noi era fondamentale riuscire a gestire la configurazione a tempo di compilazione. Infatti il rischio, se non si adotta questa pratica, è quello di introdurre errori difficilissimi da trovare e risolvere; senza considerare il rischio, nel peggiore dei casi, di rompere la scheda. Il precedente *driver*, sviluppato in *C*, implementava alcune funzionalità lasciando all'utente la piena responsabilità delle sue azioni, permettendogli quindi di creare configurazioni sbagliate. La difficoltà è stata, di conseguenza, quella di creare un'interfaccia che rimuovesse questa classe di errori, senza inficiare sul numero di funionalità offerte, obiettivo che è stato raggiunto con successo.

Durante l'implementazione mi sono anche accorto di come i microcontrollori, facenti parte della stessa famiglia, avessero caratteristiche, rispetto al modulo *EVADC*, molto simili. L'unica differenza era nel numero di *ADC* presenti e nella loro distribuzione. Così abbiamo aggiunto la possibilità di compilare il codice per due diverse schede, utilizzando un comando specifico in fase di compilazione. Il risultato di questo ulteriore vincolo di progettazione, aggiunto in corso d'opera, ci ha garantito che il *driver*, potrà essere sviluppato in unica soluzione e poi configurato in base al microcontrollore scelto. Di conseguenza il codice alla base risulta più facilmente manutenibile e privo di discrepanze tra versioni per diversi microcontrollori.

```rust
// Definisco gli stati che il modulo può assumere
pub trait ModuleState {}
pub struct Unitialized {}
pub struct Initialized {}

// [...]

// L'unica azione permessa quando, si ha un modulo non inizializzato, è inizializzarlo con una configurazione
impl EvadcModule<Unitialized> {
    pub fn initialize(self, config: &EvadcModuleConfig) -> EvadcModule<Initialized> {
        // [...]
    }
}

impl EvadcModule<Initialized> {
    pub fn split(self) -> ModuleSections {
        // [...]
    }
}

let unitialized_module: EvadcModule<Unitialized> = EvadcModule::new();

// chiamare il metodo split non è consentito su un modulo non inizializzato
let sections = unitialized_module.split(); // ERRORE DI COMPILAZIONE

// [...]

let init_module = unitialized_module.initialize(&config);
let sections = init_module.split(); // Queste operazioni rispettano il contratto
```

Codice 3.3: esempio parziale di definizione e utilizzo di un contratto con il modello *typestate* in *Rust*.

3.1.4. Verifica e validazione

La prima validazione l'abbiamo eseguita facendo corrispondere, nella tabella di tracciamento, i requisiti, con l'implementazione dell'architettura che li soddisfava. Successivamente ho compiuto una prova manuale

per ogni requisito, direttamente sulla scheda elettronica, andando a verificare che tutti i casi d'uso fossero coperti e che venissero rispettate le attese in termini di funzionalità.

Non abbiamo implementato alcuna prova automatica del *software*, principalmente a causa del tempo e della priorità che esse avevano rispetto ad altre attività. Di fatto, una prova molto interessante è avvenuta nel campo della verifica formale.

Ho provato ad integrare *Prusti*, uno strumento di verifica formale, utile per provare la correttezza di proprietà logiche a tempo di compilazione, cioè in modo statico, senza eseguire codice. In generale questi strumenti trasformano il codice, e le condizioni che si vogliono verificare, in una forma tale per cui possano essere controllate matematicamente. Tuttavia, lo scopo del suo utilizzo non era capirne il funzionamento nel dettaglio, bensì fare uno studio sulla semplicità di utilizzo, pur non sapendone i dettagli implementativi.

```
// Prusti cerca di assicurarsi che la configurazione passata in questo
// metodo abiliti il componente in questione
#[requires(config.queues[self.id].enabled)]
pub fn configure(&self, config: Configuration) {
    // [...]
}
```

Codice 3.4: esempio di funzione con una pre-condizione, scritta in *Rust* attraverso la libreria *Prusti*.

3.2. Esempio di creazione dell'architettura

Di seguito voglio fornire un esempio, completo di tutte le attività di analisi e progettazione, di una sezione dell'intero progetto. La sezione in esame è l'inizializzazione del modulo *EVADC*.

Durante la fase preliminare dell'analisi, ho affrontato i programmi esistenti, cioè gli esempi di utilizzo scritti in *C*, proposti dalla casa madre del microcontrollore. Oltre a provare il loro funzionamento direttamente

sull'*hardware*, ho stilato una lista di vincoli e operazioni necessarie, al fine di inizalizzare, in modo basico, la periferica. Questa lista è stata il punto di inizio per la scrittura dei casi d'uso e dei requisiti, ma è stata fondamentale anche al fine di imparare come il microcontrollore gestisce le periferiche tramite la memoria.

```
940    IfxEvadc_Status IfxEvadc_Adc_initModule(IfxEvadc_Adc *evadc, const IfxEvadc_Adc_Config *config)
941    {
942        IfxEvadc_Status status   = IfxEvadc_Status_noError;
943        Ifx_EVADC      *evadcSFR = config->evadc;
944        evadc->evadc = evadcSFR;
945        uint8           inputClassNum;
946
947        /* Enable EVADC kernel clock */
948        IfxEvadc_enableModule(evadcSFR);
949
950        /* configure Global input class registers */
951        for (inputClassNum = 0; inputClassNum < IFXEVADC_NUM_GLOBAL_INPUTCLASSES; inputClassNum++)
952        {
953            /* configure ADC channel  conversion mode  */
954            IfxEvadc_setGlobalConversionMode(evadcSFR, inputClassNum, config->globalInputClass[inputClassNum].conversionMode);
955        }
956
957        IfxEvadc_Adc_setGlobalConfigurations(evadcSFR, config);
958
959        return status;
960    }
```

Figura 3.8: funzione all'interno di uno degli esempi in *C* di *Infineon* dalla quale ho studiato come funziona il modulo *EVADC*.

Registers Values

- `CMS` Conversion Mode for Standard conversions:
  - `0b00` Standard conversion;
  - `0b01` Noise reduction conversion level 1, 1 additional conversion step;
  - `0b10` Noise reduction conversion level 2, 3 additional conversion steps;
  - `0b11` Noise reduction conversion level 3, 7 additional conversion steps.
- `USC` Defines the way the analog clock is generated:
  - `0b0` Synchronized mode (initial clock pulse is defined by the phase synchronizer);
  - `0b1` Unsynchronized mode (the analog clock is generated independently).

Figura 3.9: sezione del documento di analisi preliminare dove andavo a descrivere il significato dei singoli registri utili.

Basic Initialization

1. Enable Module:
    1. `psw =` get CPU watchdog password;
    2. clear CPU Endinit (`psw`);
    3. `EVADC.CLC.U = 0x00000000` ;
    4. set CPU Endinit (`psw`).
2. For every global input class `i` :
    1. `EVADC.GLOB.ICLASS[i].B.CMS = ?` .
3. Set global configuration:
    1. `EVADC.GLOBCFG.B.CPWC = 0b1` ;
    2. `EVADC.GLOBCFG.B.USC = ?` ;
    3. `EVADC.GLOBCFG.B.SUPLEV = ?` ;
    4. `EVADC.GLOBCFG.B.SUCAL = ?` .

Figura 3.10: sezione del documento di analisi preliminare che descrivi i passaggi minimi per l'inizializzazione del modulo.

Arrivati a questo punto ho scritto i casi d'uso ed i relativi requisiti. Queste attività hanno richiesto attenzione particolare, in quanto non era mai stato fatto un lavoro di questo tipo in azienda, ossia con questa granularità, per quanto riguarda progetti legati a dei *driver*. Oltre alla produzione di documenti per il progetto di tirocinio stesso, abbiamo anche aggiornato la documentazione aziendale di conseguenza. Al fine di validare la mia comprensione di funzionamento della periferica, ho creato un *PoC* in *Rust* che andava ad inizializzare il modulo, verificando, grazie al *software UDE*, che lo stato dei registri, dopo l'inizializzazione, fosse lo stesso, sia con gli esempi in *C*, sia con l'*PoC* in *Rust*.

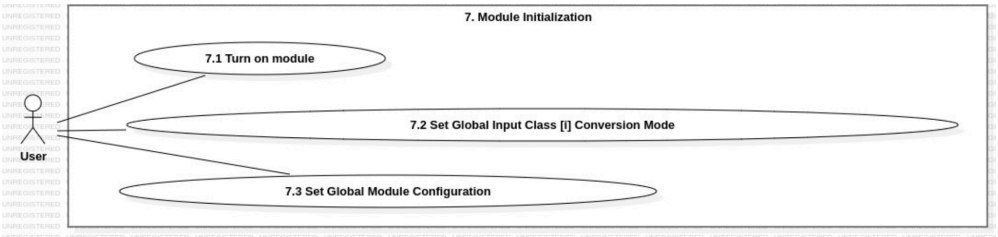


Figura 3.11: diagramma dei casi d'uso specifico riguardo l'inizializzazione del modulo *EVADC*.

Functional

ID	Title	Type	Description
1	Initialization	Functional	Driver shall enable the user to initialize the components with the a configuration.

Figura 3.12: requisito funzionale specifico riguardo l'inizializzazione del modulo *EVADC*.

Design Constraints

ID	Title	Description	Rationale
1	EVADC initialization	Driver shall enforce the initialization of the EVADC module itself before anything else.	Doing the other way around could leave the system in a inconsistent state.

Figura 3.13: requisito di vincolo specifico riguardo l'inizializzazione del modulo *EVADC*.

4. Driver shall enable the user to setup the EVADC module by setting the following registers:

- For each input class `i`:
  - `GLOBICLASSi.CMS = <conversion mode>`
- `GLOBCFG.CPWC = 1`
- `GLOBCFG.USC = <clock generation mode>`
- `GLOBCFG.SUPLEV = <supply voltage level>`
- `GLOBCFG.SUCAL = <sturtup calibration>`

> In order to do so, module clock must be enabled, setting `CLC.DISR = 0`. Without clock enabled EVADC module can't perform any operations, aside from the configuration of its submodules.

Figura 3.14: requisito funzionale di basso livello riguardo l'inizializzazione del modulo *EVADC*.

La progettazione, come ho già detto sopra, è stata l'attività più difficile in termini di ragionamento. Partendo dai requisiti mi sono cimentato nella creazione di un'architettura adatta. La prima attività, e probabilmente la più utile, è stata definire gli stati e le transizioni tra di essi, nei quali il *software* sarebbe potuto essere. Per fare ciò ho sviluppato un secondo *PoC*, senza funzionalità, con il solo scopo di provare un modello di progettazione che secondo me era il più adatto per questo caso, il *typestate*.

```rust
impl Default for EvadcModule<Composed, Unit> {
    fn default() -> Self {
        Self::new()
    }
}
impl EvadcModule<Decomposed, Init> {
    pub fn disable(self) -> EvadcModule<Decomposed, Unit> {
        EvadcModule {
            state: self.state,
            _state: PhantomData,
        }
    }
}
impl EvadcModule<Decomposed, Unit> {
    pub fn enable(self) -> EvadcModule<Decomposed, Init> {
        EvadcModule {
            state: self.state,
            _state: PhantomData,
        }
    }
}
```

Figura 3.15: sezione di codice del secondo *PoC* per provare il funzionamento degli stati.

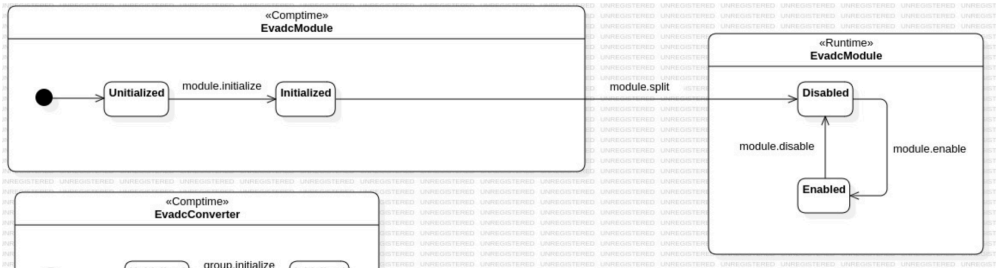


Figura 3.16: diagramma degli stati per il modulo *EVADC*, diviso per stati a tempo di compilazione e tempo di esecuzione.

Dopo questo passaggio ho iniziato, grazie a tutte le informazioni ottenute, a scrivere i documenti di specifica tecnica, nei quali erano presenti anche due diagrammi, che hanno aggiunto valore alla progettazione stessa. Essi sono stati in ordine, il diagramma delle classi e quello di sequenza. Il primo è servito per costruire l'architettura del sistema, in questo caso quella del modulo *EVADC* stesso. Secondo i casi d'uso ed

i requisiti, il compito di questa sezione del sistema era quello di fornire un'interfaccia per:

1. avere la possibilità di creare una configurazione per la periferica;
2. avere la possibilità di applicare la configurazione creata;
3. avere la possibilità di ottenere i sotto-componenti collegati alla periferica, in modo controllato;
4. abilitare o disabilitare la periferica.

Il primo ed il secondo punto li ho risolti creando un oggetto di configurazione da passare al metodo omonimo. Da notare è il fatto che per fare questo è stato usato il *typestate pattern*, quindi già a tempo di compilazione ci si può assicurare che, se si vuole usare il modulo, bisogna per forza aver creato e applicato una configurazione corretta. Per l'abilitazione della perifierica ho usato invece un normale *state pattern*, allo scopo di gestire in modo controllato le transizioni. Infine, per rispndere al requisito di vincolo, che chiedeva l'inizializzazione del modulo generico prima di ogni altro, ho fatto in modo che si potessero ottenere i sotto-componenti collegati solo dopo, aver applicato la configurazione e lanciato il metodo per bloccarla definitivamente. Inoltre, grazie al concetto di *ownership* di *Rust*, ho fatto in modo che i sotto-componenti potessero essere presenti, all' interno del sistema, una volta sola, così da rispettare le proprietà fisiche dell'*hardware*.

Per concludere, voglio citare anche un altro modello di progettazione utilizzato, non solo in questa parte, ma in tutto il *driver*, che è l'*adapter*. Si tratta di un modello che aveva come scopo quello di inserirsi tra il mio sistema e una libreria esterna, dove erano definiti tutti i registri della scheda elettronica. Essendo questa libreria in rapida evoluzione, ho preferito inserire un "cuscinetto" che la andasse ad astrarre, così da rendere il codice più mantenibile.

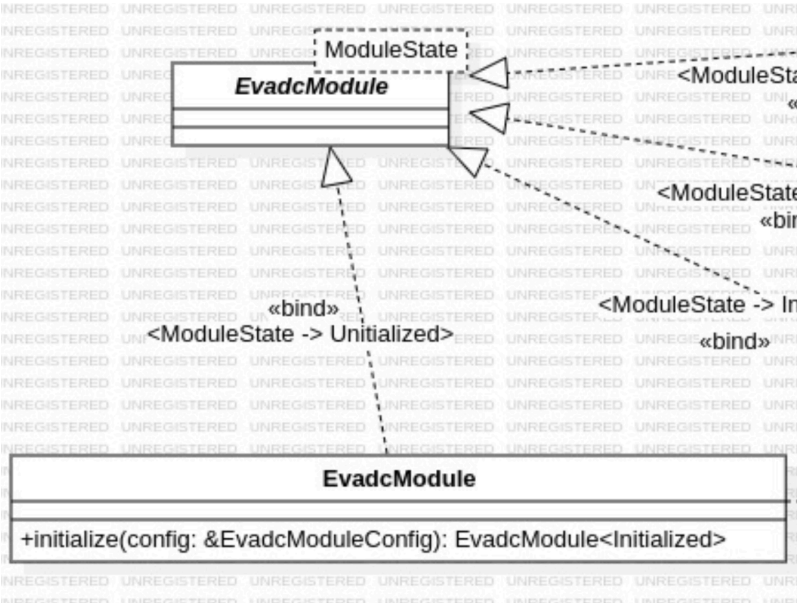


Figura 3.17: sezione del diagramma delle classi che rappresenta il modo di applicare una configurazione al modulo *EVADC*.

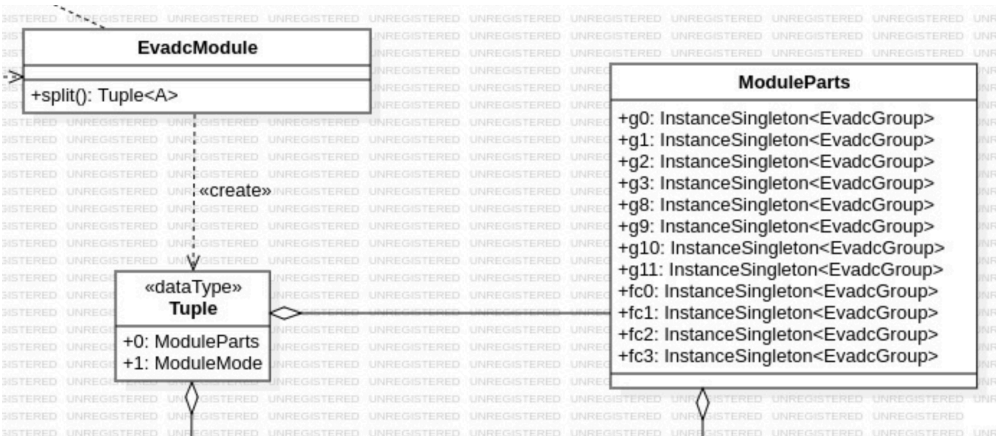


Figura 3.18: sezione del diagramma delle classi che rappresenta il modo bloccare la configurazione del modulo *EVADC* e la classe che fornisce i sotto-componenti.

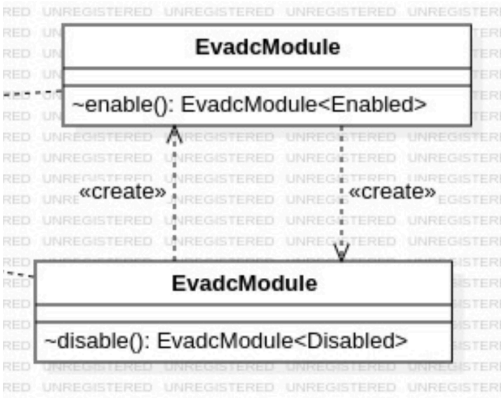


Figura 3.19: sezione del diagramma delle classi che rappresenta i metodi per il cambio di stati dinamico del modulo *EVADC*.

Finita una prima versione dell'architettura, ho creato un diagramma di sequenza, volto a capire quali fossero effettivamente i metodi da eseguire al fine di inizializzare il modulo. Il diagramma in questione si è poi rivelato fondamentale anche come guida per capire il funzionamento di tutto il sistema da parte di un potenziale utilizzatore.

L'ultima attività è stata l'implementazione, dato che, come descritto sopra, le prove di funzionamento le avevo fatte attraverso un *PoC* apposito. Ci tengo a sottolienare che tutte le attività sono state svolte in modo incrementale e non a cascata. La conseguenza è che, queste ultime, sono state portate a termine in parallelo, con continue modifiche e miglioramenti, pur avendole descritte in maniera sequenziale.

```
impl EvadcModule<Unitialized> {
    pub fn initialize(self, config: &EvadcModuleConfig) -> EvadcModule<Initialized> {
        let evadc = pac::EVADC;

        Self::enable_clock();

        for (i, gic) in config.global_input_classes.iter().enumerate() {
            unsafe {
                evadc.glob().globiclassi()[i]
                    .modify(|r| r.cms().set(gic.conversion_mode_standard_conversions.into()));
            };
        }

        unsafe {
            // Modify atomic's purpose is to set cpwc to 0x1 to enable modifications for suplev and
            // sucal registers
            evadc.globcfg().modify_atomic(|r| {
                r.cpwc()
                    .set(true)
                    .usc()
                    .set(config.clock_generation_mode.into())
                    .suplev()
                    .set(config.supply_voltage_level.into())
                    .sucal()
                    .set(config.startup_calibration.into())
            });
        };
```

Figura 3.20: sezione di codice del *driver* che scrive sui registri del microcontrollore una configurazione del modulo *EVADC*.

```
fn main() -> ! {
    //Module Initialization
    let evadc_config = EvadcModuleConfig {
        ..Default::default()
    };
    let evadc = EvadcModule::new().initialize(&evadc_config);
    let (evadc_parts, mut evadc) = evadc.split();

    evadc.enable();
```

Figura 3.21: sezione di codice del *driver* che, in ordine, genera una configurazione di *default* per modulo *EVADC*, la applica, la rende permanente e accende la periferica.

3.3. Risultati ottenuti

Qualitativi

I risultati qualitativi più rilevanti sono sicuramente nell'ambito della sicurezza funzionale, infatti moltissimi degli errori legati alla memoria di *C*, sono stati evitati grazie al semplice uso di *Rust*. Il risultato rafforza lo studio effettuato da Shea Newton [7], riguardo a *MISRA*, uno *standard* per il linguaggio *C* che prevede regole rigide riguardo alla scrittura

del codice, dove ha evidenziato come solo trentacinque delle centoquarantacinque regole *MISRA* analizzate si applicassero anche a *Rust*. La conseguenza rilevante è la maggior difficoltà nell'introdurre determinate classi di errori, provato dal fatto che, durante i *test* manuali, non siamo riusciti a mettere la scheda in uno stato di fallimento, attraverso il *driver* creato.

Un risultato che ha apportato un miglioramento al processo di analisi è stato in merito alla scrittura dei requisiti. Partendo da una base comune abbiamo introdotto, nella documentazione esistente, nuovi punti di vista e diversi approcci per affrontare requisiti di un prodotto che non verrà mai utilizzato da solo, ma solamente come supporto alle operazioni di sistemi più grandi.

Lo studio che ho condotto ha svelato, inoltre, come l'ecosistema *Rust* non sia ancora del tutto pronto ad essere integrato nei sistemi esistenti in produzione. Uno dei problemi principali sono le versioni che vengono rilasciate: per accedere alle ultime funzionalità del linguaggio bisogna usare versioni recenti, ma molti strumenti sono stati costruiti per versioni più vecchie e non sono più stati aggiornati. Rilasciando così tante versioni del linugaggio in PoCo tempo, visto che si tratta di una tecnologia in rapida crescita, è difficile per gli sviluppatori tenere aggiornate le librerie *software*. Anche durante il mio tirocinio abbiamo dovuto investire molto tempo nello scovare errori legati a questa problematica, spesso trovando soluzioni temporanee.

Altro risultato qualitativo riguarda tutti i documenti prodotti, che potranno essere visionati dai colleghi, come fonte di informazioni, sui temi trattati.

Quantitativi

I prodotti creati sono i seguenti:

- una libreria *software* contenente il codice per l'uso del modulo *EVADC* per i microcontrollori *Infineon TC37X* e *TC39X* per un totale di millesettecentoventisette (1727) righe di codice;
- analisi preliminare sulla configurazione minima necessaria per il funzionamento del modulo *EVADC*, a supporto dell'implementazione;
- diagramma delle classi, di sequenza e degli stati, utili alla descrizione dell'architettura;
- diagramma dei casi d'uso, utile alla derivazione dei requisiti;
- documento per i requisiti di basso livello, utili all'implementazione del *software*;
- un report sul caso specifico del *pattern* che usa i tipi per creare una macchina a stati finiti;
- un report sull'uso dell'*UDE* con il linguaggio *Rust*;
- un report sull'uso di *Prusti* come strumento di verifica formale;
- un documento di specifica dei requisiti, a supporto dell'implementazione;
- un documento di specifica tecnica, dove spiegare le scelte progettuali fatte;
- lucidi di presentazione del lavoro svolto per spiegazione ai colleghi.

Il totale ammonta a millecinquecento (1500) righe di documentazione divise in dieci documenti differenti. Altro risultato quantitativo degno di nota è il tempo per portare a compimento il progetto, che nel complesso ha richiesto otto settimane, perfettamente in linea con quanto programmato e disponibile.

Capitolo 4.

Retrospettiva finale

4.1. Grado di soddisfacimento

L'azienda si è rivelata entusiasta del lavoro svolto, grazie al quale si è potuta sperimentare in un ambito in forte crescita. In particolare, essa userà i prodotti di questo tirocinio, come base per future ricerche e approfondimenti, ampliando così le sue prospettive nel settore della sicurezza funzionale. In particolare il mio lavoro sulla verifica formale del *software*, sarà il punto di partenza per uno dei prossimi tirocini in avviamento. Inoltre, quanto ho sviluppato, può essere facilmente replicato e utlizzato in diversi progetti, specialmente allo scopo di creazione di prototipi. Lo studio, di come *Rust* potrà continuare a migliorare la sicurezza e l'esperienza durante lo sviluppo, ha confermato le nostre aspettative iniziali, grazie anche a tirocini effettuati in precedenza. Non mancano i dubbi sulla sua maturità e su quella delle librerie collegate, che è uno dei motivi che fa desistere l'azienda dall'utilizzarlo in produzione.

L'unico argomento che, a causa del tempo e delle priorità, non siamo riusciti ad affrontare, è stato l'implementazione di *test* automatici per la verifica del *driver*. L'obiettivo non era comunque richiesto, sarebbe stato un'aggiunta interessante, qualora ci fosse stato il tempo. I requisiti trovati, per implementare le funzionalità minime, nonché le più richieste, sono stati soddisfatti al 100%, come si può vedere dalla tabella di tracciamento riportata sotto, in cui si vede che tutti sono stati soddisfatti.

La qualità del lavoro che ho portato a termine, è risultata di gradimento all'azienda, in termini di processi, documentazione e qualità del codice.

ID	Satisfaction	Source
FUN1	Fulfilled	`EvadcModule::<Uninitialized>::initialize()`

FUN2	Fulfilled	`EvadcConverter::::split()`
		`EvadcQueue::add_channel()`
		`EvadcQueue::::start()`

FUN3	Fulfilled	`EvadcConverter::::split()`
		`EvadcQueue::add_channel()`
		`EvadcQueue::::start()`

FUN4	Fulfilled	`EvadcConverter::::split()`
		`EvadcQueue::add_channel()`
		`EvadcQueue::::start()`

FUN5	Fulfilled	`EvadcConverter::::split()`
		`EvadcQueue::add_channel()`
		`EvadcQueue::::start()`

FUN6	Fulfilled	`EvadcQueue::<Stopped>::start()`
FUN7	Fulfilled	`EvadcQueue::<Started>::stop()`
FUN8	Fulfilled	`EvadcChannel::<Ready>::get_result()`
DC1	Fulfilled	`EvadcModule::<Initialized>::split()`
DC2	Fulfilled	`EvadcConverter::<Initialized>::split()`
DC3	Fulfilled	`<Structure>Parts` & Ownership
DC4	Fulfilled	`EvadcQueue::::start()` `EvadcQueue::::stop()`
DC5	Fulfilled	Macros

Figura 4.1: tabella di tracciamento dei requisiti del docuemnto di specifica tecnica.

Il tirocinio ha superato di gran lunga le mie aspettative iniziali. L'ambiente stimolante ed energico mi ha consentito di svolgerlo al meglio e di portare a compimento tutti gli obiettivi che mi ero prefissato. A partire da quelli che richiedevano abilità meno pratiche: migliorare il modo di approcciare le sfide, arricchire le mie conoscenze in ambito dell'ingegneria del *software*; finendo con quelli più tangibili e verificabili:

- ho acquisito le conoscenze necessarie per leggere uno schema elettrico e capirne le connessioni;
- ho imparato a leggere il manuale utente e, più in generale, i manuali tecnici di un microcontrollore, capendone la struttura e la spiegazione dei singoli registri delle periferiche;
- ho ampliato le mie conoscenze del linguaggio *Rust*, in particolare le caratteristiche che si applicano al contesto dei sistemi di controllo;
- ho capito come si scrivono programmi che aderiscono a *standard* e regole che li rendono certificabili, come l'*ISO 26262*.

Un punto di miglioramento, avendo avuto più tempo, sarebbe migliorare la documentazione, in termini di estensione ed esaustività. Ad esempio, avrei potuto scrivere un documento di specifica di *test* e un conseguente *report* di copertura degli stessi.

4.2. Delta tra competenze pregresse e richieste

Prima di iniziare lo *stage*, ero consapevole della grande mole di competenze che avrei dovuto acquisire, a causa principalmente del mio percorso di studi, che tratta solo minimamente l'ambito *hardware* ed elettronico. L'unico corso rilevante in merito trattava l'architettura degli elaboratori, da cui l'omonimo nome del corso, in cui si parlava anche di microcontrollori. Al contrario, è stato molto semplice adattarsi all'ambiente di lavoro in termini di processi adottati, in quanto vengono ampliamente studiati, e soprattutto provati sul campo, in corsi come "Ingegneria del *software*" e "Metodi e tecnologie per lo sviluppo *soft-*

ware". In aggiunta, diversi corsi prevedono un progetto, di gruppo, in cui il coordinamento tra tutti i membri è fondamentale per la sua buona riuscita. Di conseguenza, la capacità degli individui di lavorare in *team* aumenta, apportando benefici anche nel mondo lavorativo.

In generale ero consapevole di dover ampliare le mie conoscenze ma, quelle che già avevo guadagnato all'università, mi hanno permesso, non solo di superare le difficoltà tecniche, ma anche di avere una solida base per riuscire ad analizzare e risolvere problemi complessi, in maniera metodica e scientifica.

Glossario

ISO 26262: *standard* internazionale per la sicurezza funzionale dei sistemi elettronici installati nei veicoli da strada. 6.

***ADC* – analog-to-digital converter**: componente elettronico in grado di effettuare conversioni da segnali analogici a digitali. 5.

Agile: insieme di metodi e principi che promuovono lo sviluppo di prodotti attraverso: piccoli gruppi di lavoro autogestiti, un approccio iterativo e incrementale, il coinvolgimento diretto del cliente finale ed una propensione al cambiamento. 1.

***ASIL D* – Automotive Safety Integrity Level D**: schema di classificazione del rischio più alto nella scala definita dallo standard *ISO 26262*. 6.

automotive: settore automobilistico. 12.

crate: nel gergo *Rust* si tratta di una libreria *software* che racchiude delle funzionalità specifiche. 15.

***EVADC* – Enhanced Versatile Analog-to-Digital Converter**: modulo presente all'interno di alcuni microcontrollori *Infineon* che controlla diversi *ADC* di tipo *SAR*, offrendo capacità di sincronizzazione e parallelismo. 5.

framework: termine generico che indica una struttura di supporto o una serie di metodologie che hanno un determinato scopo comune. 4.

requisiti di alto livello: requisiti che descrivono un comportamento o un'azione che il sistema deve svolgere, senza specificare come essa si compie. 18.

***ITS* – issue tracking system**: strumento di gestione di progetto che permette di tracciare il lavoro svolto. 8.

requisiti di basso livello: requisiti che hanno un grande livello di dettaglio e vanno a specificare le singole operazioni da svolgere sui registri del microcontrollore. 18.

Markdown: linguaggio di rappresentazione testuale facilmente convertibile in *HTML*. 18.

ownership: serie di regole definite nel linguaggio *Rust* in merito alla gestione della memoria. 26.

***PO* – product owner**: persona a capo di un progetto che si occupa di assicurarsi che esso venga svolto nei modi e nei tempi previsti. Si occupa anche di interfacciarsi con il cliente. 2.

***PoC* – Proof of Concept**: versione minimalista di un prodotto, spesso usato per verificare fattibilità ed efficacia ed ottenere un riscontro rapido. Solitamente è sviluppato in tempi brevi e per questo non implementa tutte le funzionalità. 22.

***R&D* – Research and Development**: ricerca e sviluppo. 2.

***SAR* – successive approximation register**: tipologia di *ADC* che funziona attraverso l'approsimazione iterativa del valore analogico. Esso effettua una ricerca binaria di tutti i livelli di precisione che il componente stesso offre. 5.

Scrum: particolare sistema di supporto per gestione di progetti complessi, fondato sui principi *agile*. 16.

sicurezza funzionale: sicurezza che in un sistema, solitamente elettronico, viene attuata in modo automatico, riducendo quindi il livello di rischio. 12.

singleton: modello di progettazione *software* che prevede l'istanziazione univoca di una classe. 27.

typestate: modello di progettazione *software* che prevede il trasferimento delle informazioni di stato di un oggetto, dall'esecuzione alla compilazione. 29.

tuple: struttura dati di *Rust* che può contenere più valori di uguale o diverso tipo. 25.

***UML* – Unified Modeling Language**: notazione standard per la creazione di diversi diagrammi. 23.

verifica formale: una prova del *software* che si basa su principi formali matematici di correttezza. 14.

Bibliografia

[1] Infineon Technologies, «Pagina ufficiale del kit di sviluppo Infineon TC375». [Online]. Disponibile su: https://www.infineon.com/cms/en/product/evaluation-boards/kit_a2g_tc375_lite/

[2] HighTec EDV-Systeme GmbH, «HighTec Rust Development Platform». [Online]. Disponibile su: https://hightec-rt.com/products/rust-development-platform

[3] Infineon Technologies, «Manuale utente specifico per la famiglia di microcontrollori Infineon Aurix TC37x». [Online]. Disponibile su: https://www.infineon.com/dgdl/Infineon-AURIX_TC37x-UserManual-v02_00-EN.pdf?fileId=5546d4627506bb32017530759f185ed8

[4] Robert C. Martin, Kent Beck, Martin Fowler, e altri, «Manifesto for Agile Software Development». [Online]. Disponibile su: http://agilemanifesto.org/

[5] International Organization for Standardization (ISO), «ISO 26262: Road vehicles - Functional safety». [Online]. Disponibile su: https://www.iso.org/standard/68383.html

[6] International Electrotechnical Commission (IEC), «IEC 61508: Sicurezza funzionale dei sistemi elettrici/elettronici/programmabili per la sicurezza». [Online]. Disponibile su: https://www.iec.ch/functional-safety

[7] Shea Newton (PolySync Technologies), «MISRA Rust». [Online]. Disponibile su: https://github.com/PolySync/misra-rust/tree/master?tab=readme-ov-file