UNIVERSITÀ POLITECNICA DELLE MARCHE

FACOLTA' DI INGEGNERIA
Corso di Laurea in Ingegneria Informatica e dell'Automazione

DESIGN ED IMPLEMENTAZIONE DI UN DATA LOGGER REAL-TIME CONFIGURABILE A 300 CANALI/ms

DESIGN AND IMPLEMENTATION OF A 300 CHANNEL/ms CONFIGURABLE REAL-TIME DATA LOGGER

Relatore:
Chiar.mo Prof.
SIMONE FIORI

Presentata da:
FEDERICO PRETINI

Correlatore:
Dott. magistrale
EMILIANO MAGINI

Sessione Laurea Luglio
Anno Accademico 2022/23

Alla mia famiglia
e ai miei nonni

Indice

1	Introduzione a TwinCAT Beckhoff	6
1.1	TwinCAT real-time engine	7
1.2	Scheduler TwinCAT	8
1.3	Linguaggi di programmazione	11
2	Introduzione all'hardware Beckhoff	13
3	Sintesi dell'approccio	13
4	Analisi dei Requisiti	16
4.1	Requisiti funzionali	16
4.2	Requisiti non funzionali	17
5	Progettazione iniziale del software	18
5.1	Descrizione dell'architettura generale del data-logger	18
5.2	Descrizione della struttura generale del progetto TwinCAT	19
5.3	Sistemi di buffering valutati	24
5.4	Scelta del sistema di buffering	26
5.5	Sistema di logging	27
6	Progettazione ed implementazione del primo data logger	28
6.1	Implementazione logger manager	28
6.2	Implementazione buffer	29
6.3	Implementazione logger	31
6.4	Analisi delle prestazioni	36
7	Ottimizzazione del software	38
7.1	Analisi dei limiti e delle inefficienze del primo logger	38
7.2	Progettazione ed implementazione del secondo data-logger	39
7.3	Progettazione ed implementazione del terzo data-logger	41
7.4	Progettazione ed implementazione del quarto data-logger	44
7.5	Modifiche software	44
7.6	Progettazione ed implementazione del quinto data-logger	49
8	Validazione e test	55
9	VISU	56
9.1	Realizzazione VISU	56
9.2	Collegamento GVL con VISU	59
10	Conclusioni ed eventuali sviluppi futuri	62

Introduzione

Nell'era moderna l'importanza dell'elaborazione di segnali digitali riveste ormai un ruolo cruciale in molti campi, dalle telecomunicazioni fino ad arrivare alla medicina. Questo processo parte però dalla raccolta dei dati, nella così detta fase di campionamento dei segnali.
Essa permette di convertire segnali continui nel dominio del tempo in sequenze discrete di valori noti come campioni.
Attraverso questa tecnica, i segnali analogici vengono misurati a intervalli regolari nel tempo, catturando i valori in un formato digitale che può essere elaborato, trasmesso e archiviato in modo efficiente. Ad esempio, nella fase di test di componenti per il settore automobilistico, come il motore, le elettrovalvole e le altre componentistiche meccaniche è fondamentale acquisire e registrare i segnali di risposta che forniscono informazioni preziose sul funzionamento delle parti e consentono di valutare le prestazioni e l'affidabilità dei componenti stessi.
L'efficienza del campionamento dipende sia dall'hardware utilizzato per acquisire i segnali che dal software per la gestione del processo di campionamento e archiviazione dei dati.
La seguente tesi tratterà del progetto svolto presso il gruppo Loccioni come tirocinio conclusivo del percorso triennale, al suo interno verranno mostrate le procedure per creare un software che consenta al PLC normalmente utilizzato per la gestione e la movimentazione dei banchi test, di potersi occupare anche della parte di campionamento dei segnali e del relativo salvataggio dati su file, in questo caso con estensione csv (comma separated value). Inoltre sarà possibile configurare, agendo opportunamente sull'interfaccia grafica, alcuni parametri come:

- L'intervallo di campionamento che può arrivare ad un minimo di 1 millisecondo;
- La dimensione massima espressa in MegaByte del file, superata la quale sarà aperto un nuovo file csv per continuare la scrittura;
- Il tempo massimo di scrittura su file, superato il quale sarà aperto un nuovo file csv per continuare la scrittura;
- L'header personalizzato da associare ad ogni colonna del file, ovvero dare ad ogni colonna un nome che potrebbe, ad esempio, corrispondere al nome del sensore che ha generato tutto il set di dati contenuto nelle righe sottostanti alla medesima colonna;
- Il nome del file creato e il relativo path in cui è possibile specificare la destinazione di salvataggio;
- Il numero di segnali da campionare contemporaneamente, ovvero il numero di sensori di cui si desidera campionare il relativo segnale.

Come hardware è stato fornito dall'azienda un PLC Beckhoff abbinato al software di sviluppo TwinCat.

Le motivazioni che hanno spinto l'impresa a realizzare questa nuova soluzione sono molteplici:

- L'impossibilità, utilizzando le componenti software di base, di poter campionare contemporaneamnte più di 10 segnali;
- L'impossiblità di poter segmentare il relativo file con estensione .csv per dimensione o per tempo;
- La possibilità di poter eliminare hardware e software acquistati da terzi per poter svolgere l'attività di logging, risparmiando notevolmente sul costo di produzione finale del banco.
- La presenza di hardware Beckhoff sulla maggior parte dei loro banchi test.

In generale la tesi si comporrà di 10 capitoli, dove nei primi 2 saranno illustrate le principali caratteristiche del linguaggio e dell'hardware al fine di facilitare la comprensione da parte del lettore dell'elaborato. Nei successivi capitoli verrà poi data una panoramica generale delle modalità di sviluppo alle motivazioni delle scelte progettuali che hanno portato alla creazione di una prima versione di data-logger dalla quale si è successivamente partiti per effettuare vari miglioramenti in modo da incrementare in maniera notevole le performance.
Nei capitoli finali saranno commentati e mostrati i test fatti sulle varie versioni di data-logger, insieme saranno fornite anche diverse possibilità di sviluppo futuro.

Va evidenziato che l'obiettivo iniziale, ovvero quello di campionare tra i 30 e i 50 canali al millisecondo, era già stato raggiunto nella prima versione.
Gli step migliorativi realizzati nel programma attraverso l'ottimizzazione software hanno consentito di estendere questa capacità di campionamento fino a raggiungere i 300 canali al millisecondo.

1 Introduzione a TwinCAT Beckhoff

The **Win**dows **C**ontrol and **A**utomation **T**echnology [1] è un sistema di controllo per l'automazione industriale PC-based, che trasforma ogni PC in un sistema realtime, integrando al suo interno l'ambiente di sviluppo. Per ottenere le massime prestazioni dal sistema nelle applicazioni industriali, si fa solitamente uso di PC Beckhoff che si basano su sistemi operativi Windows ottimizzati per il funzionamento realtime. All'interno dello stesso software è quindi possibile integrare la gestione del PLC, della parte di motion ed HMI[1] in modo da ottenere una riduzione della complessità dell'interfaccia, tempi ciclo inferiori, diagnostica migliorata e per ultimo, ma non meno importante, una significativa riduzione dei costi. In generale i sistemi operativi Windows non hanno caratteristiche realtime e quindi non sono stati progettati per gestire task di controllo. La presenza stessa dello scheduler implica infatti che i vari task possano essere interrotti durante la loro esecuzione da una serie di altri eventi con priorità maggiore. Questa tipologia di funzionamento va in contrasto con la normale esecuzione ciclica di un programma PLC che prevede le fasi di lettura input, esecuzione codice e scrittura output come mostrato in Figura 1.

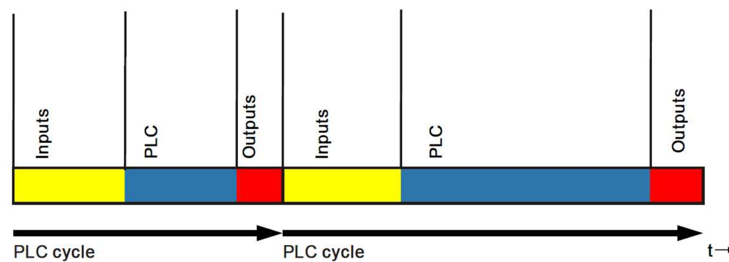


Figura 1: Scansione standard in un PLC

TwinCAT, opera tramite il suo real-time engine che si posiziona allo stesso livello del kernel, quando esso è installato e in run mode, Windows diviene un semplice task in esecuzione e questo permette al TwinCAT real-time engine di poter continuare ad operare anche in caso di crash del sistema operativo.

In un soft-PLC i programmi sono eseguiti nella stessa forma di un PLC convenzionale. L'interfaccia utente (HMI) è eseguita durante gli intervalli temporali riservati dal sistema alle normali operazioni del sistema operativo. TwinCAT assicura sia l'esecuzione delle operazioni del sistema operativo che dei programmi PLC attraverso una speciale implementazione e gestione dei task integrata nel sistema operativo stesso come riportato in Figura 2.

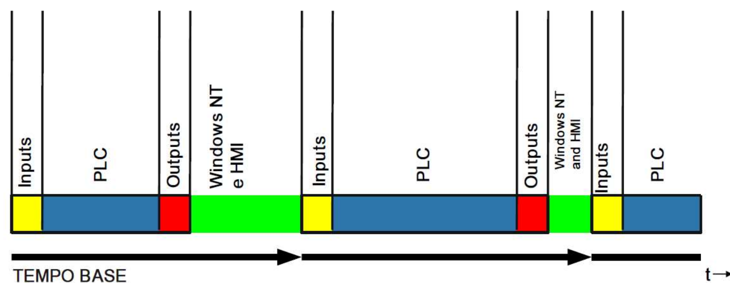


Figura 2: Scansione PLC con TwinCAT

[1]Human Machine Interface, sono interfacce per facilitare l'utilizzo della macchina da parte dell'utente

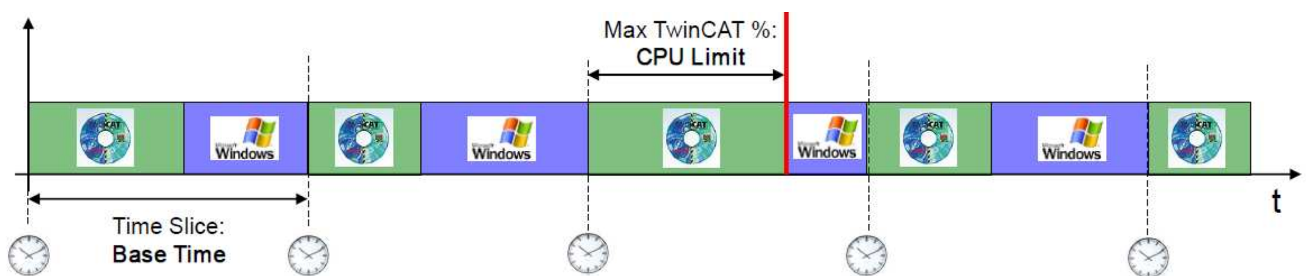
1.1 TwinCAT real-time engine

Il motore TwinCAT Real-Time Engine[2] offre un'elaborazione in tempo reale affidabile e di alta precisione per il controllo di macchine e processi industriali. È in grado di gestire operazioni con tempi di ciclo molto brevi, garantendo una rapida risposta ai segnali di input e consentendo una sincronizzazione accurata delle azioni di controllo.

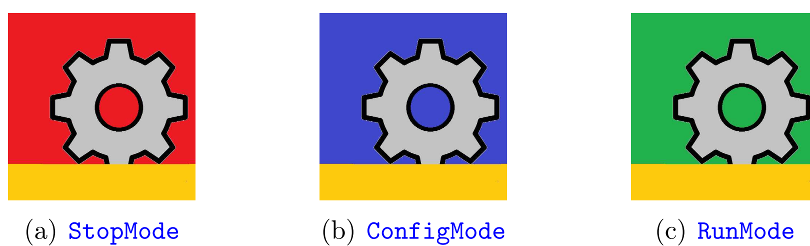
Esso può assumere 3 modalità operative:

- **Stop Mode**: dove sia real-time che funzionalità di alto livello sono disabilitate;
- **Config Mode**: dove real-time disabilitato, funzionalità di alto livello abilitate;

- **Run Mode**:dove sia real-time che funzionalità di alto livello sono abilitate.



In Figura 3 si riporta la simbologia che segnala gli stati operativi dell' engine, che sono rispettivamente:



(a) StopMode (b) ConfigMode (c) RunMode

Figura 3: Stati operativi

1.2 Scheduler TwinCAT

Lo scheduler divide il tempo in incrementi (ticks) definiti dal *Base Time*, ed ogni volta che viene raggiunto un tick, il controllo passa da Windows a TwinCAT. Se il TwinCAT deve eseguire qualsiasi task, lo inizia allo "scatto" del tick, altrimenti il controllo viene restituito nuovamente a Windows. Oltre al *Base Time* viene definito anche un *CPU limit* che esprime la percentuale massima di tempo di tick che un task TwinCAT può utilizzare su un determinato processore.
Infine si ha il *Task Cycle Time* che rappresenta l'intervallo di tempo impiegato da un task specifico per completare il proprio ciclo di esecuzione, ovvero il tempo che un task richiede per svolgere le proprie operazioni di controllo. Mentre il task cycle time è specifico per un task individuale, il base time definisce la risoluzione del sistema di controllo e influenza la frequenza di esecuzione di tutti i task nell'applicazione. È importante bilanciare adeguatamente sia il task cycle time che il base time per garantire un controllo preciso e un utilizzo efficiente delle risorse di automazione.
A titolo esemplificativo imponiamo per i 3 parametri i seguenti valori:

- **Base Time:** 1 ms;
- **CPU Limit(%):** 80;
- **Task Cycle Time:** 3 ms;

I tick TwinCAT hanno una precisione di +/-12 μs per il *determinismo del controllo*[2]. Imporre il *Task Cycle Time* a 3ms con un *Base Time* di 1ms equivale a dire che il task inizierà ogni 3 ms come riportato in Figura 4.
Un altro parametro molto importate da tenere in considerazione è il tempo di esecuzione del task (*Task Execution Time*) che rappresenta il tempo impiegato dal task per eseguire tutte le istruzioni previste. Ovviamente il tempo di esecuzione dei task è tanto inferiore quanto più è performante il processore, inoltre esso è anche indipendente anche dal *Base Time*. In figura 4 è stato riportato un esempio considerando un *Task Execution Time* di 1 ms.

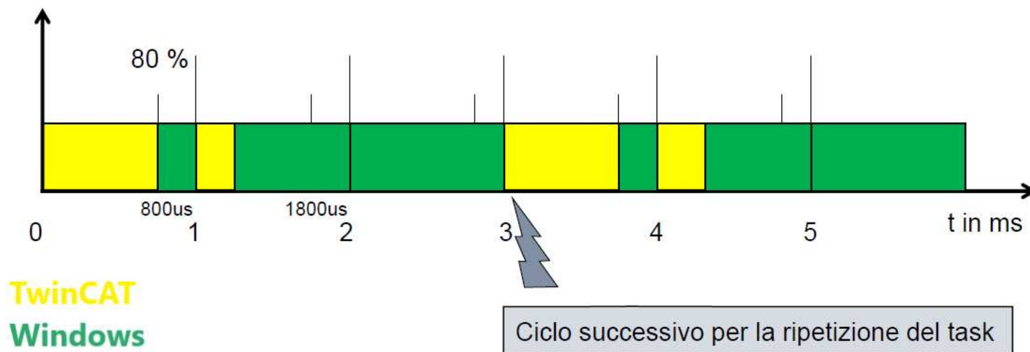


Figura 4: Esempio di funzionamento dello scheduler con tempo di esecuzione del task < del task cycle time

[2]determinare se restituire il controllo al S.O. oppure al task

Nel caso in cui invece si decidesse di andare a modificare il tempo di ciclo del task da 3 ms ad 1 ms si provocherebbe un **overrun**, ovvero il task TwinCAT non riuscirebbe a terminare nel tempo disponibile. Il sistema in questo caso lo limita all'80% bloccando il task, permettendo così a Windows di riprendere l'esecuzione come mostrato in Figura 5, infatti Se TwinCAT potesse usare il 100% del Base Time, il PC andrebbe in blocco lasciando come unica opzione quella del riavvio.

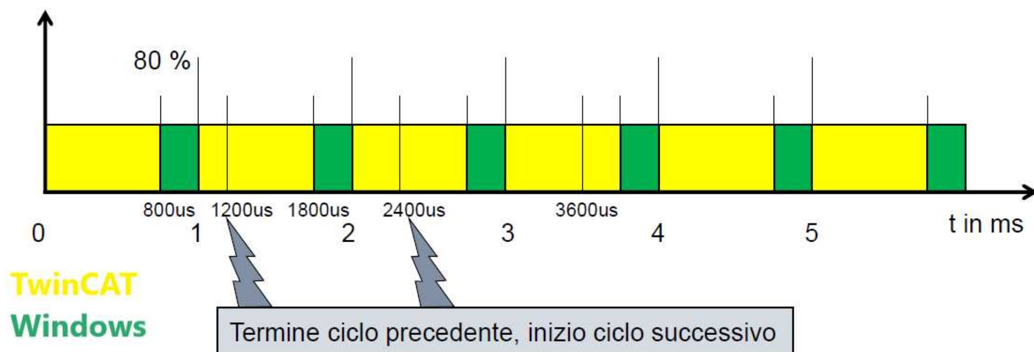


Figura 5: Esempio di funzionamento dello scheduler con tempo di esecuzione del task < del task cycle time

Per questo progetto il problema non è stato tanto quello di creare un programma funzionante ma più quello di creare un programma che non generasse questi overrun, Esistono variabili di sistema che indicano se e quando un task eccede il suo tempo ciclo, un esempio è l'exceed counter riportato in Figura 6.

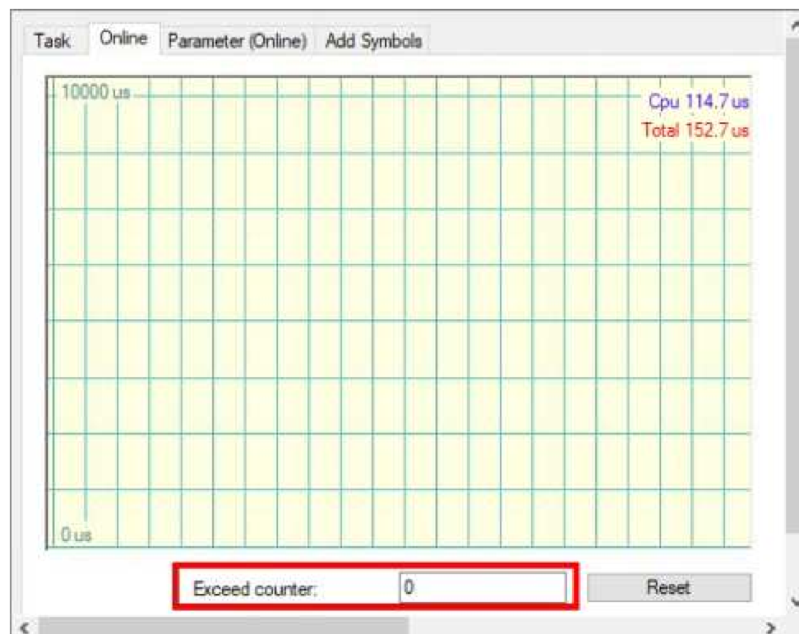


Figura 6: Esempio di exceed counter

Nel progetto di tesi sono stati assegnati i seguenti valori:

- **Base Time:** 1 ms;
- **CPU Limit(%):** 80;
- **Task Cycle Time:** 1 ms;

Come mostrato in Figura 7

Figura 7: Impostazioni TwinCAT

1.3 Linguaggi di programmazione

TwinCAT supporta diversi linguaggi di programmazione basati sullo standard IEC 61131-3[15], che è ampiamente utilizzato nell'automazione industriale. I linguaggi di programmazione messi a disposizione sono:

1. **Ladder Diagram (LD):** È un linguaggio di programmazione a contatti che utilizza simboli grafici per rappresentare la logica di controllo. Rappresenta il linguaggio più utilizzato per la programmazione di PLC e offre una modalità di programmazione visuale e intuitiva delle operazioni logiche.

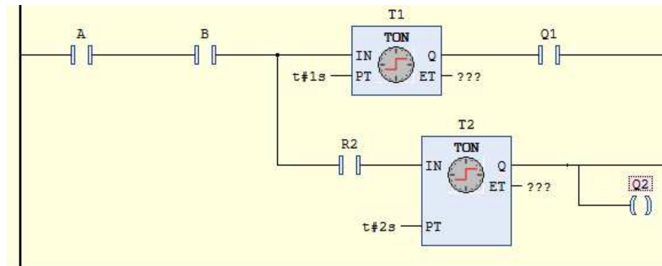


Figura 8: Esempio LD

2. **Function Block Diagram (FBD):** È un linguaggio di programmazione che utilizza blocchi funzionali collegati tra loro per definire la logica di controllo. I blocchi rappresentano funzioni logiche o matematiche e possono essere collegati per creare la sequenza di operazioni desiderata.

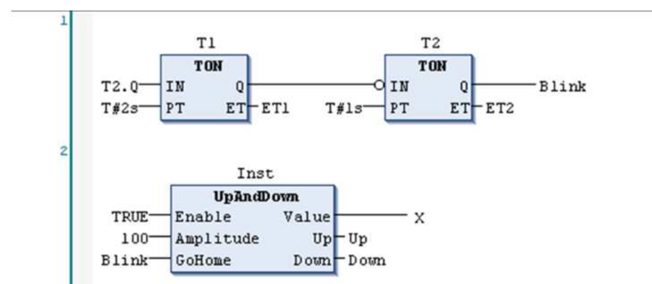


Figura 9: Esempio FBD

3. **Sequential Function Chart (SFC):** È un linguaggio di programmazione che consente di definire la sequenza e il flusso di esecuzione delle attività. È particolarmente utile per rappresentare i processi sequenziali e gli stati di un'applicazione.

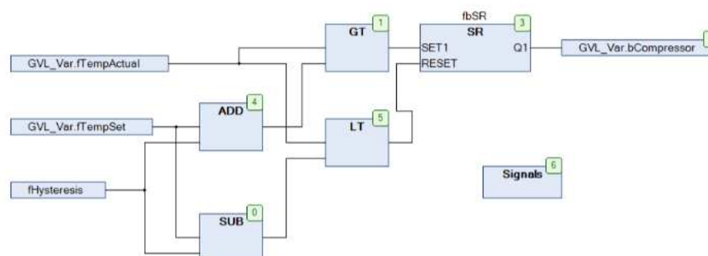


Figura 10: Esempio SFC

4. **Structured Text (ST):** È un linguaggio di programmazione testuale che utilizza la sintassi simile a C per scrivere il codice. Offre una maggiore flessibilità e potenza rispetto ai linguaggi grafici e consente di implementare algoritmi complessi e personalizzati.

```
R_TRIG_START(CLK:=start);
F_TRIG_STOP(CLK:=start);

IF R_TRIG_START.Q THEN
    GVL_LOGGER.FB_Logger_0.startLogger();
END_IF


IF F_TRIG_STOP.Q THEN
    GVL_LOGGER.FB_Logger_0.stopLogger();
END_IF
```

Figura 11: Esempio ST

5. **Instruction List (IL):** È un linguaggio di programmazione a basso livello che utilizza istruzioni mnemoniche per definire la logica di controllo. È simile al linguaggio assembly ed è utilizzato per ottimizzare le prestazioni in applicazioni specifiche.

È possibile combinare questi linguaggi all'interno di un progetto TwinCAT, a seconda delle esigenze specifiche dell'applicazione. Ogni linguaggio ha i suoi punti di forza e si adatta meglio a determinati casi d'uso. La scelta del linguaggio dipenderà dalla complessità dell'applicazione, dalle preferenze del programmatore e dalle convenzioni di programmazione dell'azienda.
Nel caso del progetto in questione sono state utilizzate una combinazione dei primi 4 linguaggi di programmazione.

Allo standard internazionale fanno riferimento anche la struttura del programma (POU), la dichiarazione delle variabili e gli operatori e funzioni di base. Per quanto riguarda le POU (Program Organization Unit) la normativa prevede tre differenti unità di software:

- Programmi (PRG), possono chiamare blocchi funzioni, funzioni e altri programmi e mantengono memoria dello stato delle variabili interne anche nei cicli PLC successivi;
- Blocchi funzionali (FB), prevedono variabili in ingresso ed in uscita ma è necessario istanziarle. Anche queste mantengono in memoria lo stato delle variabili interne nei cicli PLC successivi;
- Funzioni (FUN), possono avere più variabili in ingresso ma solo una in uscita, non richiedono istanze e non mantengono in memoria lo stato delle variabili interne.

2 Introduzione all'hardware Beckhoff

Per il DataLogger è stata fornita una CX5140 (Figura 12) dotata di un processore quad-core Intel Atom® con una frequenza di clock di 1,91 GHz che rende possibile una vera e propria tecnologia multi-core nel segmento dei PC embedded. Sono disponibili due interfacce Ethernet indipendenti con capacità dell'ordine dei Gigabit, quattro USB 2.0 e un'interfaccia DVI-I. Queste caratteristiche hardware fanno di lui uno dei modelli migliori utilizzati nei banchi test.

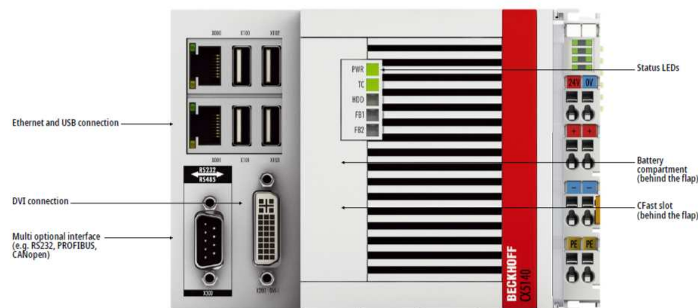


Figura 12: CX-5140 con gli ingressi e le uscite

3 Sintesi dell'approccio

Per sviluppare il software, si è partiti dalla studio del function block FB_CTRL_LOG_DATA[9] della Beckhoff in figura 13 che, come viene detto nella documentazione ufficiale, è un blocco che consente di creare un file di log in formato *.csv, in cui possono essere registrati al massimo 10 canali contemporaneamente. Le intestazioni (header) delle colonne specificate dall'utente sono scritte nella prima riga del file, i dati di input invece vengono scritti a intervalli di tempo uguali nelle righe successive dove le singole voci sono separate da una virgola. L'intervallo di tempo tra le voci è specificato nel parametro *tLogCycleTime*. Se, ad esempio, venisse scelto un *tLogCycleTime* := T#2s, nel file verrebbe inserita una voce ogni 2s. Il *time stamp*[3], espresso in secondi, è memorizzato nella prima colonna del file. Le altre colonne contengono i dati campionati dal blocco funzione *fLogData*[4].

FB_CTRL_LOG_DATA

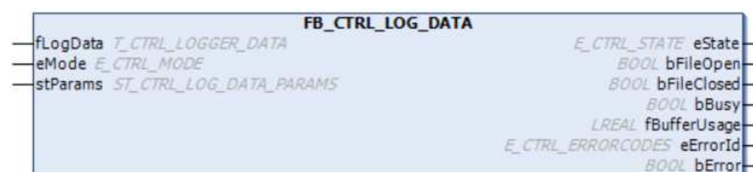


Figura 13: Blocco funzione FB_CTRL_LOG_DATA

[3]istante in cui il dato è stato registrato

[4]array contenente i valori da scrivere nel file di log

Una volta studaite le funzionalità base di questo logger si è deciso, come primo obiettivo, di cercare di creare un logger con le stesse caratteristiche e di far aumentare il numero di canali campionabili contemporaneamente da 10 ad un minimo di 30 al ms e di dare la possibilità all'utente di poter segmentare il file *.csv per tempo o per dimensione, ovvero:

- **Segmentazione per tempo:** Si supponga di voler campionare 300 canali ogni millisecondo e di voler e di voler segmentare il file ogni 30 secondi. In questo intervallo di tempo verrebbe generato un file da 30.000 righe con 300 colone, e allo scoccare dei 30 secondi sarebbe chiuso il file corrente e ne verrebbe aperto uno nuovo all'interno del quale si continuerebbero a scrivere gli altri valori campionati (ovvero quelli che nel primo file si sarebbero scritti dalla riga 30.001) a partire dalla riga 1. Questo verrà fatto fino a che i 30 secondi non scadranno nuovamente, a quel punto il ciclo sopra descritto verrà reiterato fino a che non sarà l'utente stesso a bloccare l'acquisizione premendo il pulsante di stop;
- **Segmentazione per dimensione:** Si supponga di voler campionare 300 canali ogni millisecondo e di voler e di voler segmentare il file ogni 30 MB. Al raggiungimento dei 30 MB sarebbe chiuso il file corrente e ne verrebbe aperto uno nuovo all'interno del quale si continuerebbero a scrivere gli altri valori campionati a partire dalla riga 1. Questo verrà fatto fino a che i 30 MB non saranno raggiunti nuovamente, a quel punto il ciclo sopra descritto verrà reiterato fino a che non sarà l'utente stesso a bloccare l'acquisizione premendo il pulsante di stop.

Una volta definite le funzionalità, per lo svilupo del software si è pensato di utilizzare 3 Function Block:

1. **FB_FileOpen[6]:** il quale crea un nuovo file *.csv o ne apre uno già esistente per la modifica.
2. **FB_FileWrite[7]:** il quale si occupa di scrive i dati all'interno del file *.csv. Per l'accesso in scrittura il file deve essere stato aperto nella modalità corrispondente e deve essere nuovamente chiuso per l'eventuale elaborazione da parte di programmi esterni.
3. **FB_CSVMemBufferWriter[4]:** Questo function block può essere utilizzato per generare set di dati in un array di byte esterno in formato CSV. Il contenuto dell'array potrà poi essere scritto in un file utilizzando i function block elencati sopra per l'accesso al file. Il nuovo dato da memorizzare può essere trasferito al blocco tramite la variabile *putValue* (stringa) o tramite le variabili opzionali *pValue* e *cbValue* (fa uso dei puntatori). La fine del set di dati viene aggiunta automaticamente al campo di dati se la variabile *bCRLF* è stata impostata su TRUE durante la scrittura dell'ultimo valore.
Va considerato inoltre che il blocco aggiunge in automatico il separatore di campo (il punto e virgola) tra un valore ed un altro.
4. **FB_FileClose[5]:** il quale chiude il file, mettendolo in uno stato definito per l'ulteriore elaborazione da parte di altri programmi. Se il file non venisse chiuso correttamente quest'ultimo non potrebbe essere aperto da programmi esterni per l'ulteriore elaborazione.

Combinando correttamente questi 3 blocchi è quindi possibile scrivere all'interno di un file *.csv, ma questo non è sufficiente in quanto, come riportato dai risultati sperimentali, scrivere su un file *.csv 30 canali contemporaneamente richiede più di 1 ms con conseguenti perdite di dati nel campionamento. Come soluzione a questo problema si è pensato di adottare un sistema di buffering, in particolare si è utilizzata una logica a doppio buffer la quale verrà approfondita in seguito insieme a quanto detto sopra.

Una volta ottenuto un sistema di logging con caratteristiche accettabili in grado di campionare almeno 30 canali/ms, sono state apportate delle importanti ottimizzazioni software che hanno consentito al logger di arrivare a campionare un numero di canali pari a 10 volte quelli di partenza, ovvero 300 canali/ms. Le modifiche software si sono divise essenzialmente in 5 steps:

1. Buffer e Logger rappresentati da due FB divisi: rappresentazione più intuitiva ma meno efficiente, il logger infatti campiona al massimo 30 canali/ms ;
2. Buffer e Logger uniti con assegnazione e concatenzione di stringhe;
3. Buffer e Logger uniti con puntatori a interi e concatenazione di stringhe;
4. Buffer e Logger uniti con puntatori a interi e senza concatenazione di stringhe;
5. Buffer e Logger divisi in due PRG con utilizzo di variabili globali

Per ognuno di queste versioni sono stati effettuati dei test sui limiti con le medesime configurazioni:

- *Tempo di campionamento :* 1 ms
- *CPU Limit(%) :* 80

I risultati verranno illustrati e discussi nel Capitolo 6 e 7.

Infine per la quarta tipologia di DataLogger della lista si sono eseguiti anche due stress test lasciando la CX accesa per tutta la notte ed eseguendo la segmentazione per tempo (ogni 10 minuti) e per dimensione (ogni 100 MB).

4 Analisi dei Requisiti

Di seguito vengono riportate due tabelle che rappresentano in maniera più leggibile e chiara quanto detto nei precedenti capitoli riguardo le funzionalità che il data-logger dovrebbe avere. I requisiti sono stati divisi in funzionali e non funzionali.

4.1 Requisiti funzionali

REQUISITO	DESCRIZIONE
RF1 Start	L'utente dovrà avere la possibilità di avviare manualmente il data-logger
RF2 Stop	L'utente dovrà avere la possibilità di stoppare manualmente il data-logger
RF3 Nome File	L'utente deve poter scegliere il nome del file
RF3 Path	L'utente deve poter scegliere dove verranno salvati i file *.csv
RF4 Header	Per ogni canale da loggare l'utente deve poter scegliere un nome da associare ai valori campionati da quel canale. Tale nome sarà presente in testa alla colonna all'interno del file *.csv
RF5 Numero di canali da loggare	L'utente deve poter scegliere quanti canali loggare contemporaneamente
RF6 Tempo di campionamento	L'utente deve poter scegliere ogni quanti ms campionare i canali
RF7 Numero minimo di canali	L'applicazione dovrà essere in grado di campionare almeno 30 canali/ms
RF8 Nessuna perdita dati	Il sistema deve riuscire a campionare più canali alla frequenza massima senza perdere alcun dato, nè in fase di buffering nè in fase di scrittura su file
RF9 Segmentazione file per dimensione	L'utente deve avere la possibilità di scegliere la grandezza massima del file *.csv
RF10 Segmentazione file per tempo	L'utente deve avere la possibilità di indicare il tempo massimo di campionamento
RF11 Numero incrementale	L'applicazione dovrà aggiungere in coda al nome del file un numero incrementale utile in caso di segmentazione a capire in che ordine sono stati generati i file

4.2 Requisiti non funzionali

REQUISITO	DESCRIZIONE
RNF1 TwinCAT	L'applicazione dovrà essere sviluppata usando l'ambiente TwinCAT e i linguaggi che esso mette a disposizione
RNF2 Affidabilità	Il software deve funzionare per lunghi periodi di tempo senza interruzioni o errori
RNF3 Efficienza	Il software deve fare un buon uso delle risorse di sistema
RNF4 Usabilità	L'utente deve poter utilizzare in maniera facile e agevole il programma

5 Progettazione iniziale del software

Nel seguente capitolo, sarà illustrato in maniera generale il data-logger, e verrà fatto un focus sulle sue due componenti più importati, ovvero il buffer ed il logger. Verranno mostrate infatti quali sono state le scelte progettuali e le motivazioni che hanno portato alla creazione della prima versione di data-logger. Le due componenti saranno una costante in tutte e cinque le versioni di data-logger sviluppate,anche se in alcune versioni la distinzione tra le due sarà meno netta; questo si capirà meglio nel capitolo 6 e 7 quando si andranno a trattare nel particolare i data-logger. Alla fine di questo capitolo verrà fornita anche una visione generale della struttura del progetto TwinCAT.

5.1 Descrizione dell'architettura generale del data-logger

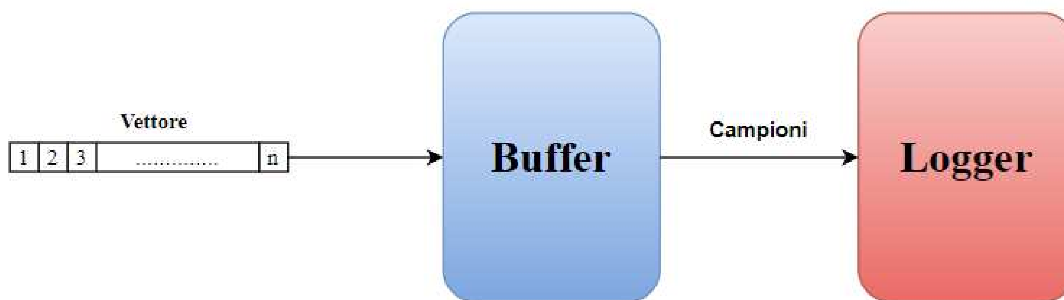


Figura 14: Architettura generica data-logger

Viene fatto notare che per il progetto non avendo a disposizione 30 o più sensori da collegare al PLC, i valori "campionati" sono stati inventati andando ad inserire all'interno di un vettore di n elementi dei valori casuali che poi venivano variati manualmente al fine di assicurarsi che questo cambiamento avvenisse anche in fase di scrittura sul file *.csv . Il vettore contenente i dati da "campionare" è stato rappresentato come se fosse composto da un numero generico n di elementi, questo per rendere la rappresentazione valida anche per le altre versioni di logger in quanto, in base all'ottimizzazione, sarà possibile loggare una diversa quantità massima di canali. Il data-logger, mostrato in figura 14, presenta due parti fondamentali:

- Il **Buffer :** Esso viene utilizzato per far comunicare due componenti che hanno velocità diverse, infatti essendo la fase di scrittura più lenta di quella di acquisizione dati si rende necessario creare una struttura in grado di mantenere in memoria i dati acquisiti fino a che essi non vengono definitivamente salvati dal logger tramite scrittura su file .csv .
- Il **Logger :** Esso si occupa di prelevare correttamente i dati dal buffer e di andarli a scrivere su file *.csv gestendo contemporaneamente anche tutti quelli che possono essere dei possibili errori andandoli a segnalare all'utente.

5.2 Descrizione della struttura generale del progetto TwinCAT

Dopo aver creato il progetto TwinCAT e aver configurato l'ambiente di sviluppo le cartelle di progetto sono strutturate come mostrato in figura 15.

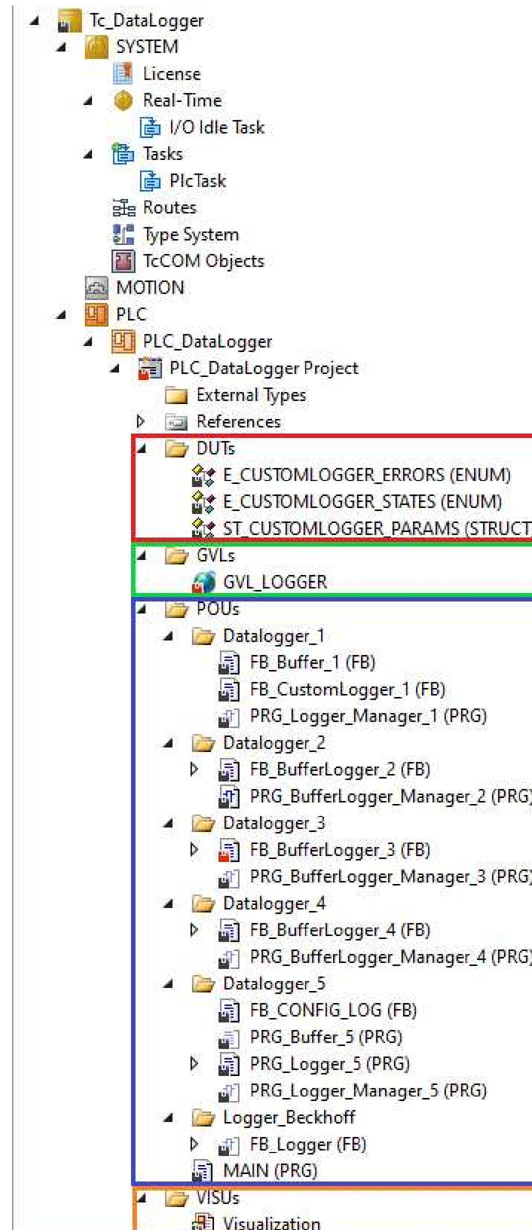


Figura 15: Organizzazione del progetto

In un progetto base TwinCAT si hanno quattro cartelle fondamentali la cartella DUTs (Data Unit Types) evidenziata da un rettangolo rosso, la cartella GVLs (Global Variable Lists) evidenziata da un rettangolo verde, la cartella POUs (Program Organization Units) evidenziata da un rettangolo blu, la cartella VISUs (Visualizations) evidenziata da un rettangolo arancione.

Cartella DUTs

La cartella DUTs[3] viene utilizzata per contenere i tipi di dati definiti in maniera personalizzata. Questi tipi di dati personalizzati possono essere utilizzati successivamente nelle POUs, ad esempio, per definire variabili, parametri, o argomenti di funzioni.I tipi personalizzati usati nel progetto sono due:

- Tipo **ENUM :** esso è utilizzato per definire un insieme di costanti con nomi associati, permettendo così di creare una lista di valori possibili che possono essere assegnati ad una variabile specifica. L'utilizzo di dati di tipologia ENUM semplifica la gestione di un insieme di errori o stati possibili, fornendo nomi significativi ed evitando così la scomodità di dover confrontare valori numerici arbitrari. Inoltre, l'ENUM può migliorare la leggibilità del codice e rendere più intuitiva la comprensione delle variabili che rappresentano uno stato o un'opzione specifica. La tipologia ENUM in questo progetto è stata usata per definire un insieme di errori e stati come mostrato negli estratti di codice riportati sotto.
- Tipo **STRUCT :** esso viene utilizzato per definire una struttura che raggruppa più variabili o elementi correlati in una singola entità, permettendo così di creare un nuovo tipo di dato personalizzato che contiene campi o membri di dati di tipo diverso. Nel progetto è stato utilizzato, come è possibile vedere sotto, per rappresentare tutti i possibili parametri di configurazione che l'utente può impostare.

```
1 TYPE E_CUSTOMLOGGER_ERRORS :
2 (
3   eERROR_NOERROR     := 0,
4   eERROR_FILEOPEN    := 1,
5   eERROR_FILEWRITE   := 2,
6   eERROR_FILECLOSE   := 3,
7   eERROR_INVALID_SEGMENTATION_PARAM := 4
8 );
9 END_TYPE
```

Listing 1: ENUM per errori

```
1 TYPE E_CUSTOMLOGGER_STATES :
2 (
3     eSTATE_ACQUIRING_TIME          := 0,
4   eSTATE_WAITING_ACQUISITION       := 5,
5   eSTATE_FILENAME_AND_PATH_CREATION    := 10,
6   eSTATE_OPENING_FILE          := 15,
7   eSTATE_WAITING_FILE_OPENING      := 20,
8   eSTATE_FILEHEADER_CREATION       := 25,
9   eSTATE_CHECKING_BUFFER         := 30,
10  eSTATE_FILEDATA_PROCESSING       := 35,
11  eSTATE_WRITING_FILE          := 40,
12  eSTATE_CLOSING_FILE          := 45,
13  eSTATE_IDLE              := 99,
14  eSTATE_ERROR             := 999
15 );
16 END_TYPE
```

Listing 2: ENUM per stati macchina

```
1  TYPE ST_CUSTOMLOGGER_PARAMS :
2  STRUCT
3    sFileName                   : STRING;
4    sFilePath                   : STRING;
5    sNetId                      : STRING;
6    nNumberOfColumns            : INT(1..MAX_COLUMNS-1);
7    arColumnHeadings            : ARRAY [1..MAX_COLUMNS-1] OF STRING;
8    tSegmentationTime           : TIME;
9    nSegmentationDimension      : INT;
10   tLogCycleTime               : TIME;
11 END_STRUCT
12 END_TYPE
13 }
```

Listing 3: ENUM per stati macchina

Cartella GVLs

La cartella GVLs[8] contiene le liste delle variabili globali che possono essere accessibili e utilizzate in diverse parti del progetto TwinCAT. Questa cartella viene quindi utilizzata per definire e organizzare le variabili globali che si sono rivelate molto utili nell'ultima versione di data logger dove grazie al loro impiego è stato possibile disaccoppiare nuovamente le due componenti software. Sotto viene riportato il file GVL_LOGGER di progetto.

```
1  VAR_GLOBAL CONSTANT
2    MAX_ROWS       : INT := 1000;
3    MAX_COLUMNS    : INT := 301;
4  END_VAR
5
6  VAR_GLOBAL
7
8    (* VAR SHARED BETWEEN BUFFER AND LOGGER *)
9
10   bStart           : BOOL;
11   stParams         : ST_CUSTOMLOGGER_PARAMS;
12   nBufferActive    : SINT;
13   arDataBuffer1    : ARRAY [1..MAX_ROWS, 1..MAX_COLUMNS] OF STRING ;
14   arDataBuffer2    : ARRAY [1..MAX_ROWS, 1..MAX_COLUMNS] OF STRING ;
15   arData           : ARRAY [1..MAX_COLUMNS] OF INT :=[1,2,3,4,5,6,7,8,9,
16                         10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,
17                         26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,
18                         42,43,44,45,46,47,48,49,50];
19   nRowActive     : INT;
20
21   (* OUTPUT VAR OF LOGGER *)
22
23     bError           : BOOL;
24     eErrorId         : E_CUSTOMLOGGER_ERRORS;
25     bFileOpen        : BOOL;
26     bFileClosed      : BOOL;
27     bBusy            : BOOL;
28     eLoggerState     : E_CUSTOMLOGGER_STATES;
29
30
31
32
33   (* CONFIGURATION VAR AVAIABLE ONLY ON v.5 *)
34
35    bConfig                    : BOOL;
36    FB_CONFIG_LOG_0            : FB_CONFIG_LOG;
37    sFileName                  : STRING;
38    sFilePath                  : STRING;
39    tLogCycleTime              : TIME;
40    tSegmentationTime          : TIME;
41    nNumberOfColumns           : INT(1..MAX_COLUMNS-1);
42    arColumnHeadings           : ARRAY [1..MAX_COLUMNS-1] OF STRING
43    nSegmentationDimension     : INT;
44
45
46
47   (* VAR BUFFER AND LOGGER WITH CONCAT AND ASSIGNMENT *)
```

```
48
49    FB_CustomLogger_0       : FB_CustomLogger_1;
50    FB_Buffer_0             : FB_Buffer_1;
51
52   (* VAR BUFFER + LOGGER WITH CONCAT AND ASSIGNMENT *)
53
54    FB_BufferLogger_0     : FB_BufferLogger_2;
55
56   (* VAR BUFFER + LOGGER WITH POINTER + CONCAT *)
57
58    FB_BufferLogger_1     : FB_BufferLogger_3;
59
60   (* VAR BUFFER + LOGGER WITHOUT CONCAT*)
61
62    FB_BufferLogger_2     : FB_BufferLogger_4;
63
64 END_VAR
```

Listing 4: Dichiarazione lista di variabili globali (GVL)

Cartella POUs

La cartella POUs[10] contiene le unità di organizzazione del programma. Esse sono le componenti principali di un progetto TwinCAT e possono includere programmazione strutturata (ST), funzioni, funzioni di blocco, diagrammi di sequenza, ecc. In questa cartella vengono quindi organizzati e archiviati i file sorgente che costituiscono il cuore di qualsiasi programma. Nel caso di progetto al suo interno possiamo notare cinque cartelle diverse ognuna delle quali contiene una versione di data-logger sviluppata. Queste versioni verranno analizzate poi nel dettaglio nel Capitolo 6 e 7.
Infine è possibile notare un'altra cartella contenente il function block con limitazioni messo a disposizione da Beckhoff per il data logging, questo blocchetto è stato utilizzato come detto in precedenza solo per definire le funzionalità di base del nuovo data-logger.

Cartella VISUs

All'interno della cartella troviamo il file *Visualization*[11] di cui si discuterà in maniera approfondita nel Capitolo 9. Per questo capitolo è sufficiente dire che nella cartella VISUs vengono create le HMI.

Scope Project

Ora che è stata data una descrizione di alto livello della struttura del progetto si può anche fare una breve deviazione ed illustrate con la figura 16 uno degli strumenti che si è rivelato molto utile per il debug: il TwinCAT 3 Scope View[14]. Si tratta di un oscilloscopio digitale basato su software, fornito gratuitamente con TwinCAT 3, che si è rivelato utile per la risoluzione di fastidiose condizioni transitorie di cui si discuterà approfonditamente nel Capitolo 8 mostrando l'andamento temporale delle variabili.

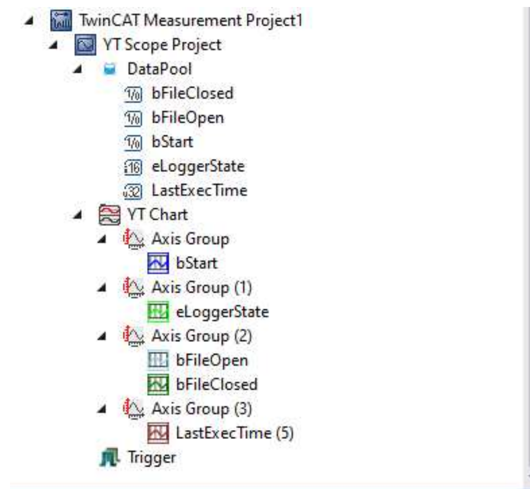


Figura 16: Organizzazione del progetto

5.3 Sistemi di buffering valutati

La funzione principale del buffer, come detto sopra, è quella di mitigare la differenza di velocità o il ritardo tra la produzione e il consumo dei dati. Nel caso di progetto è stato interposto tra il vettore, che ogni millisecondo veniva riempito con valori casuali, e il logger vero e proprio che si occupava invece di scriverli su file, impiegando però più di un millisecondo. Senza di esso tutti i "nuovi" valori all'interno del buffer, presenti allo scattare del millisecondo successivo sarebbero andati persi, così come tutti i dati successivi fino a che il logger non fosse tornato nuovamente disponibile. Il buffer consente di disaccoppiare l'acquisizione dalla scrittura su file, senza la necessità di essere sincronizzato o disponibili contemporaneamente ai nuovi valori inseriti nel vettore. Durante la fase di progettazione è stato necessario scegliere tra due diverse possibilità implementative, ovvero :

- Il **Buffer circolare[13]**
- Il **Doppio Buffer[12]**

Di seguito viene riportata una spiegazione del loro funzionamento e delle considerazioni che hanno portato infine alla scelta di implementare la logica a doppio buffer:

Buffer Circolare

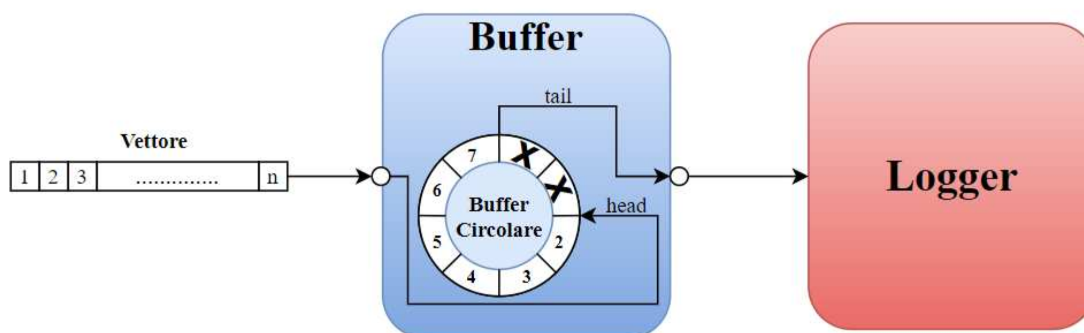


Figura 17: Esempio di buffer circolare

Il buffer circolare è una tipologia di buffer che ha dimensioni fisse e può essere inteso come una struttura dati che utilizza un buffer come se fosse collegato da un'estremità all'altra dando appunto l'impressione di circolarità; l'aggiunta di ulteriori informazioni quando il buffer è pieno provoca la sovrascrittura del primo elemento memorizzato nel buffer. Esso viene anche chiamato *buffer ad anello* o *buffer ciclico*.

Per capire meglio come funziona ci si può basare su un esempio e sulla figura 17. Immaginando di avere un buffer circolare che ha dimensione di 8 celle, numerate da 0 a 7.
In questo caso allora i dati verrebbero sempre aggiunti all'interno della cella in cui il puntatore *head* sta puntando, ed una volta inserito il valore il puntatore si sposterebbe alla cella successiva e così via fino ad arrivare alla cella 2. Per quanto riguarda la rimozione invece, i dati verrebbero sempre rimossi dall'altra estremità ovvero dalla cella puntata dal puntatore *tail*.
Ricordano che la dimensione del buffer è fissa e che quindi ha una certa capacità massima, si fa notare che quando il buffer è pieno, si iniziano a sovrascrivere i dati in testa.
Prima che ciò accada però, il logger dovrà aver scritto su file i dati che vengono sovrascritti

in modo da non incorrere in nessun rischio di perdita informazioni. La capacità massima del buffer deve essere impostata in anticipo e, sebbene questo numero possa essere modificato in qualsiasi momento, in tal caso tutti i dati esistenti presenti nel buffer andrebbero persi.

L'attributo più vantaggioso di un buffer circolare è il modo in cui memorizza i dati, essi non vengono mischiati quando viene rimosso un oggetto alla fine della riga. Se il buffer fosse non circolare, tutti gli elementi presenti nel buffer dovrebbero cambiare posizione quando almeno uno di essi viene rimosso. Un buffer circolare può essere pensato come un tipo di buffer First In First Out (FIFO), mentre un buffer standard assomiglia più a un tipo di buffer Last In First Out (LIFO)

Buffer doppio

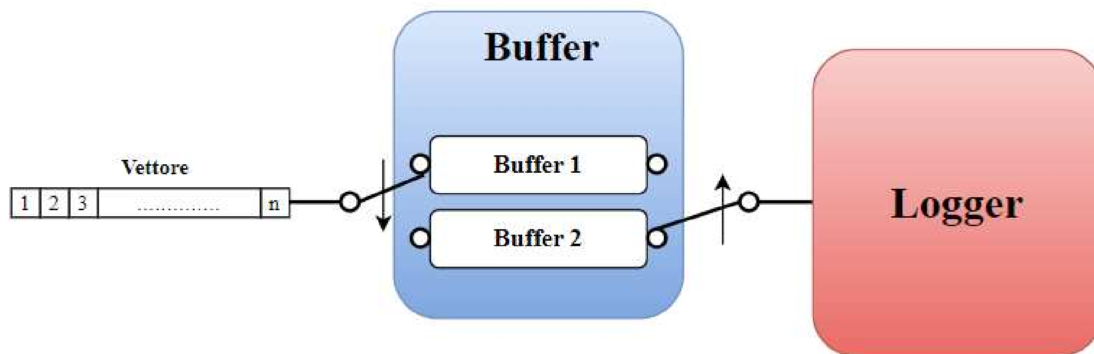


Figura 18: Esempio di buffer doppio

Il doppio buffer è una tipologia di buffer che ha dimensioni fisse e può essere inteso come una struttura dati che utilizza due buffer i quali si scambiano i ruoli al raggiungimento di una determinata occorrenza.
Per funzionare correttamente il buffer avrà bisogno quindi di una sorta di BufferManager che sceglie quale deve essere scritto e quale deve essere invece letto.

Questa tipologia consente di risparmiare tempo e di eseguire più processi contemporaneamente in quanto è possibile utilizzare una serie di dati mentre un'altra serie viene raccolta, scambiandosi ogni volta di ruolo.

Per capire meglio come funziona ci si può basare su un esempio e sulla figura 18. Immaginando di avere due buffer con m celle, ciascuna delle quali può contenere un vettore da n elementi, il BufferManager provvederà inizialmente a far sì che il Buffer1 venga designato come quello da essere riempito da m vettori. Dopodiché, una volta raggiunta la capacità massima, il manager si occuperà di cambiare il buffer su cui inserire i vettori passando dal Buffer1 al Buffer2; a questo punto il Buffer1 potrà essere preso in carico dal Logger che lo scriverà su file. in questa fase è essenziale che il tempo, impiegato dal logger per svuotare il buffer andandolo a salvare su file, sia inferiore al tempo impiegato dal Buffer2 a riempirsi, altrimenti questo comporterebbe un errato salvataggio dei dati. Questo processo verrebbe poi iterato fino al completamento delle operazioni di logging.

L'uso di più buffer aumenta la velocità complessiva di un dispositivo e aiuta a prevenire i colli di bottiglia che non sono altro che battute d'arresto che si verificano quando troppi dati entrano in una sorgente, causando un rallentamento che nel nostro caso porterebbe ad uno scorretto salvataggio dei dati.

5.4 Scelta del sistema di buffering

Dopo aver valutato i pro e i contro delle due soluzioni si è ritenuto più opportuno optare per la logica a doppio buffer. Le motivazioni sono state tendenzialmente 2:

- **Controllo dei conflitti:** La logica a doppio buffer può aiutare a evitare conflitti di accesso ai dati. Poiché i processi di registrazione e lettura/elaborazione lavorano su buffer separati, per questo non ci sarà alcun conflitto o sovrascrittura accidentale dei dati mentre questi vengono elaborati dal logger. Questo aiuta quindi a garantire l'integrità e la coerenza dei dati registrati.
- **Separazione dei processi:** Con una logica a doppio buffer, è possibile separare chiaramente il processo di registrazione dei dati dal processo di lettura e scrittura dei dati su file.

In particolare il Buffer1 e Buffer2 sono stai realizzati come mostrato in figura 19 ovvero, considerando il singolo buffer, come vettore di vettori. In sostanza sia il buffer1 che il buffer2 possono contenere 1000 vettori, ciascuno dei quali è composto da n elementi, dove con n si rappresenta il numero di canali generico che si desidera loggare. Dato che a livello software la dimensione delle matrici va specificato prima (linguaggio staticamente tipizzato) e dato anche il fatto che la quantità massima di canali loggabili al ms con il logger più performante era di 300 canali si è deciso che una matrice di 1000 righe per 301 colonne si potesse ritenere sufficiente.

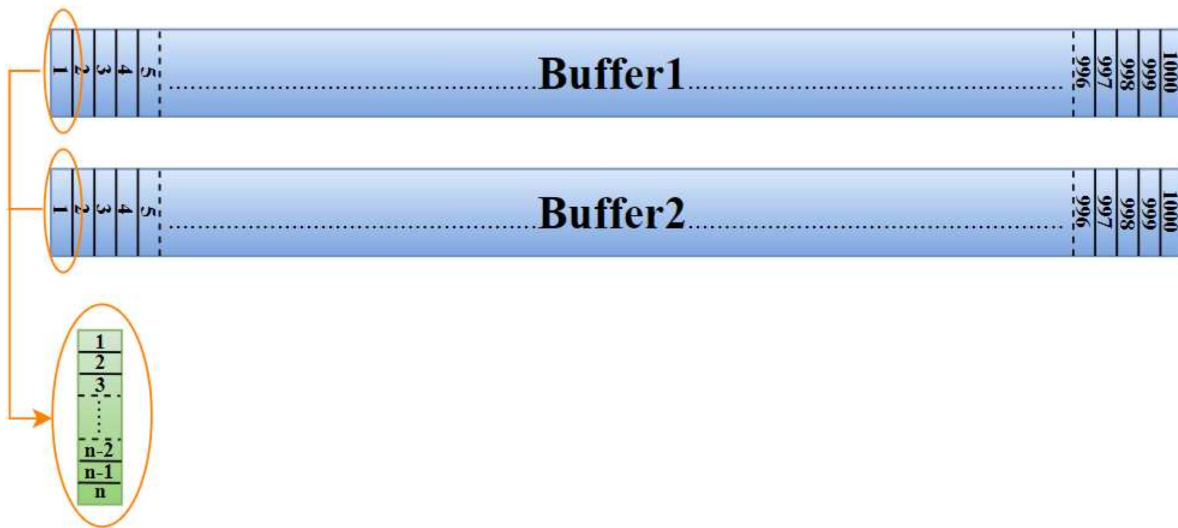


Figura 19: Struttura interna del Buffer1 e del Buffer2

Viene fatto notare inoltre, che potendo i buffer contenere 1000 elementi dove ogni elemento è un vettore con lunghezza pari al numero di canali che si è deciso di campionare, e dovendo il programma effettuare un campionamento al ms, un buffer verrà riempito in 1 s. Questo implica che il logger debba svuotare l'altro buffer scrivendo tutti i valori in un tempo che è inferiore al secondo, proprio per non perdere nessun valore campionato.

5.5 Sistema di logging

Nel sistema di logging, è stata adottata la tecnica di implementazione nota come "macchina a stati", ampiamente utilizzata nell'automazione industriale. Questo approccio prevede di suddividere il processo in diverse fasi, chiamate "stati macchina". Si parte solitamente dal primo stato, in cui le variabili vengono inizializzate in risposta a un determinato evento, nel nostro caso la pressione del pulsante di avvio.

Per passare poi da uno stato macchina al successivo, è necessario attendere il completamento dello stato corrente. Nel linguaggio ST, la macchina a stati viene implementata utilizzando la struttura di controllo "CASE". Ogni istruzione CASE è associata a una variabile di stato macchina (di tipo numerico) che, in base al valore associato, permette di entrare in uno stato specifico. All'interno di CASE è possibile definire una serie di stati, ciascuno dei quali viene assegnato a un valore numerico specifico. Spesso, questi valori sono scalati di un fattore cento per consentire l'inserimento di nuovi stati tra quelli esistenti, senza interrompere la sequenza degli stati successivi.

In sostanza, la macchina a stati offre un approccio strutturato per gestire il processo di logging, consentendo una transizione controllata tra i vari stati e garantendo che le variabili vengano correttamente gestite in base agli eventi che si verificano durante il processo di logging.

Nell'esempio di codice riportato sotto è possibile vedere come funziona una macchina a stati.

```
VAR
  currentState: INT := 0;
END_VAR

CASE currentState OF
  0:
    // Start state 0
    // State actions 0
    currentState := 1; // Transaction a the next state
  1:
    // State 1
    // State actions 1
    IF <Trasaction condition> THEN
      currentState := 2; // Transaction a the next state
    ELSE
      currentState := 3; // Transaction to another state
    END_IF
  2:
    // State 2
    // State actions 2
    IF <Trasaction condition> THEN
      currentState := 1; // Transaction a the next state
    ELSE
      currentState := 3; // Transaction to another state
    END_IF
  3:
    // End state 3
    // State actions 3
    // No transition in this state
END_CASE

\par{
}
```

Listing 5: Esempio di macchina a stati

6 Progettazione ed implementazione del primo data logger

In questa prima versione sviluppata il Buffer ed il Logger sono costituiti da due function block divisi, in grado di comunicare tra loro, mediante lo scambio delle matrici tramite quella che viene detta assegnazione. Sia il buffer che il logger sono dotati di due matrici (301 x 1000), ed ogni qual volta una di queste matrici, appartenenti al buffer, si riempie tutto il su contenuto viene copiato ed assegnato ad una delle due matrici del Logger richiedendo un grosso sforzo a livello di CPU. Questa operazione dovrà poi essere ripetuta all'interno del logger in modo da avere un unica matrice finale su cui poter andare a lavorare chiamata *arDataBuffer*. Questo ulteriore lavoro di copiatura è necessaria per rendere più snello il codice, evitando così duplicazioni inutili dovute alla presenza di due matrici separate da dover gestire all'interno del codice. Il buffer ed il logger saranno gestiti da un PRG chiamato *PRG_Logger_Manager_1* il quale appunto si occuperà di inizializzare tutti i parametri di configurazione utili al buffer ed al logger. Per le prime quattro versioni di data-logger non è possibile effettuare la configurazione mediante l'HMI. Infine all'interno dello stato 35 per cercare di alleggerire l'operazione di immissione dei dati loggati all'interno del blocco funzione CSV_MEM_BUFFER, che si pensava essere un operazione dispendiosa,venivano concatenati all'interno di una stringa che una volta raggiunta la dimensione massima veniva trasferita in blocco al blocco funzione.

6.1 Implementazione logger manager

Come è possibile vedere dalla figura 20 l'implementazione del logger manager è stata fatta usando il linguaggio FBD. Le righe come la 1 e la 2 costituiscono una semplice assegnazione di un valore ad una variabile, tutte queste saranno poi messe in ingresso ai due blocchi funzione che hanno anche delle uscite a cui sono associate altre variabili. Per il buffer ad esempio abbiamo 5 uscite alle quali sono assegnate 5 variabili che saranno poi valorizzate nel momento in cui viene eseguito il codice all'interno del FB stesso. La logica di esecuzione è dall'alto verso il basso perciò risulta rilevante anche l'ordine in cui sono poste.

Figura 20: Logger manager

6.2 Implementazione buffer

Come già ampiamente discusso nel Capitolo 5 si è deciso di optare per l'utilizzo di una logica a doppio buffer la quale è stata implementata via software nella seguente maniera:

```
FUNCTION_BLOCK FB_Buffer_1
VAR_INPUT
  bStart            : BOOL;
  tLogCycleTime    : TIME;
  arData           : ARRAY [1..MAX_COLUMNS] OF INT;
  nColumns          : INT(1..MAX_COLUMNS);
END_VAR
VAR_OUTPUT
    (* := 0 -> nothing, 1 -> Buffer1 is active and 2 -> Buffer2 is active *)
  nBufferActive : INT:=0;
    bBusy        : BOOL;
  nRowActive     : INT;
  arBuffer1      : ARRAY [1..MAX_ROWS, 1..MAX_COLUMNS] OF LREAL;
  arBuffer2      : ARRAY [1..MAX_ROWS, 1..MAX_COLUMNS] OF LREAL;
END_VAR
VAR
  R_TRIG_0     : R_TRIG;
  F_TRIG_0     : F_TRIG;
    TON_0            : TON;
  t      : TIME:= T#0S;
  nColumn      : INT := 1;
  nRow       : INT := 0;
END_VAR
```

Listing 6: Dichiarazione variabili Buffer

```
R_TRIG_0(CLK:=bStart);
F_TRIG_0(CLK:=bStart);
TON_0(PT := tLogCycleTime);
t := t+(UDINT_TO_TIME(_TaskInfo[GETCURTASKINDEXEX()].CycleTime)/ 10000);

IF R_TRIG_0.Q THEN
  TON_0(IN:=TRUE);
  bBusy := TRUE;
  nBufferActive := 1;
  nRow := 0;
  t := T#0S;
END_IF;

IF F_TRIG_0.Q THEN
  TON_0(IN:=FALSE);
  bBusy := FALSE;
  nBufferActive := 0;
END_IF

IF TON_0.Q THEN
  nRow := nRow + 1;
  IF nRow > MAX_ROWS THEN
    nRow := 1;

    IF nBufferActive = 1 THEN
      nBufferActive := 2;
    ELSIF nBufferActive = 2 THEN
      nBufferActive := 1;
    END_IF

  END_IF

  IF nBufferActive = 1 THEN
    arBuffer1[nRow,1] := TIME_TO_LREAL(t);
    FOR nColumn := 2 TO nColumns+1 BY 1 DO
      arBuffer1[nRow,nColumn] := arData[nColumn-1];
    END_FOR;
  ELSIF nBufferActive = 2 THEN
    arBuffer2[nRow,1] := TIME_TO_LREAL(t);
    FOR nColumn := 2 TO nColumns+1 BY 1 DO
      arBuffer2[nRow,nColumn] := arData[nColumn-1];
    END_FOR;
  END_IF;

  nRowActive := nRow;

  TON_0(IN:=FALSE);
  TON_0(IN:=TRUE);
END_IF;
```

Listing 7: Codice Buffer

6.3 Implementazione logger

Per questa prima versione il codice del logger viene riportato in maniera integrale, per il capitolo successivo invece verranno solamente riportate le modifiche effettuate fino alla quinta versione di data-logger. Infatti anche per l'ultima versione di data-logger verrà riportato il codice completo.

```
FUNCTION_BLOCK FB_CustomLogger_1
VAR_INPUT
  bStart              : BOOL;
  stParams            : ST_CUSTOMLOGGER_PARAMS;
  arDataBuffer1           : ARRAY[1..MAX_ROWS,1..MAX_COLUMNS] OF LREAL;
  arDataBuffer2           : ARRAY[1..MAX_ROWS,1..MAX_COLUMNS] OF LREAL;
  nRowActive            : INT;
  nBufferActive : INT:=0; // := 0 nothing, 1 Buffer1 e 2 Buffer2 is active
               // := 1 Buffer2 is active
END_VAR
VAR_OUTPUT
  bError              : BOOL;
    eErrorId              : E_CUSTOMLOGGER_ERRORS;
    bFileOpen           : BOOL;
    bFileClosed           : BOOL;
  bBusy                   : BOOL;
  eLoggerState          : E_CUSTOMLOGGER_STATES;
END_VAR
VAR
// Start and Stop trigger
  R_TRIG_0            : R_TRIG;
  R_TRIG_1            : R_TRIG;

// Open, csvBuffer, Write and Close FB
  FB_FileOpen_0         : FB_FileOpen;
  FB_FileClose_0          : FB_FileClose;
  FB_BufferWriter_0       : FB_CSVMemBufferWriter;
  FB_FileWrite_0          : FB_FileWrite;

//This FB take the current time for file name
  NT_GetTime_0          : NT_GetTime;
  arDataBuffer            : ARRAY[1..MAX_ROWS,1..MAX_COLUMNS] OF LREAL;
  nRowToWrite           : INT;


//General Variable
  eState              : E_CUSTOMLOGGER_STATES := E_CUSTOMLOGGER_STATES.eSTATE_IDLE; //The log
    start in IDLE state
  sPathName           : T_MaxString;
  sCSVField           : STRING(MAX_STRING_LENGTH);
  hFile             : UINT;
  nColumn             : INT;
  nRow              : INT(0..MAX_ROWS);
  F_TRIG_1            : F_TRIG;
  nRowToLog           : INT;
  bFirstLog           : BOOL := TRUE;
  bLastLog            : BOOL := FALSE;
  address             : POINTER TO ARRAY [1..10] OF STRING;
  sCSVLine            : ARRAY[1..1000000] OF BYTE;
  nFileDimension_MByte      : LREAL;
  nFileDimension_Byte       : ULINT;
  nSaveCounter          : INT;
  bTimeExpired          : BOOL;
  bSegmentationBySize       : BOOL;
  bMaxSizeReached         : BOOL;
  sValueToWrite: STRING;
  nActualRow: INT;
  nLastBufferActive: INT;
  R_TRIG_2: R_TRIG;
END_VAR
```

Listing 8: Dichiarazione variabili Logger

```
R_TRIG_0(CLK:=nBufferActive = 2 AND nLastBufferActive=1);
R_TRIG_1(CLK:=nBufferActive = 1 AND nLastBufferActive=2);

F_TRIG_1(CLK:=bStart);
R_TRIG_2(CLK:=bStart);

IF R_TRIG_2.Q THEN
  bFirstLog:=TRUE;
  bLastLog:=FALSE;
END_IF

//Check if any type of segmentation has been chosen
IF stParams.tSegmentationTime > T#0S AND bStart AND stParams.nSegmentationDimension = 0 THEN
  T_Timer(IN:= TRUE, PT:= stParams.tSegmentationTime);
ELSIF stParams.nSegmentationDimension <> 0 AND stParams.tSegmentationTime = T#0S  AND stParams.nSegmentationDimension < nFileDimension_MByte THEN
  nFileDimension_Byte := 0;
  nFileDimension_MByte := 0;
  nSaveCounter := nSaveCounter+1;
  sPathName := '';
  bMaxSizeReached := TRUE;
ELSIF stParams.tSegmentationTime > T#0S  AND stParams.nSegmentationDimension > 0 OR stParams.tSegmentationTime < T#0S OR stParams.nSegmentationDimension < 0 THEN
  eErrorId := E_CUSTOMLOGGER_ERRORS.eERROR_INVALID_SEGMENTATION_PARAM;
  eState := E_CUSTOMLOGGER_STATES.eSTATE_ERROR;
END_IF

//This trigger detects the stop event
IF F_TRIG_1.Q THEN
  IF nBufferActive = 2 THEN
    arDataBuffer := arDataBuffer2;
  ELSE
    arDataBuffer := arDataBuffer1;
  END_IF
  nRowToLog := nRowActive;
  nActualRow := 1;
  nRow := 1;
  IF bFirstLog THEN
    eState := E_CUSTOMLOGGER_STATES.eSTATE_ACQUIRING_TIME;
  ELSE
    eState := E_CUSTOMLOGGER_STATES.eSTATE_FILEDATA_PROCESSING;
  END_IF
  bLastLog := TRUE;
END_IF

//this trigger detects that buffer 1 is full and ready to be written
IF R_TRIG_0.Q THEN
  bBusy := TRUE;
  arDataBuffer := arDataBuffer1;
  IF bStart AND bFirstLog THEN
    eState := E_CUSTOMLOGGER_STATES.eSTATE_ACQUIRING_TIME;
  ELSIF NOT bFirstLog THEN
    eState := E_CUSTOMLOGGER_STATES.eSTATE_FILEDATA_PROCESSING;
  END_IF
  nRowToLog := MAX_ROWS;
  nActualRow:=1;
  nRow := 1;
END_IF

//This trigger detects that buffer 2 is full and ready to be written
IF R_TRIG_1.Q THEN
  bBusy := TRUE;
  arDataBuffer := arDataBuffer2;
  IF bStart AND bFirstLog THEN
    eState := E_CUSTOMLOGGER_STATES.eSTATE_ACQUIRING_TIME;
  ELSIF NOT bFirstLog THEN
    eState := E_CUSTOMLOGGER_STATES.eSTATE_FILEDATA_PROCESSING;
  END_IF
  nRowToLog := MAX_ROWS;
  nActualRow:=1;
  nRow := 1;
END_IF

//This trigger performs the segmentation by time
IF T_Timer.Q  THEN
```

```
74    T_Timer(IN:=FALSE);
75    nSaveCounter := nSaveCounter +1;
76    sPathName := '';
77    bTimeExpired := TRUE;
78  END_IF
79
80  //This case statement will handle the sequencing of the writing
81  //===============================================================
82  CASE eState OF
83    E_CUSTOMLOGGER_STATES.eSTATE_ERROR: //Error State
84      bError := TRUE;
85      bBusy := FALSE;
86
87    E_CUSTOMLOGGER_STATES.eSTATE_IDLE:    // IDLE (Wait for the start trigger)
88      bBusy := FALSE;
89
90    E_CUSTOMLOGGER_STATES.eSTATE_ACQUIRING_TIME: //Acquire the time
91      bError := FALSE;
92      eErrorId := E_CUSTOMLOGGER_ERRORS.eERROR_NOERROR;
93      NT_GetTime_0(START := FALSE);
94      NT_GetTime_0(NETID:= stParams.sNetId, START := TRUE);
95      eState := E_CUSTOMLOGGER_STATES.eSTATE_WAITING_ACQUISITION;
96
97    E_CUSTOMLOGGER_STATES.eSTATE_WAITING_ACQUISITION: //Check if the time was acquired
98      NT_GetTime_0(START := FALSE);
99      IF NOT NT_GetTime_0.BUSY AND NOT NT_GetTime_0.ERR THEN
100       NT_GetTime_0(START := FALSE);
101       eState := E_CUSTOMLOGGER_STATES.eSTATE_FILENAME_AND_PATH_CREATION;

102     END_IF
103
104   E_CUSTOMLOGGER_STATES.eSTATE_FILENAME_AND_PATH_CREATION: // Construct the path and file name
105
106       sPathName := CONCAT(CONCAT(stParams.sFilePath, stParams.sFileName),'_');
107     sPathName := CONCAT(sPathName, WORD_TO_STRING(NT_GetTime_0.TIMESTR.wYear));
108     sPathName := CONCAT(sPathName,'-');
109     sPathName := CONCAT(sPathName, WORD_TO_STRING(NT_GetTime_0.TIMESTR.wMonth));
110     sPathName := CONCAT(sPathName,'-');
111     sPathName := CONCAT(sPathName, WORD_TO_STRING(NT_GetTime_0.TIMESTR.wDay));
112     sPathName := CONCAT(sPathName,'_');
113     sPathName := CONCAT(sPathName, WORD_TO_STRING(NT_GetTime_0.TIMESTR.wHour));
114     sPathName := CONCAT(sPathName,'.');
115     sPathName := CONCAT(sPathName, WORD_TO_STRING(NT_GetTime_0.TIMESTR.wMinute));
116     sPathName := CONCAT(sPathName,'.');
117     sPathName := CONCAT(sPathName, WORD_TO_STRING(NT_GetTime_0.TIMESTR.wSecond));
118     sPathName := CONCAT(CONCAT(sPathName, '_'),INT_TO_STRING(nSaveCounter));
119     sPathName := CONCAT(sPathName,'.csv');
120
121     eState:= E_CUSTOMLOGGER_STATES.eSTATE_OPENING_FILE;
122
123   E_CUSTOMLOGGER_STATES.eSTATE_OPENING_FILE: //Create and open the file
124     FB_FileOpen_0(bExecute := FALSE );
125      FB_FileOpen_0(sNetId := stParams.sNetId, sPathName := sPathName, nMode := FOPEN_MODEWRITE
      OR FOPEN_MODEBINARY,
126                         ePath := PATH_GENERIC, bExecute := TRUE );
127     eState:= E_CUSTOMLOGGER_STATES.eSTATE_WAITING_FILE_OPENING;
128
129   E_CUSTOMLOGGER_STATES.eSTATE_WAITING_FILE_OPENING: //Make sure the FB is free and there isn'
      t error
130     FB_FileOpen_0( bExecute := TRUE, hFile => hFile );
131     IF NOT FB_FileOpen_0.bBusy AND NOT FB_FileOpen_0.bError THEN
132       bFileOpen := TRUE;
133       bFileClosed := FALSE;
134       FB_FileOpen_0(bExecute := FALSE);
135       eState := E_CUSTOMLOGGER_STATES.eSTATE_FILEHEADER_CREATION;
136     ELSIF FB_FileOpen_0.bError THEN
137       eErrorId := E_CUSTOMLOGGER_ERRORS.eERROR_FILEOPEN;
138       eState := E_CUSTOMLOGGER_STATES.eSTATE_ERROR;
139     END_IF
140
141   E_CUSTOMLOGGER_STATES.eSTATE_FILEHEADER_CREATION: // Create the Header of file
142     FB_BufferWriter_0.eCmd := eEnumCmd_First;
143     sCSVField := 'ms';
144     FOR nColumn := 1 TO stParams.nNumberOfColumns+1 BY 1 DO
145       IF nColumn < stParams.nNumberOfColumns+1 THEN
```

```
146         FB_BufferWriter_0(pBuffer := ADR( sCSVLine ), cbBuffer := SIZEOF( sCSVLine ), putValue
      := sCSVField,
147                   pValue := 0, cbValue := 0,  bCRLF := FALSE);
148       ELSE
149         FB_BufferWriter_0(pBuffer := ADR( sCSVLine ), cbBuffer := SIZEOF( sCSVLine ), putValue
      := sCSVField,
150                   pValue := 0, cbValue := 0, bCRLF := TRUE);  (* bCRLF == TRUE => Write CRLF
    after the last field value *)
151       END_IF
152       FB_BufferWriter_0.eCmd := eEnumCmd_Next;
153       sCSVField := stParams.arColumnHeadings[nColumn];
154     END_FOR
155
156     eState := E_CUSTOMLOGGER_STATES.eSTATE_WRITING_FILE;
157
158
159   E_CUSTOMLOGGER_STATES.eSTATE_FILEDATA_PROCESSING: //Create the buffer of data to write
    inside the file
160
161     WHILE nRow <= 2 AND nActualRow < nRowToLog+1 DO
162
163       sCSVField := '';
164
165       FOR nColumn := 1 TO stParams.nNumberOfColumns+1 BY 1 DO // one iteration to add ms and
    one iteration to add CRLF
166
167         sValueToWrite := LREAL_TO_STRING(arDataBuffer[nActualRow, nColumn]);
168
169         IF LEN(sCSVField) + LEN(sValueToWrite) < UDINT_TO_DINT(MAX_STRING_LENGTH) THEN // AND
    nColumn < stParams.nNumberOfColumns+1 THEN
170           sCSVField := CONCAT(sCSVField, sValueToWrite);
171           sCSVField := CONCAT(sCSVField, ';');
172         ELSE
173           FB_BufferWriter_0(pBuffer := ADR( sCSVLine ), cbBuffer := SIZEOF( sCSVLine ),
    putValue := sCSVField, pValue := 0, cbValue := 0,
174                     bCRLF := FALSE);
175           nColumn := nColumn-1;
176           sCSVField := '';
177           FB_BufferWriter_0.eCmd := eEnumCmd_Next;
178         END_IF
179       END_FOR
180       FB_BufferWriter_0(pBuffer := ADR(sCSVLine), cbBuffer := SIZEOF(sCSVLine), putValue :=
    sCSVField, pValue := 0, cbValue := 0,
181                 bCRLF := TRUE); // bCRLF == TRUE => Write CRLF after the last field value
182       FB_BufferWriter_0.eCmd := eEnumCmd_Next;
183       nActualRow := nActualRow + 1;
184       nRow := nRow + 1;
185
186     END_WHILE
187
188     nRow := 1;
189     IF nActualRow = nRowToLog+1 THEN
190       eState := E_CUSTOMLOGGER_STATES.eSTATE_WRITING_FILE;
191       nActualRow := 1;
192     ELSE
193       eState := E_CUSTOMLOGGER_STATES.eSTATE_FILEDATA_PROCESSING;
194     END_IF
195
196
197
198   E_CUSTOMLOGGER_STATES.eSTATE_WRITING_FILE:  // Write the buffer into the file
199
200     FB_FileWrite_0( sNetId := stParams.sNetId, hFile := hFile, bExecute := TRUE, pWriteBuff :=
      ADR(sCSVLine), cbWriteLen := FB_BufferWriter_0.cbSize);
201     IF NOT FB_FileWrite_0.bBusy AND NOT FB_FileWrite_0.bError THEN
202       FB_FileWrite_0(bExecute:= FALSE);
203       nFileDimension_Byte := nFileDimension_Byte + FB_BufferWriter_0.cbSize;
204       nFileDimension_MByte := ULINT_TO_LREAL(nFileDimension_Byte)/EXPT(2,20);
205       IF (bLastLog AND NOT bFirstLog) OR bTimeExpired OR bMaxSizeReached THEN
206         eState := E_CUSTOMLOGGER_STATES.eSTATE_CLOSING_FILE;
207       ELSIF bFirstLog THEN
208         eState := E_CUSTOMLOGGER_STATES.eSTATE_FILEDATA_PROCESSING;
209         bFirstLog := FALSE;
210
211       ELSE
```

```
212         eState := E_CUSTOMLOGGER_STATES.eSTATE_IDLE;
213       END_IF
214 
215     FB_BufferWriter_0.eCmd := eEnumCmd_First;
216 
217     ELSIF FB_FileWrite_0.bError THEN
218       bError := TRUE;
219       eErrorId := E_CUSTOMLOGGER_ERRORS.eERROR_FILEWRITE;
220       eState := E_CUSTOMLOGGER_STATES.eSTATE_CLOSING_FILE;
221     END_IF
222 
223 
224   E_CUSTOMLOGGER_STATES.eSTATE_CLOSING_FILE: //Close File
225     FB_FileClose_0(sNetId := stParams.sNetId, bExecute := TRUE, hFile:= hFile);
226     IF NOT FB_FileClose_0.bBusy AND NOT FB_FileClose_0.bError THEN
227       bFileOpen := FALSE;
228       bFileClosed := TRUE;
229         FB_FileClose_0(bExecute := FALSE);
230       IF bTimeExpired OR bMaxSizeReached THEN
231         bTimeExpired := FALSE;
232         bMaxSizeReached := FALSE;
233         eState := E_CUSTOMLOGGER_STATES.eSTATE_FILENAME_AND_PATH_CREATION;
234       ELSE
235         eState:= E_CUSTOMLOGGER_STATES.eSTATE_IDLE;
236       END_IF
237     END_IF
238 
239 END_CASE
240 eLoggerState := eState;
241 nLastBufferActive := nBufferActive;
```

Listing 9: Codice Logger

6.4 Analisi delle prestazioni

Per l'analisi delle prestazioni si sono fatti registrare 100 canali al millisecondo in modo da mettere sotto stress il logger, i risultati come ci si aspettava, non sono stati dei più soddisfacenti in quanto, come riportato in figura 24, si sono riscontrate delle grosse perdite di dati unite a dei picchi di tempo ciclo della CPU che nel punto più alto sforava i 6 ms. Non considerando i picchi, che comunque rappresentano un problema, il tempo ciclo di regime era superiore ai 4 ms (4 volte il tempo ciclo imposto pari a 1 ms).
Nell'immagine sottostante vengono riportati i valori registrati dallo scope durante il funzionamento del logger che sono:

- **bStart**: Rappresentato in blu indica per quanto tempo il data-logger ha campionato dati. Per tutte quante le versioni di data logger è stato utilizzato un tempo pari a 30 secondi.
- **eLoggerState**: Rappresentato in verde lime indica lo stato in cui il data-logger si è venuto a trovare nei 30 secondi, è possibile notare negli istanti iniziali un picco verso il basso che non è altro che il passaggio rapido del data loger attraverso i primi stati (dallo 0 al 25) nei quali crea ed apre in scrittura un nuovo file *.csv. Dopodiché non ci sarà altro che l'alternarsi dello stato 99 di IDLE, 35 (stato in cui avviene la preparazione per la scrittura) che si è rivelato essere il più pesante e lungo e lo stato 40 (stato in cui avviene l'effettiva scrittura). Infine si passerà allo stato 45 per la chiusura del file.
- **bFileOpen**, **bFileClosed**: Rappresentati rispettivamente in arancione e in verde chiaro indicano quando il file viene aperto o chiuso.
- **LastExecTime**:Rappresentato in marrone descrive il tempo richiesto dal processore per il completamento dell'ultimo ciclo espresso in millisecondi.

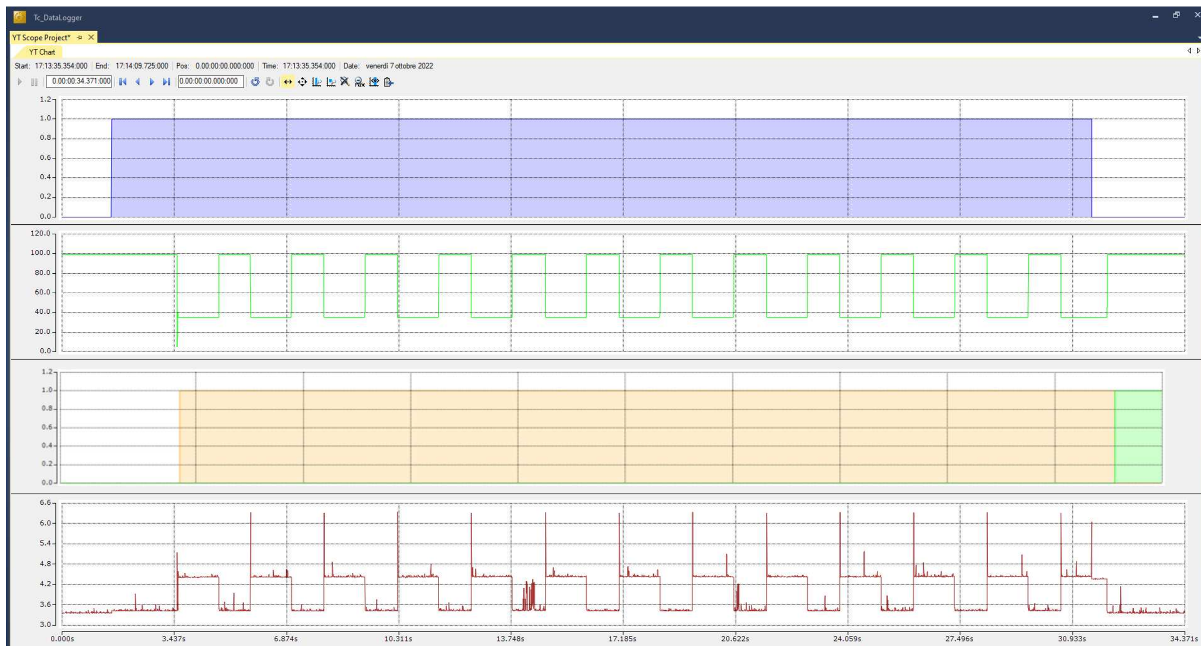


Figura 21: Grafico generale prima versione

Inoltre nelle figure sottostanti viene fatto un focus sui valori di picco e regime della variabile LastExecTime a testimonianza di quanto detto sopra.

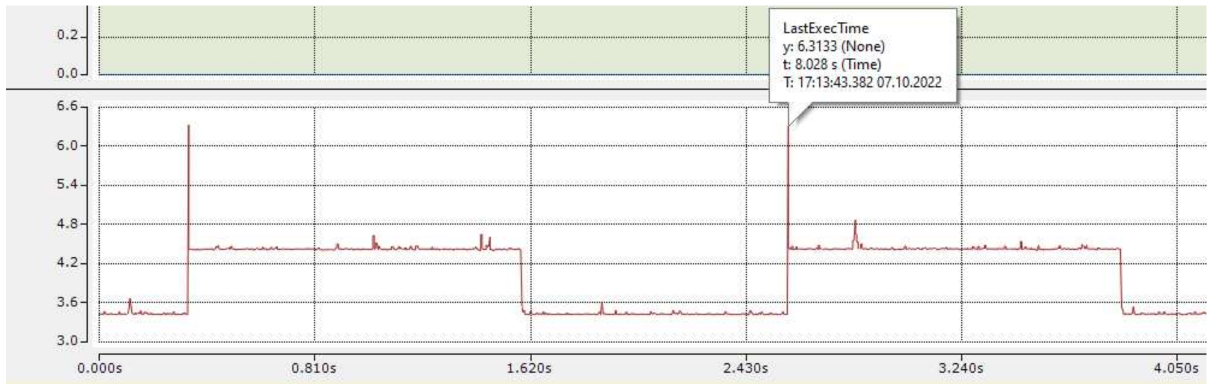


Figura 22: Valore di picco della prima versione

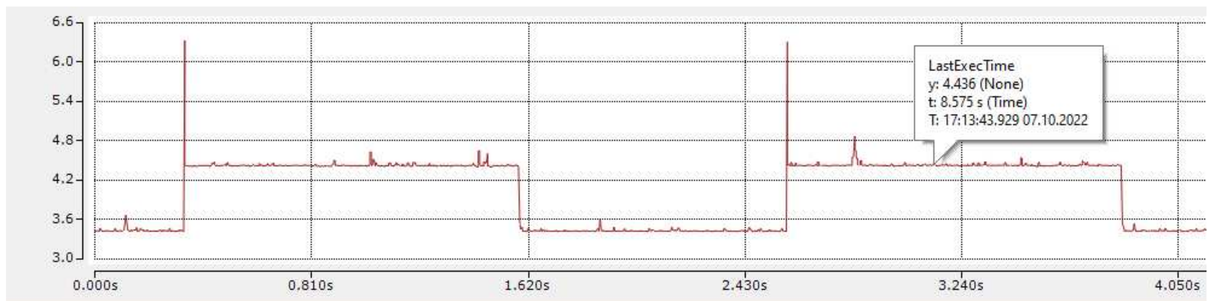


Figura 23: Valore di regime della prima versione

Alla fine dei 30 secondi viene prodotto un file *.csv, aprendolo ci si può accorgere che queste continue deviazioni temporali non hanno fatto altro che far perdere dati in fase di buffering e scrittura. Nella figura sottostante viene riportato un piccolo estratto del file dove si può vedere nella colonna di sinistra che il tempo invece che progredire 1 ms alla volta progredisce di 4 ms.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA
1	ms	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
2	T#4ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
3	T#8ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
4	T#12ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
5	T#16ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
6	T#20ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
7	T#24ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
8	T#28ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
9	T#32ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
10	T#36ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
11	T#40ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
12	T#44ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
13	T#48ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
14	T#52ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
15	T#56ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
16	T#60ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
17	T#64ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
18	T#68ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
19	T#72ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
20	T#76ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
21	T#80ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
22	T#84ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
23	T#88ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
24	T#92ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
25	T#96ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
26	T#100ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6

Figura 24: Estratto del file *.csv generato dal primo logger

7 Ottimizzazione del software

Partendo ora dalla prima versione di logger "funzionante" si è passati all' analizzare quali fossero le sue criticità e da cosa fossero causate. Mano a mano si è proceduto nel risolverle fino ad arrivare alla quarta versione che si poteva considerare quella definitiva, infatti per quest'ultimo i requisiti funzionali era soddisfatti, mancava solamente un' HMI. La quinta versione è stata sviluppata solo per rendere più modulare il codice e per separare nuovamente buffer e logger in modo da poter inserire in futuro le calibrazioni.

7.1 Analisi dei limiti e delle inefficienze del primo logger

Partendo dallo scope in figura 21 è stato possibile fin da subito notare che il picco si manifestava nello stato 35 ovvero nello stato in cui il FB_CSVMemBuffer è preparato per essere scritto.
Un ulteriore problema risiedeva anche nel fatto che, come è stato detto più volte, le operazioni di copiatura e assegnazione delle matrici risultano parecchio pesanti e questa operazione così dispendiosa veniva fatta in più punti che erano:

- All'interno del PRG_LoggerManager_1 in figura 20 più precisamente alla riga 3 e 8 dove è possibile vedere le matrici arDataBuffer1 e arDataBuffer2 come output a cui vengono assegnate le due prodotte dal buffer, mentre in riga 8 vengono assegnate al FB_CustomLogger_1 che si occuperà poi di gestirle per la scrittura.
- Alla riga 29,31,47 e 61 del *Listing* 9 dove viene determinato il buffer da scrivere su file *.csv .

Un'altra operazione che si è rivelata particolarmente pesante inaspettatamente è stata quella di concatenazione delle stringhe, infatti si vedrà che la sue eliminazione giocherà un ruolo fondamentale nel passaggio dalla terza alla quarta versione di data logger.
Una volta individuate queste debolezze si è proceduto alla loro mitigazione mediante l'utilizzo di puntatori, unificazione di buffer e logger ed eliminazione delle concatenazioni.

7.2 Progettazione ed implementazione del secondo data-logger

Al fine di ridurre il numero di volte in cui le matrici venivano copiate e assegnate si è proceduto con l'unificazione dei due blocchi funzionali Buffer e Logger. Ora essendo i due blocchi contenuti all'interno dello stesso codice ST esse condividono anche le variabili eliminando il bisogno di dover compiere quello scambio che veniva fatto all'interno del PRG_Logger_Manager_1 . In questa versione permane comunque l'assegnazione all'interno del codice del logger.

Analisi prestazioni

Dopo aver unificato i due codici, aver rinominato alcune variabili e aver apportato alcune piccole modifiche, si è proceduto con i test per l'analisi delle prestazioni che hanno prodotto i risultati riportati in figura 25. I test sono stati eseguiti sempre impostando 100 canali/ms .

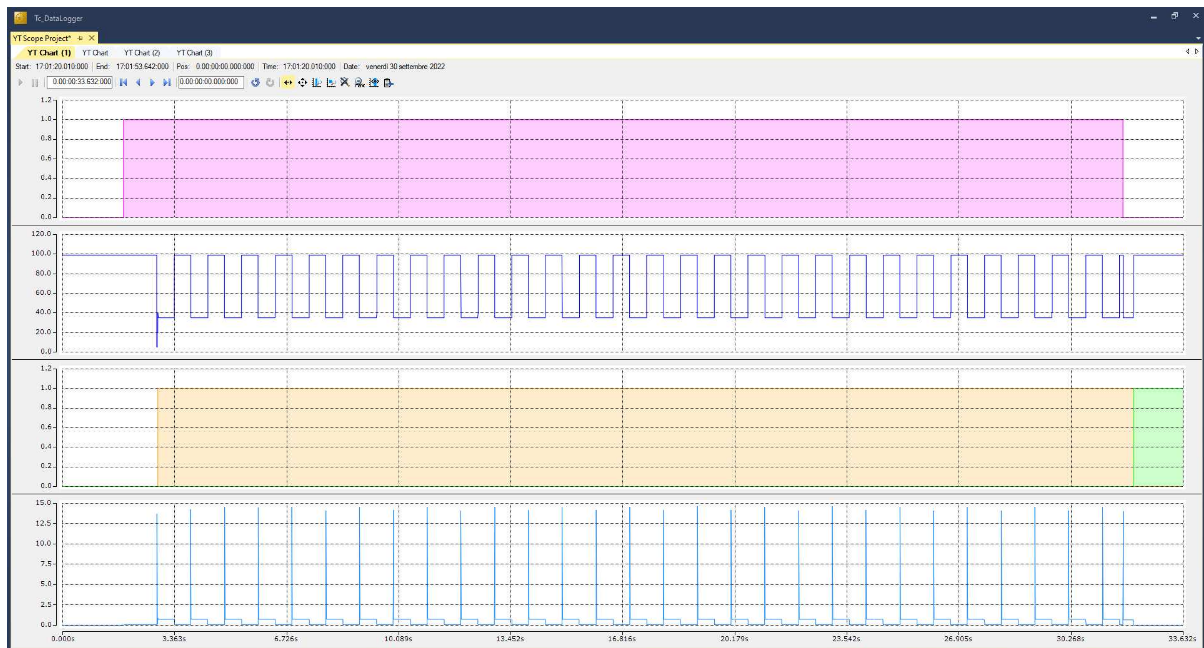


Figura 25: Grafico generale seconda versione

Anche in questo caso sono stati riportati dei focus sui valori di picco e regime della variabile LastExecTime

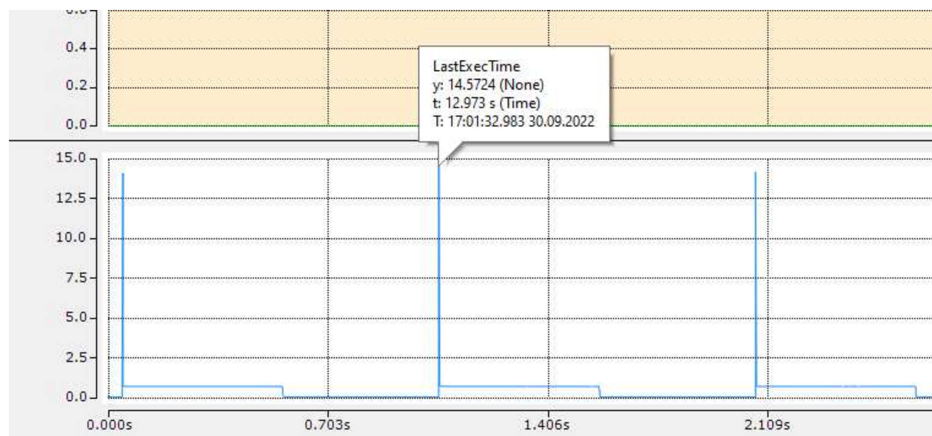


Figura 26: Valore di picco della seconda versione

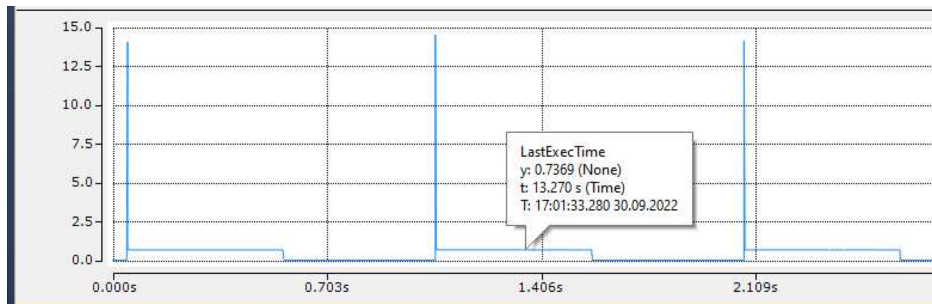


Figura 27: Valore di regime della seconda versione

Come si può notare dalla figura 26 in questa versione si ha un peggioramento del picco che passa da circa 6.2 ms a 14.6 ms ma, come mostrato in figura 27, si ha un netto miglioramento del valore di regime che arriva a stare al di sotto del millisecondo. Questo risultato è molto importante dal punto di vista di affidabilità del data logger in quanto alla fine dei 30 secondi viene prodotto un file *.csv al cui interno tutti i valori sono correttamente campionati e scritti, senza alcuna perdita. In figura 28 viene riportato un estratto del file dove nella colonna di sinistra è possibile vedere che i valori sono campionati correttamente ogni millisecondo.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA
1	ms	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
2	T#1ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
3	T#2ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
4	T#3ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
5	T#4ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
6	T#5ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
7	T#6ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
8	T#7ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
9	T#8ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
10	T#9ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
11	T#10ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
12	T#11ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
13	T#12ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
14	T#13ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
15	T#14ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
16	T#15ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
17	T#16ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
18	T#17ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
19	T#18ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
20	T#19ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
21	T#20ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
22	T#21ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
23	T#22ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
24	T#23ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
25	T#24ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
26	T#25ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
27	T#26ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
28	T#27ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
29	T#28ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
30	T#29ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
31	T#30ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6

Figura 28: Estratto del file *.csv generato dal secondo logger

7.3 Progettazione ed implementazione del terzo data-logger

Pur avendo unito buffer e logger, rimaneva ancora all'interno del codice inerente al logger un'assegnazione tra la matrice arDataBuffer e uno dei due buffer ('arDataBuffer1' o 'arDataBuffer2') dove vengono accodati i valori da andare a trasferire nel blocchetto FB_CSVMemBufferWriter. Se non si fosse fatto così si sarebbe dovuto sdoppiare il codice controllando con alcuni 'if' quale buffer doveva essere scritto.
Alla luce di ciò è facile capire che l'arDataBuffer è sostanzialmente usato per aumentare la leggibilità e la pulizia del codice. La soluzione che è stata pensata per evitare anche questa assegnazione è quella di usare l'arDataBuffer come puntatore ad un indirizzo di memoria anziché come matrice vera e propria. L'inidirizzo di memoria a cui punterà sarà quello contenete una delle due matrici a seconda della necessità. In questo modo si è riusci a mantenere la pulizia del codice e ad aumentare le prestazioni.

Modifiche software

A livello software si è passati da

```
(* Nella dichiarazione delle variabili *)

arDataBuffer    : ARRAY[1..MAX_ROWS,1..MAX_COLUMNS] OF STRING;

(* All'interno del Buffer *)

sValueToWrite   := arDataBuffer[nActualRow,nColumn];

IF nBufferActive = 2 THEN
    arDataBuffer := arDataBuffer2;
ELSE
    arDataBuffer := arDataBuffer1;
END IF
```

A quanto riportato sotto

```
(* Nella dichiarazione delle variabili *)

arDataBuffer    : POINTER TO ARRAY[1..MAX_ROWS,1..MAX_COLUMNS] OF STRING;

(* All'interno del Buffer *)

sValueToWrite   := arDataBuffer^[nActualRow,nColumn];

IF nBufferActive = 2 THEN
    arDataBuffer := ADR(arDataBuffer2);
ELSE
    arDataBuffer := ADR(arDataBuffer1);
END IF
```

Analisi prestazioni

Dopo aver apportato le modifiche sopra illustrate si è proceduto con la solita analisi prestazionale a 100 canali/ms.

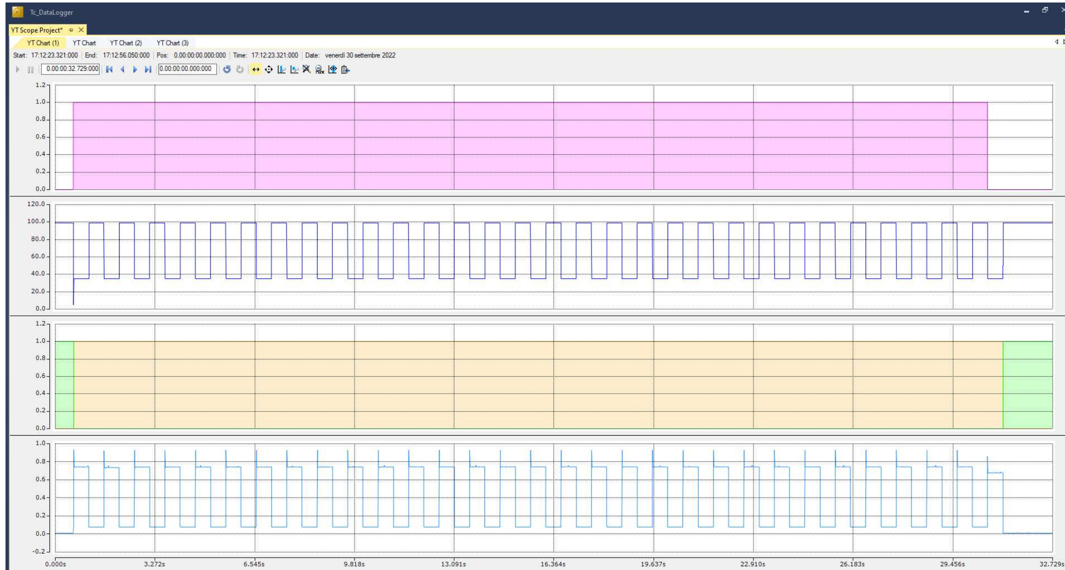


Figura 29: Grafico generale terza versione

Anche in questo caso si è riportato un focus sui valori di picco e regime della variabile LastExecTime

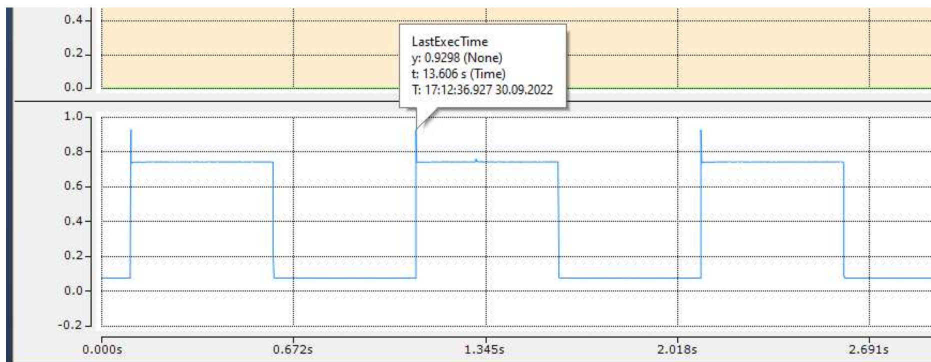


Figura 30: Valore di picco della terza versione

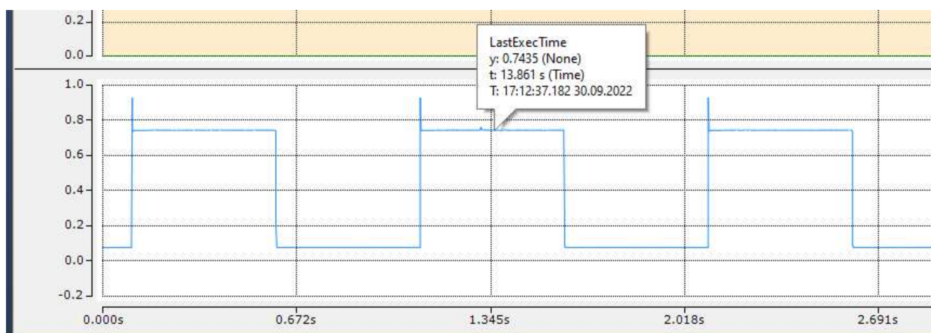


Figura 31: Valore di regime della terza versione

Come si può notare dalla figura 30 in questa versione si ha un netto miglioramento del picco che passa da circa 14.6 ms a 1 ms, e come mostrato in figura 31, il valore di regime rimane pressoché invariato. Allo scadere dei 30 secondi di campionamento viene prodotto un file *.csv al cui interno tutti i valori sono correttamente campionati e scritti, senza alcuna perdita. In figura 32 viene riportato un estratto del file dove nella colonna di sinistra è possibile vedere che anche in questo caso i valori sono campionati correttamente ogni millisecondo.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA
1	ms	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
2	T#1ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
3	T#2ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
4	T#3ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
5	T#4ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
6	T#5ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
7	T#6ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
8	T#7ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
9	T#8ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
10	T#9ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
11	T#10ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
12	T#11ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
13	T#12ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
14	T#13ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
15	T#14ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
16	T#15ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
17	T#16ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
18	T#17ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
19	T#18ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
20	T#19ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
21	T#20ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
22	T#21ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
23	T#22ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
24	T#23ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
25	T#24ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
26	T#25ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
27	T#26ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
28	T#27ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
29	T#28ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
30	T#29ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6
31	T#30ms	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6

Figura 32: Estratto del file *.csv generato dal terzo logger

7.4 Progettazione ed implementazione del quarto data-logger

In questa quarta versione sviluppata, è stata "eliminata" la concatenazione all'interno dello stato 35 sostituendola con la scrittura diretta sul CSV_MEM_BUFFER. Con questa modifica ci si è accorti che nonostante le scritture sul blocco funzione aumentassero il tempo ciclo diminuiva ancora. Questo tentativo ha assodato la teoria iniziale che l'operazione di CONCAT() e più in generale l'operare con stringhe fossero operazioni altamente dispendiose a livello di CPU. In questa versione, essendo state notevolmente migliorate le performance del data-logger, si sono effettuati test sulle prestazioni non solo per 100 canali ma anche per 200 e 300 canali.

7.5 Modifiche software

A livello software si è agito solamente sullo stato 35 che è passato da

```
1  E_CUSTOMLOGGER_STATES.eSTATE_FILEDATA_PROCESSING: //Create the buffer of data to write inside
       the file
2
3      WHILE nRowLogger <= 2 AND nActualRow < nRowToLog+1 DO
4
5        sCSVField := '';
6
7        FOR nColumnLogger := 1 TO stParams.nNumberOfColumns+1 BY 1 DO // one iteration to add ms
         and one iteration to add CRLF
8
9          sValueToWrite := arDataBuffer^[nActualRow, nColumnLogger];
10
11
12         IF LEN(sCSVField) + LEN(sValueToWrite) < UDINT_TO_DINT(MAX_STRING_LENGTH) THEN
13           sCSVField := CONCAT(CONCAT(sCSVField, sValueToWrite), ';');
14         ELSE
15
16           FB_BufferWriter_0(pBuffer := ADR( sCSVLine ), cbBuffer := SIZEOF( sCSVLine ),
       putValue := sCSVField, pValue := 0,
17                       cbValue := 0, bCRLF := FALSE);
18           nColumnLogger := nColumnLogger-1;
19           sCSVField := '';
20           FB_BufferWriter_0.eCmd := eEnumCmd_Next;
21         END_IF
22       END_FOR
23
24       FB_BufferWriter_0.eCmd := eEnumCmd_Next;
25       nActualRow := nActualRow + 1;
26       nRowLogger := nRowLogger + 1;
27
28     END_WHILE
29
30     nRowLogger := 1;
31     IF nActualRow = nRowToLog+1 THEN
32       eState := E_CUSTOMLOGGER_STATES.eSTATE_WRITING_FILE;
33       nActualRow := 1;
34     ELSE
35       eState := E_CUSTOMLOGGER_STATES.eSTATE_FILEDATA_PROCESSING;
36     END_IF
```

A quanto riportato sotto

```
E_CUSTOMLOGGER_STATES.eSTATE_FILEDATA_PROCESSING: //Create the buffer of data to write inside
    the file

    WHILE nRowLogger <= 2 AND nActualRow < nRowToLog+1 DO

      FOR nColumnLogger := 1 TO stParams.nNumberOfColumns+1 BY 1 DO // one iteration to add ms
     and one iteration to add CRLF

        FB_BufferWriter_0(pBuffer := ADR( sCSVLine ), cbBuffer := SIZEOF( sCSVLine ),
                            putValue := arDataBuffer^[nActualRow, nColumnLogger], pValue := 0,
                 cbValue := 0, bCRLF := (nColumnLogger = stParams.nNumberOfColumns+1));

        FB_BufferWriter_0.eCmd := eEnumCmd_Next;

      END_FOR

      FB_BufferWriter_0.eCmd := eEnumCmd_Next;
      nActualRow := nActualRow + 1;
      nRowLogger := nRowLogger + 1;

    END_WHILE

    nRowLogger := 1;
    IF nActualRow = nRowToLog+1 THEN
      eState := E_CUSTOMLOGGER_STATES.eSTATE_WRITING_FILE;
      nActualRow := 1;
    ELSE
      eState := E_CUSTOMLOGGER_STATES.eSTATE_FILEDATA_PROCESSING;
    END_IF
```

Analisi prestazioni per 100 canali

Dopo aver apportato le modifiche sopra illustrate si è proceduto con la solita analisi prestazionale a 100 canali/ms, utile per fare il confronto con i risultati precedenti.

Figura 33: Grafico generale quarta versione

Anche in questo caso sono stati riportati dei un focus sui valori di picco e regime della variabile LastExecTime, solamente che in questo caso i valori di picco sono due, uno all'inizio dell'onda quadra e uno alla fine. Il picco finale comincia ad essere ben visibile superati i 200 canali e si ipotizza che questo comportamento sia anche dovuto ai 100 canali in più che il blocco funzionale FB_FileWriter deve scrivere ogni secondo, infatti facendo un semplice calcolo, considerando che una matrice è composta da 1000 righe e ci sono per ogni riga 100 colonne in più il blocco si ritrova a dover scrivere nella stessa quantità di tempo 100.000 celle in più.

Figura 34: Valore del primo picco della quarta versione a 100 canali

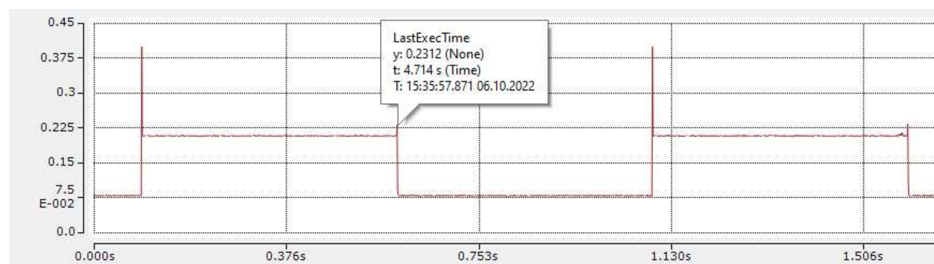


Figura 35: Valore del secondo picco della quarta versione a 100 canali

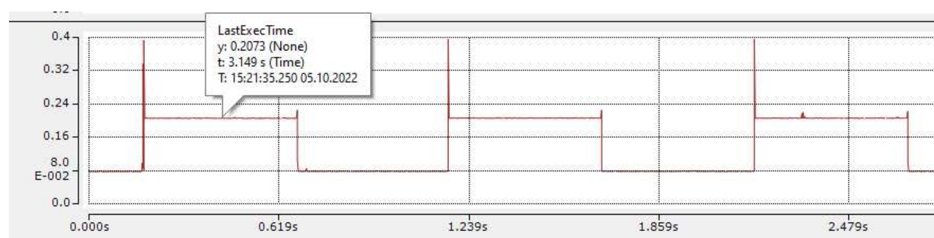


Figura 36: Valore del secondo picco della quarta versione a 100 canali

Come si può notare dalla figura 34 in questa versione si ha un netto miglioramento del primo picco che passa da circa 1 ms a 0.4 ms e, come mostrato in figura 36, il valore di regime passa da 0.7 ms a circa 0.2 ms. Infine in figura 35 è stato fornito anche un focus sul secondo picco che assume all'incirca un valore di 2.3 ms, un differenza quasi trascurabile rispetto al valore di regime. Allo scadere dei 30 secondi di campionamento viene prodotto un file *.csv al cui interno tutti i valori sono correttamente campionati e scritti, da ora in poi non verranno più mostrati estratti del file *.csv generato in quanto sono del tutto identici a quello mostrato in figura 32.

Analisi prestazioni per 200 canali

Essendo questa quarta versione nettamente più efficiente rispetto alle versioni precedenti si è scelto di provare a spingere ulteriormente sul numero di canali da registrare contemporaneamente ottenendo i seguenti risultati:

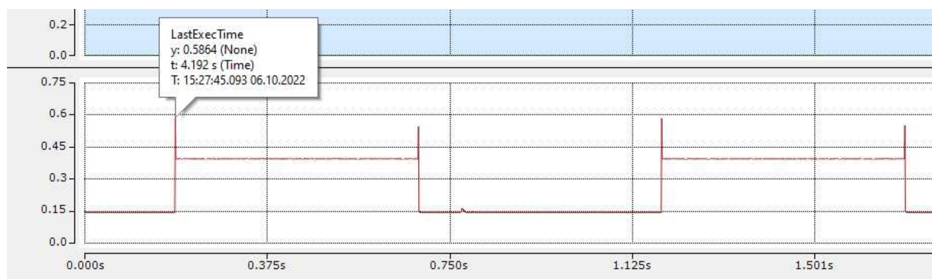


Figura 37: Valore del primo picco della quarta versione a 200 canali

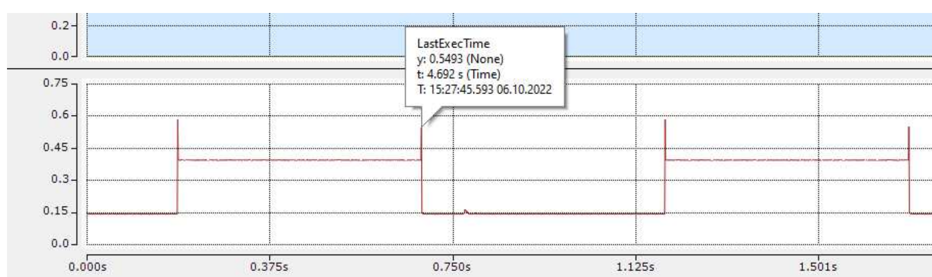


Figura 38: Valore del secondo picco della quarta versione a 200 canali

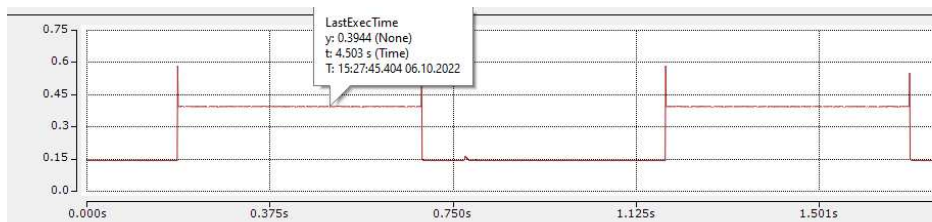


Figura 39: Valore di regime della quarta versione a 200 canali

Analizzando i risultati sopra esposti ci si può accorgere che il valore del secondo picco è aumentato se messo in proporzione rispetto al primo picco ma che comunque si è ancora lontani dal tempo ciclo massimo pari a 1 ms .

Analisi prestazioni per 300 canali

Per cercare di valutare il limite massimo di canali registrabili in un millisecondo dal data-logger si è provato a passare ad un numero pari a 300 canali al millisecondo. I risultati sono stati i seguenti:

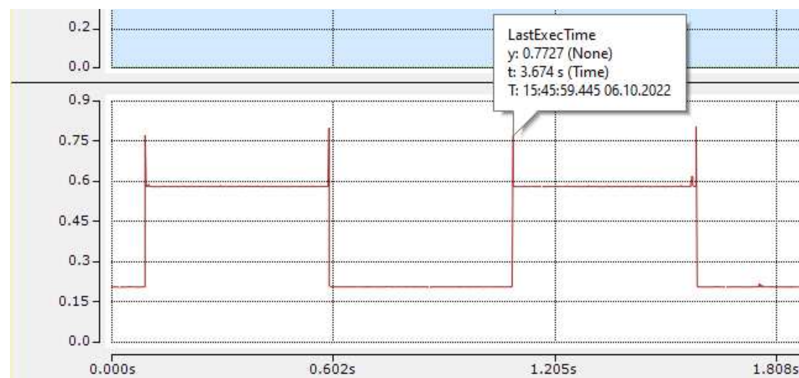


Figura 40: Valore del primo picco della quarta versione a 300 canali

Figura 41: Valore del secondo picco della quarta versione a 300 canali

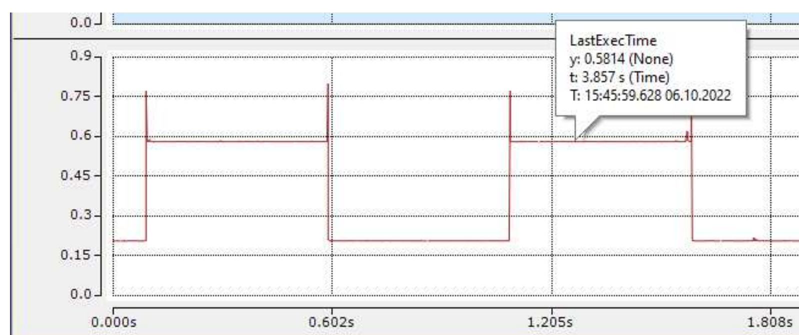


Figura 42: Valore di regime della quarta versione a 300 canali

Ricordando, come detto nel Capitolo 3, che i test sono stati fatti considerando un CPU limit pari all' 80 % ovvero 800 µs e considerando anche i due valori di picco in figura 40 e 41 si può dire che 300 canali costituisce il tetto massimo di campionamento della quarta versione di data-logger.

7.6 Progettazione ed implementazione del quinto data-logger

Avendo raggiunto e soddisfatto tutti i requisiti funzionali si è cercato poi di andare a dividere nuovamente la logica del buffer e del logger in maniera da rendere più modulare il programma. Questa modularità permetterà poi in futuro di poter aggiungere le calibrazioni(illustrate nel Capitolo 10) tra buffer e logger. In questo caso però, separare di nuovo buffer e logger, avrebbe riportato alla luce i problemi che erano stati risolti nella seconda versione di data-logger in cui erano state eliminate le matrici arDataBuffer1 e arDataBuffer2 per far comunicare i due moduli. Per ovviare a questo problema si è pensato di creare due PRG separati e farli comunicare tra di loro mediante la GVL, la quale è rappresentata nel *listing 4*. Inoltre per questa ultima versione come detto in precedenza è stata sviluppata una HMI che sarà discussa nell'apposito Capitolo 9.

Implementazione finale buffer

```
START_R_TRIG(CLK:= GVL_LOGGER.bStart);
STOP_F_TRIG(CLK:= GVL_LOGGER.bStart);
T_ON_LOG(PT := GVL_LOGGER.stParams.tLogCycleTime);
t := t+(UDINT_TO_TIME(_TaskInfo[GETCURTASKINDEXEX()].CycleTime)/ 10000);

IF START_R_TRIG.Q THEN
  T_ON_LOG(IN:=TRUE);
  GVL_LOGGER.nBufferActive := 1;
  nRowBuffer := 0;
  t := T#0S;
END_IF;

IF STOP_F_TRIG.Q THEN
  T_ON_LOG(IN:=FALSE);
END_IF

IF T_ON_LOG.Q THEN
  nRowBuffer := nRowBuffer + 1;
  IF nRowBuffer > MAX_ROWS THEN
    nRowBuffer := 1;

    IF GVL_LOGGER.nBufferActive = 1 THEN
      GVL_LOGGER.nBufferActive := 2;
    ELSIF GVL_LOGGER.nBufferActive = 2 THEN
      GVL_LOGGER.nBufferActive := 1;
    END_IF

  END_IF

  IF GVL_LOGGER.nBufferActive = 1 THEN
    GVL_LOGGER.arDataBuffer1[nRowBuffer,1] := TIME_TO_STRING(t);
    FOR nColumnBuffer := 2 TO GVL_LOGGER.stParams.nNumberOfColumns+1 BY 1 DO
      GVL_LOGGER.arDataBuffer1[nRowBuffer,nColumnBuffer] := INT_TO_STRING(GVL_LOGGER.arData[
    nColumnBuffer-1]);
    END_FOR;
  ELSIF GVL_LOGGER.nBufferActive = 2 THEN
    GVL_LOGGER.arDataBuffer2[nRowBuffer,1] := TIME_TO_STRING(t);
    FOR nColumnBuffer := 2 TO GVL_LOGGER.stParams.nNumberOfColumns+1 BY 1 DO
      GVL_LOGGER.arDataBuffer2[nRowBuffer,nColumnBuffer] := INT_TO_STRING(GVL_LOGGER.arData[
    nColumnBuffer-1]); //arData[nColumnBuffer-1];
    END_FOR;
  END_IF;

  GVL_LOGGER.nRowActive := nRowBuffer;

  T_ON_LOG(IN:=FALSE);
  T_ON_LOG(IN:=TRUE);
END_IF;
```

Implementazione finale logger

```
Buffer1_R_TRIG(CLK:= GVL_LOGGER.nBufferActive = 2 AND nLastBufferActive=1);
Buffer2_R_TRIG(CLK:= GVL_LOGGER.nBufferActive = 1 AND nLastBufferActive=2);

STOP_F_TRIG(CLK:= GVL_LOGGER.bStart);
START_R_TRIG(CLK:= GVL_LOGGER.bStart);

IF START_R_TRIG.Q THEN
  bFirstLog:=TRUE;
  bLastLog:=FALSE;
  nSaveCounter := 0;
  nFileDimension_Byte := 0;
  nFileDimension_MByte := 0;
END_IF

//Check if any type of segmentation has been chosen
IF GVL_LOGGER.stParams.tSegmentationTime > T#0S AND GVL_LOGGER.bStart AND GVL_LOGGER.stParams.
    nSegmentationDimension = 0 THEN
  T_Segmentation_Time(IN:= TRUE, PT:= GVL_LOGGER.stParams.tSegmentationTime);
ELSIF GVL_LOGGER.stParams.nSegmentationDimension <> 0 AND GVL_LOGGER.stParams.
    tSegmentationTime = T#0S  AND GVL_LOGGER.stParams.nSegmentationDimension <
    nFileDimension_MByte THEN
  nFileDimension_Byte := 0;
  nFileDimension_MByte := 0;
  nSaveCounter := nSaveCounter+1;
  sPathName := '';
  bMaxSizeReached := TRUE;
ELSIF GVL_LOGGER.stParams.tSegmentationTime > T#0S  AND GVL_LOGGER.stParams.
    nSegmentationDimension > 0 OR GVL_LOGGER.stParams.tSegmentationTime < T#0S OR GVL_LOGGER.
    stParams.nSegmentationDimension < 0 THEN
  GVL_LOGGER.eErrorId := E_CUSTOMLOGGER_ERRORS.eERROR_INVALID_SEGMENTATION_PARAM;
  eState := E_CUSTOMLOGGER_STATES.eSTATE_ERROR;
END_IF

IF STOP_F_TRIG.Q THEN
  bStop := TRUE;
END_IF

//This trigger detects the stop event
IF bStop AND nActualRow = 1 AND NOT FB_FileWrite_0.bBusy THEN // Se viene richiesto lo stop
    del logger prima di scrivere l'ultimo buffer si assicura che il penultimo sia stato
    scritto completamente
  bStop := FALSE;
  IF GVL_LOGGER.nBufferActive = 2 THEN
    arDataBuffer := ADR(GVL_LOGGER.arDataBuffer2);
  ELSIF GVL_LOGGER.nBufferActive = 1 THEN
    arDataBuffer := ADR(GVL_LOGGER.arDataBuffer1);
  END_IF
  nRowToLog := GVL_LOGGER.nRowActive;
  nActualRow := 1;
  nRow := 1;
  IF bFirstLog THEN
    eState := E_CUSTOMLOGGER_STATES.eSTATE_ACQUIRING_TIME;
  ELSE
    eState := E_CUSTOMLOGGER_STATES.eSTATE_FILEDATA_PROCESSING;
  END_IF
  bLastLog := TRUE;
END_IF

//this trigger detects that buffer 1 is full and ready to be written
IF Buffer1_R_TRIG.Q THEN
  GVL_LOGGER.bBusy := TRUE;
  arDataBuffer := ADR(GVL_LOGGER.arDataBuffer1);

  IF GVL_LOGGER.bStart AND bFirstLog THEN //bStartLog
    eState := E_CUSTOMLOGGER_STATES.eSTATE_ACQUIRING_TIME;
  ELSIF NOT bFirstLog THEN
    eState := E_CUSTOMLOGGER_STATES.eSTATE_FILEDATA_PROCESSING;
  END_IF
  nRowToLog := MAX_ROWS;
  nActualRow:=1;
  nRow := 1;
END_IF

```

```
67 //This trigger detects that buffer 2 is full and ready to be written
68 IF Buffer2_R_TRIG.Q THEN
69   GVL_LOGGER.bBusy := TRUE;
70   arDataBuffer := ADR(GVL_LOGGER.arDataBuffer2);
71 
72   IF GVL_LOGGER.bStart AND bFirstLog THEN //bStartLog
73     eState := E_CUSTOMLOGGER_STATES.eSTATE_ACQUIRING_TIME;
74   ELSIF NOT bFirstLog THEN
75     eState := E_CUSTOMLOGGER_STATES.eSTATE_FILEDATA_PROCESSING;
76   END_IF
77   nRowToLog := MAX_ROWS;
78   nActualRow := 1;
79   nRow := 1;
80 END_IF
81 
82 //This trigger performs the segmentation by time
83 IF T_Segmentation_Time.Q  THEN
84   T_Segmentation_Time(IN:=FALSE);
85   nSaveCounter := nSaveCounter +1;
86   sPathName := '';
87   bTimeExpired := TRUE;
88 END_IF
89 
90 //This case statement will handle the sequencing of the writing
91 //================================================================
92 CASE eState OF
93   E_CUSTOMLOGGER_STATES.eSTATE_ERROR: //Error State
94     GVL_LOGGER.bError := TRUE;
95     GVL_LOGGER.bBusy := FALSE;
96 
97   E_CUSTOMLOGGER_STATES.eSTATE_IDLE:    // IDLE (Wait for the start trigger)
98     GVL_LOGGER.bBusy := FALSE;
99 
100  E_CUSTOMLOGGER_STATES.eSTATE_ACQUIRING_TIME: //Acquire the time
101    GVL_LOGGER.bError := FALSE;
102    GVL_LOGGER.eErrorId := E_CUSTOMLOGGER_ERRORS.eERROR_NOERROR;
103    NT_GetTime_0(START := FALSE);
104    NT_GetTime_0(NETID:= GVL_LOGGER.stParams.sNetId, START := TRUE);
105    eState := E_CUSTOMLOGGER_STATES.eSTATE_WAITING_ACQUISITION;
106 
107  E_CUSTOMLOGGER_STATES.eSTATE_WAITING_ACQUISITION: //Check if the time was acquired
108    NT_GetTime_0(START := FALSE);
109    IF NOT NT_GetTime_0.BUSY AND NOT NT_GetTime_0.ERR THEN
110      NT_GetTime_0(START := FALSE);
111      eState := E_CUSTOMLOGGER_STATES.eSTATE_FILENAME_AND_PATH_CREATION;
112    END_IF
113 
114  E_CUSTOMLOGGER_STATES.eSTATE_FILENAME_AND_PATH_CREATION: // Construct the path and file name
115 
116      sPathName := CONCAT(CONCAT(GVL_LOGGER.stParams.sFilePath, GVL_LOGGER.stParams.sFileName)
     ,'_');
117    sPathName := CONCAT(sPathName, WORD_TO_STRING(NT_GetTime_0.TIMESTR.wYear));
118    sPathName := CONCAT(sPathName,'-');
119    sPathName := CONCAT(sPathName, WORD_TO_STRING(NT_GetTime_0.TIMESTR.wMonth));
120    sPathName := CONCAT(sPathName,'-');
121    sPathName := CONCAT(sPathName, WORD_TO_STRING(NT_GetTime_0.TIMESTR.wDay));
122    sPathName := CONCAT(sPathName,'_');
123    sPathName := CONCAT(sPathName, WORD_TO_STRING(NT_GetTime_0.TIMESTR.wHour));
124    sPathName := CONCAT(sPathName,'.');
125    sPathName := CONCAT(sPathName, WORD_TO_STRING(NT_GetTime_0.TIMESTR.wMinute));
126    sPathName := CONCAT(sPathName,'.');
127    sPathName := CONCAT(sPathName, WORD_TO_STRING(NT_GetTime_0.TIMESTR.wSecond));
128    sPathName := CONCAT(CONCAT(sPathName, '_'),INT_TO_STRING(nSaveCounter));
129    sPathName := CONCAT(sPathName,'.csv');
130 
131    eState:= E_CUSTOMLOGGER_STATES.eSTATE_OPENING_FILE;
132 
133  E_CUSTOMLOGGER_STATES.eSTATE_OPENING_FILE: //Create and open the file
134    FB_FileOpen_0(bExecute := FALSE );
135     FB_FileOpen_0(sNetId := GVL_LOGGER.stParams.sNetId, sPathName := sPathName, nMode :=
    FOPEN_MODEWRITE OR FOPEN_MODEBINARY,
136                        ePath := PATH_GENERIC, bExecute := TRUE );
137    eState:= E_CUSTOMLOGGER_STATES.eSTATE_WAITING_FILE_OPENING;
138 
```

```
139 E_CUSTOMLOGGER_STATES.eSTATE_WAITING_FILE_OPENING: //Make sure the FB is free and there isn'
      t error
140   FB_FileOpen_0( bExecute := TRUE, hFile => hFile );
141   IF NOT FB_FileOpen_0.bBusy AND NOT FB_FileOpen_0.bError THEN
142     GVL_LOGGER.bFileOpen := TRUE;
143     GVL_LOGGER.bFileClosed := FALSE;
144     FB_FileOpen_0(bExecute := FALSE);
145     eState := E_CUSTOMLOGGER_STATES.eSTATE_FILEHEADER_CREATION;
146   ELSIF FB_FileOpen_0.bError THEN
147     GVL_LOGGER.eErrorId := E_CUSTOMLOGGER_ERRORS.eERROR_FILEOPEN;
148     eState := E_CUSTOMLOGGER_STATES.eSTATE_ERROR;
149   END_IF
150
151 E_CUSTOMLOGGER_STATES.eSTATE_FILEHEADER_CREATION: // Create the Header of file
152   FB_BufferWriter_0.eCmd := eEnumCmd_First;
153   sCSVField := 'ms';
154   FOR nColumn := 1 TO GVL_LOGGER.stParams.nNumberOfColumns+1 BY 1 DO
155     FB_BufferWriter_0(pBuffer := ADR( sCSVLine ), cbBuffer := SIZEOF( sCSVLine ), putValue :
      = sCSVField,
156               pValue := 0, cbValue := 0,  bCRLF :=  (nColumn = GVL_LOGGER.stParams.
      nNumberOfColumns+1));
157     FB_BufferWriter_0.eCmd := eEnumCmd_Next;
158     sCSVField := GVL_LOGGER.stParams.arColumnHeadings[nColumn];
159   END_FOR
160
161   eState := E_CUSTOMLOGGER_STATES.eSTATE_WRITING_FILE;
162
163
164 E_CUSTOMLOGGER_STATES.eSTATE_FILEDATA_PROCESSING: //Create the buffer of data to write
      inside the file
165
166   WHILE nRow <= 2 AND nActualRow < nRowToLog+1 DO
167
168     FOR nColumn := 1 TO GVL_LOGGER.stParams.nNumberOfColumns+1 BY 1 DO // one iteration to
      add ms and one iteration to add CRLF
169
170       FB_BufferWriter_0(pBuffer := ADR( sCSVLine ), cbBuffer := SIZEOF( sCSVLine ), putValue
       := arDataBuffer^[nActualRow, nColumn], pValue := 0,
171                cbValue := 0, bCRLF := (nColumn = GVL_LOGGER.stParams.nNumberOfColumns+1));
172
173       FB_BufferWriter_0.eCmd := eEnumCmd_Next;
174
175     END_FOR
176
177     FB_BufferWriter_0.eCmd := eEnumCmd_Next;
178     nActualRow := nActualRow + 1;
179     nRow := nRow + 1;
180
181   END_WHILE
182
183   nRow := 1;
184
185   IF nActualRow = nRowToLog+1 THEN
186     eState := E_CUSTOMLOGGER_STATES.eSTATE_WRITING_FILE;
187     nActualRow := 1;
188   ELSE
189     eState := E_CUSTOMLOGGER_STATES.eSTATE_FILEDATA_PROCESSING;
190   END_IF
191
192
193 E_CUSTOMLOGGER_STATES.eSTATE_WRITING_FILE:  // Write the buffer into the file
194
195   FB_FileWrite_0( sNetId := GVL_LOGGER.stParams.sNetId, hFile := hFile, bExecute := TRUE,
      pWriteBuff := ADR(sCSVLine), cbWriteLen := FB_BufferWriter_0.cbSize);
196   IF NOT FB_FileWrite_0.bBusy AND NOT FB_FileWrite_0.bError THEN
197     FB_FileWrite_0(bExecute:= FALSE);
198     nFileDimension_Byte := nFileDimension_Byte + FB_BufferWriter_0.cbSize;
199     nFileDimension_MByte := ULINT_TO_LREAL(nFileDimension_Byte)/EXPT(2,20);
200     IF (bLastLog AND NOT bFirstLog) OR bTimeExpired OR bMaxSizeReached THEN
201       eState := E_CUSTOMLOGGER_STATES.eSTATE_CLOSING_FILE;
202     ELSIF bFirstLog THEN
203       eState := E_CUSTOMLOGGER_STATES.eSTATE_FILEDATA_PROCESSING;
204       bFirstLog := FALSE;
205     ELSE
206       eState := E_CUSTOMLOGGER_STATES.eSTATE_IDLE;
```

```
207         END_IF
208 
209         FB_BufferWriter_0.eCmd := eEnumCmd_First;
210 
211       ELSIF FB_FileWrite_0.bError THEN
212         GVL_LOGGER.bError := TRUE;
213         GVL_LOGGER.eErrorId := E_CUSTOMLOGGER_ERRORS.eERROR_FILEWRITE;
214         eState := E_CUSTOMLOGGER_STATES.eSTATE_CLOSING_FILE;
215       END_IF
216 
217     E_CUSTOMLOGGER_STATES.eSTATE_CLOSING_FILE: //Close File
218       FB_FileClose_0(sNetId := GVL_LOGGER.stParams.sNetId, bExecute := TRUE, hFile:= hFile);
219       IF NOT FB_FileClose_0.bBusy AND NOT FB_FileClose_0.bError THEN
220         GVL_LOGGER.bFileOpen := FALSE;
221         GVL_LOGGER.bFileClosed := TRUE;
222         GVL_LOGGER.nBufferActive := 0;
223           FB_FileClose_0(bExecute := FALSE);
224         IF bTimeExpired OR bMaxSizeReached THEN
225           bTimeExpired := FALSE;
226           bMaxSizeReached := FALSE;
227           eState := E_CUSTOMLOGGER_STATES.eSTATE_FILENAME_AND_PATH_CREATION;
228           GVL_LOGGER.nBufferActive := nLastBufferActive;
229         ELSE
230           eState:= E_CUSTOMLOGGER_STATES.eSTATE_IDLE;
231         END_IF
232       END_IF
233 
234 
235 END_CASE
236 GVL_LOGGER.eLoggerState := eState;
237 nLastBufferActive := GVL_LOGGER.nBufferActive;
```

Implementazione logger manager

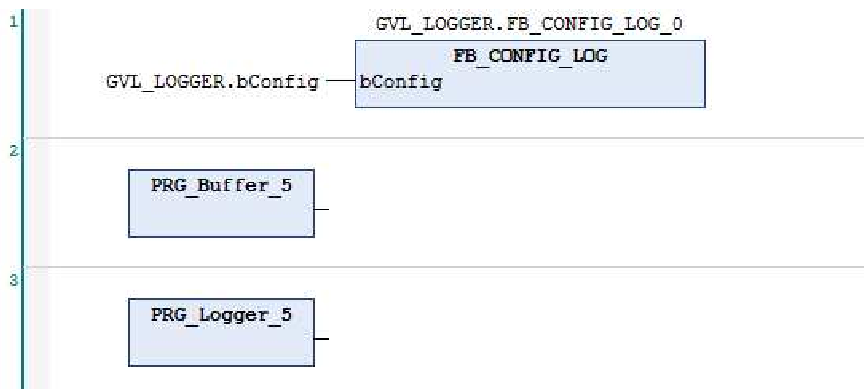


Figura 43: Logger manager quinta versione data-logger

Analisi prestazioni per 300 canali

Per l'ultima versione di data-logger vengono dati i risultati solamente per un campionamento a 300 canali in quanto i risultati sono del tutto simili alla quarta versione sviluppata.

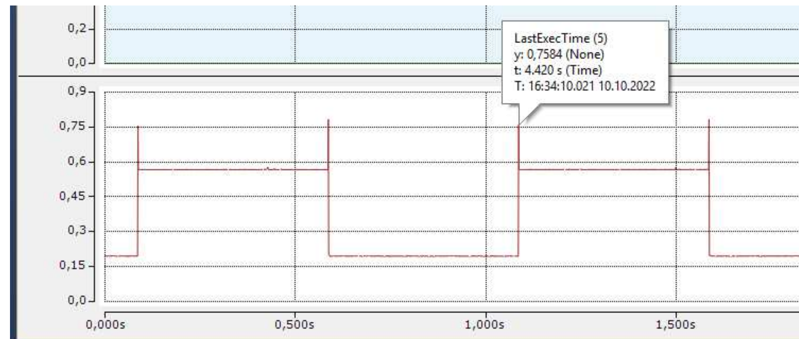


Figura 44: Valore del primo picco della quinta versione a 300 canali

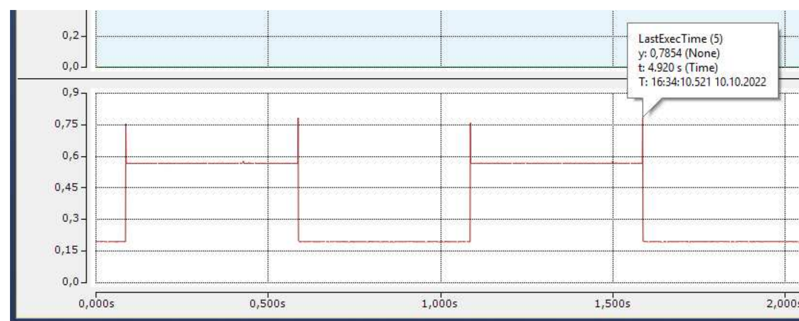


Figura 45: Valore del secondo picco della quinta versione a 300 canali

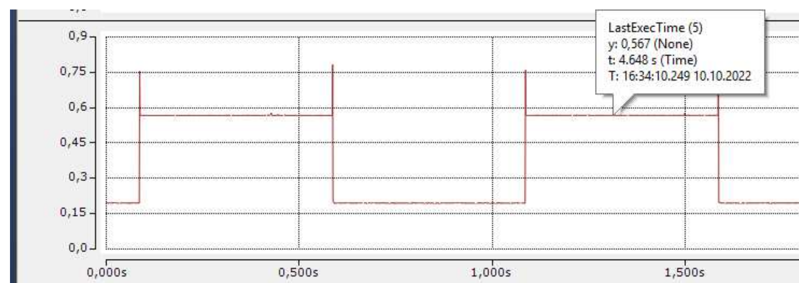


Figura 46: Valore di regime della quinta versione a 300 canali

Come possiamo osservare il tempo ciclo, anche se di poco, è addirittura migliorato ed in più ora si ha anche un programma con logica di buffer e logger separati.

8 Validazione e test

Per la parte di validazione e test oltre le prove effettuate sopra per tutte le versioni di data-logger si sono effettuati anche due stress test sulla quarta versione di dataLogger. Gli stress test consistevano nel lasciare la CX accesa per tutta la notte con il programma del data-logger in run. Questo test è stato effettuato, come detto prima, due volte:

- Una volta configurando la segmentazione per **dimensione** a 100 MB, ovvero il data-logger generava dei file ".csv" che avevano una dimensione massima pari a 100MB, una volta raggiunta la dimensione massima in scrittura il file corrente veniva chiuso e ne veniva aperto un altro con lo stesso nome ma aggiornando il numero incrementale alla fine del nome.
- Una volta configurando la segmentazione per **tempo** ogni 10 minuti, ovvero il data-logger generava dei file ".csv" in cui scriveva per 10 minuti, una volta raggiunto il tempo limite in scrittura chiudeva il file corrente e ne apriva un altro.

I risultati del test sono stati ottimi in quanto in entrambi i casi erano stati generati tutti i file in maniera corretta, ovvero senza alcuna perdita dati visibile all'interno e segmentati nella maniera corretta. In totale sono stati generati 150 file per la segmentazione per dimensione e 124 per la segmentazione per tempo.

9 VISU

Per rispettare anche il **RNF3**, ovvero quello dell'usabilità si è pensato di fare uso della VISU (acronimo di visualization) che non è altro che un componente software che consente di creare interfacce utente grafiche per il monitoraggio e il controllo dei sistemi di automazione. Essa risulta molto utile quando c'è bisogno che gli operatori possano interagire in maniera semplice e visualizzare in modo intuitivo lo stato del sistema e monitorare i dati in tempo reale tramite pulsanti, slider, menu a discesa e altri elementi interattivi. Il sistema di visualizzazione creato potrà poi essere spostato e utilizzato su PC o pannelli HMI, rendendo quindi possibile il monitoraggio e il controllo del sistema da diversi dispositivi e posizioni.

9.1 Realizzazione VISU

Per costruire la VISU come prima cosa va creato un nuovo progetto, come mostrato in figura 47 agendo dal "Solution Explorer" e facendo tasto destro su VISUs –> Add –> Visualization sarà possibile creare un nuovo progetto all'interno del quale si potrà costruire una qualsiasi HMI .

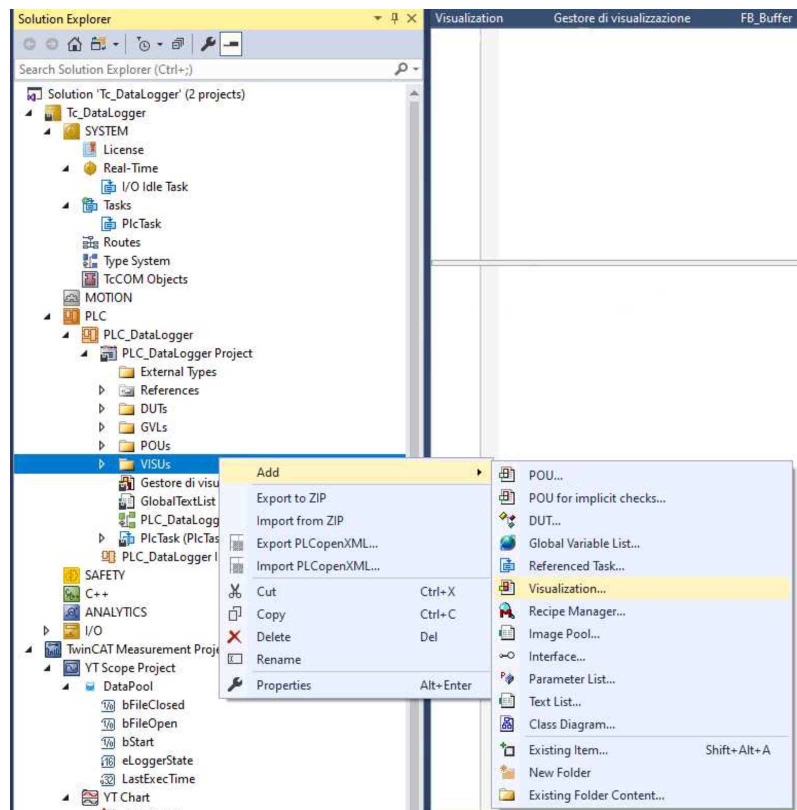


Figura 47: Come creare un nuovo progetto per VISU

Dopo aver creato il progetto per costruire l'interfaccia grafica si sono andati a cercare i componenti che si desideravano inserire nella toolbox per poi trascinarli nello spazio di lavoro. Nell'esempio in figura 48 è possibile vedere nel rettangolo in alto a destra la suddetta toolbox mentre viene cercato come componente un bottone.

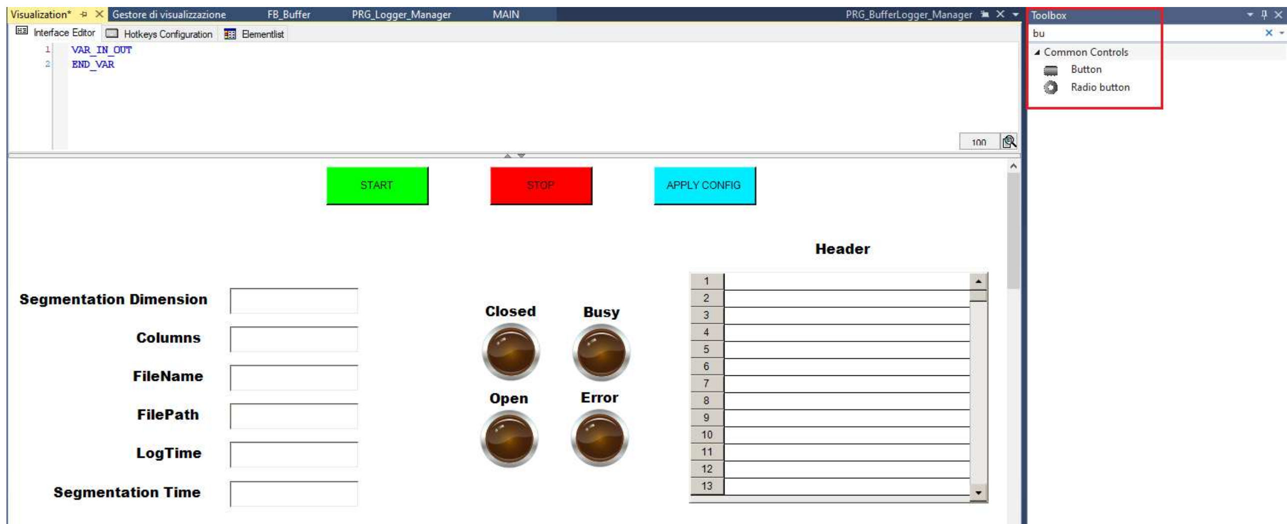


Figura 48: HMI finale

Sempre in figura 48 è possibile vedere la Visu finale con i relativi componenti, dove si possono distinguere 3 elementi di tipo *button*:

- **1 pulsante *START***: premendolo il logger inizia a funzionare e viene creato il file *.csv in cui saranno scritti e salvati i valori campionati
- **1 pulsante *STOP***: premendolo il logger smette di funzionare e il file *.csv (dopo che tutti i valori sono stati scritti) viene chiuso.
- **1 pulsante *APLLY CONFIG***: premendolo prima di premere il pulsante di START vengono applicate tutte le configurazioni scelte dall'utente.

L'utente può applicare le sue configurazioni mediante quelle che sono le *text field* e la *table* la quale permette di inserire all'interno di ogni riga (le righe sono in numero pari al numero di canali da loggare) il nome personalizzato che si vuole associare ad ogni canale da loggare. Le *text field* invece sono:

- **1 campo testo *Segmentation Dimension***: il quale consente di impostare la dimensione massima del file *.csv raggiunta la quale il file viene chiuso, numerato con un numero incrementale. Dopodiché ne viene aperto un altro nel quale il logger continuerà a scrivere da dove si era interrotto.
- **1 campo testo *Columns***: con il quale è possibile indicare il numero di canali da loggare contemporaneamente, il termine "columns" fa riferimento al numero di colonne che poi saranno presenti nel file *.csv.
- **1 campo testo *File Name***: con il quale è possibile scegliere il nome del file singolo (o di tutte le eventuali segmentazioni) alla quale (o alle quali) verrà aggiunta la data corrente e un numero incrementale che parte da 0.
- **1 campo testo *File Path***: con il quale è possibile scegliere la locazione in cui i file saranno salvati.

- **1 campo testo *Log Time***: con il quale è possibile definire ogni quanti millisecondi avviene il campionamento.
  Per questa tesi si è usato sempre un tempo di campionamento pari a 1ms.
- **1 campo testo *Segmentation Time***: il quale consente di impostare un tempo massimo di campionamento del file *.csv raggiunto il quale il file viene chiuso e numerato con un numero incrementale. Dopodiché ne viene aperto un altro nel quale il logger continuerà a scrivere da dove si era interrotto.

Infine abbiamo 4 output sotto forma di elementi di tipo *lamp* che non sono altro che delle lampade che segnalano lo stato in cui il sistema si trova, esse sono rispettivamente:

- **1 lampada *Closed***: se accesa indica che il logger sta chiudendo il file *.csv che fino a quel momento era usato per salvare dati campionati. Questo può accadere per tre motivi:
  1. L'utente ha premuto il pulsante di stop;
  2. Il tempo massimo di campionamento è stato raggiunto quindi il file corrente deve essere chiuso per lasciare spazio al nuovo file;
  3. La dimensione massima del file è stata raggiunta perciò il file corrente deve essere chiuso per lasciare spazio al nuovo file.
- **1 lampada *Busy***: se lampeggiante indica che il logger sta scrivendo su file. Il lampeggio è dovuto al fatto che la lampada si accende quando il logger è in fase di scrittura e si spegne quando finisce di scrivere, ed essendo più veloce il logger a scrivere in blocco il buffer (impiega circa 700 ms) che l'altro buffer a riempirsi(ci impiega 1 secondo preciso) c'è un intervallo di tempo in cui la lampada rimane completamente spenta.
- **1 lampada *Opened***: se accesa indica che il logger sta aprendo un nuovo file per essere scritto. Questo può accadere per le stesse 3 motivazioni espresse sopra ma per il caso di apertura del file, ovvero:
  1. L'utente ha premuto il pulsante di start
  2. Il tempo massimo di campionamento è stato raggiunto quindi il vecchio file è stato chiuso, occorre ora che venga aperto un nuovo file;
  3. La dimensione massima del vecchio file è stata raggiunta e il vecchio file è stato quindi chiuso, occorre ora che un nuovo file venga aperto
- **1 lampada *Error***: Il logger è andato in uno stato di errore controllato, i quali sono:
  1. ***eERROR_FILEOPEN***: Il logger non è riuscito ad aprire un nuovo file
  2. ***eERROR_FILEWRITE***: Il logger è andato in errore mentre scriveva sul file
  3. ***eERROR_FILECLOSE***: Il logger è andato in errore mentre cercava di chiudere il file
  4. ***eERROR_INVALID_SEGMENTATION_PARAM***: L'utente ha inserito dei parametri di segmentazione non validi, ad esempio ha richiesto che il file venisse segmentato sia per tempo che per dimensione, oppure ha inserito un tempo di segmentazione negativo.

9.2 Collegamento GVL con VISU

Collegare gli elementi della VISU alla GVL significa associare gli elementi grafici (come pulsanti, campi di testo, slider, ecc.) alle variabili corrispondenti nella GVL. In questo modo, quando un utente interagisce con un elemento della VISU, ad esempio premendo un pulsante, la VISU invia un comando o una modifica alla variabile associata nella GVL, questo consente quindi all'utente di interagire in maniera semplice e diretta con il sistema. Per collegare un elemento alla relativa variabile globale occorre cliccare con il tasto sinistro sull'elemento e andare nelle "Properties" poi da lì ogni tipologia di elemento a le sue procedure di configurazione. Di seguito vengono riportati tre modalità di collegamento.

Collegamento lampada

In questo caso il collegamento permette alla lampada di avere una variabile di input di tipo BOOL da cui potrà andare a leggere un valore e comportarsi di conseguenza, ovvero essa si illuminerà quando quella specifica variabile assumerà valore TRUE mentre si spegnerà quando assumerà valore FALSE. In figura 49 viene mostrato come è possibile eseguire l'associazione. Dopo aver cliccato sull'elemento occorre:

1. Cliccare sui 3 pallini oppure scrivere direttamente all'interno del campo il nome della variabile;

2. Nel caso si fosse cliccato il pulsante con i 3 pallini si aprirà una finestra in cui basterà andare a fare doppio click sulla variabile che si desidera associare presente all'interno della cartella GVLs.

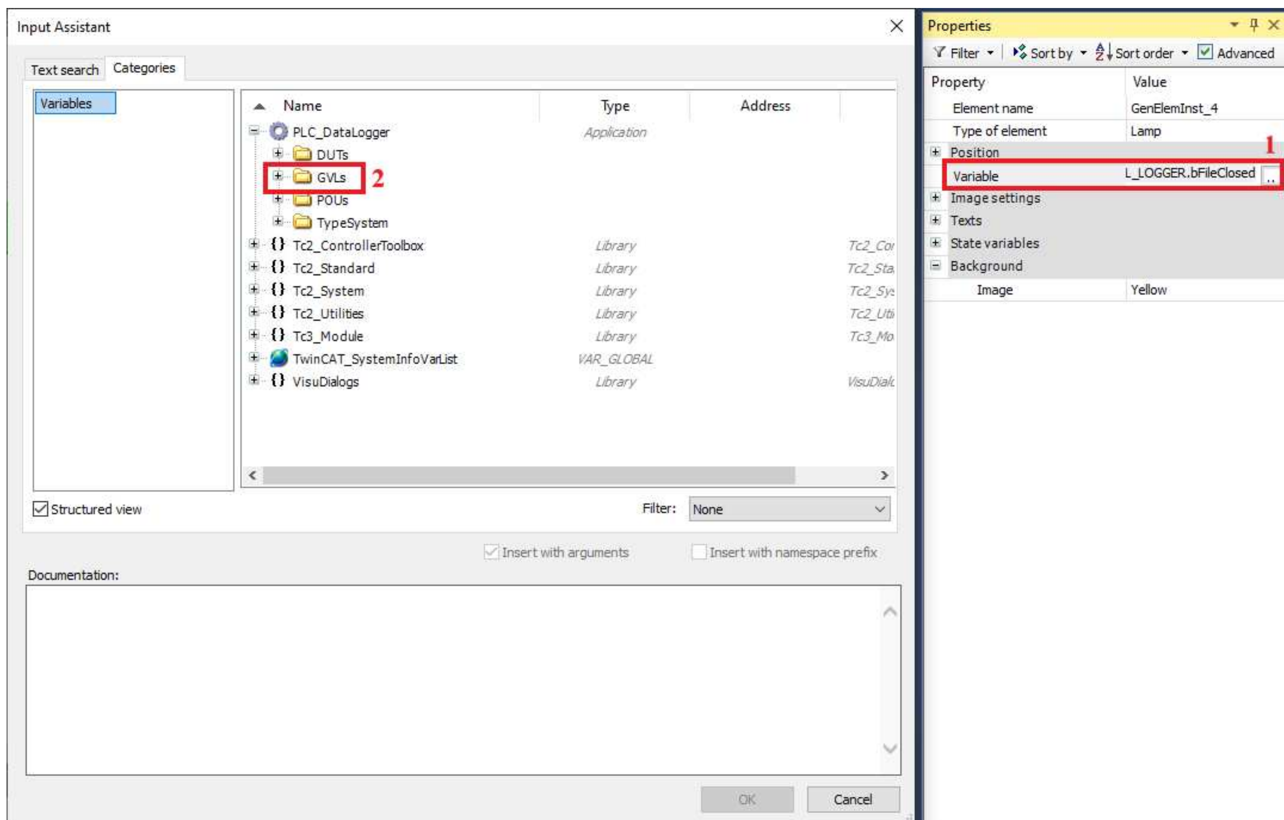


Figura 49: Procedimento per collegamento lampada con relativa GVL

Collegamento campo di testo

In questo caso il collegamento permette al campo di testo di avere una variabile all'interno della quale andare a memorizzare il valore inserito dall'utente. Questa modalità di associazione è valida anche per l'elemento di tipo *table*.
In figura 50 viene mostrato come è possibile eseguire l'associazione. Dopo aver cliccato sull'elemento occorre:

1. Cliccare sull'azione a cui si vuole far corrispondere una reazione ovvero in questo caso: *OnMouseClick*;
2. Al click del mouse vogliamo che quello che viene scritto all'interno del campo di testo venga copiato all'interno di una determinata variabile;
3. Scrivere il nome della variabile in cui vogliamo copiare il valore.

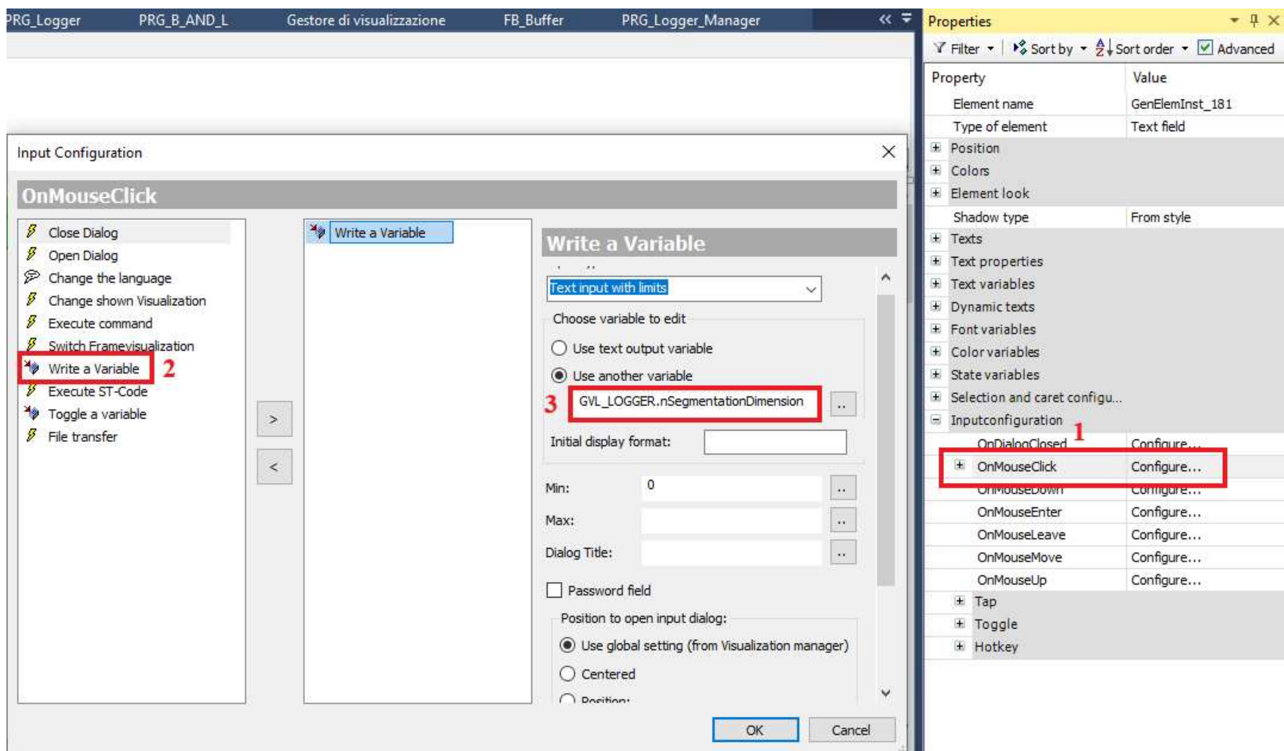


Figura 50: Procedimento per collegamento del campo di testo con relativa GVL

Collegamento bottone

In questo caso il collegamento permette al bottone di avere una variabile di tipo BOOL alla quale associa un valore uguale a TRUE quando esso viene premuto.

1. Cliccare sull'azione a cui si vuole far corrispondere una reazione ovvero in questo caso: *OnMouseClick*;
2. Al click del mouse vogliamo che venga eseguito un piccolo pezzettino di codice che è quello riportato nello step 3;
3. In questo step viene scritto il codice da eseguire al click del bottone ovvero va associato alla varibile di tipo BOOL il valore TRUE.

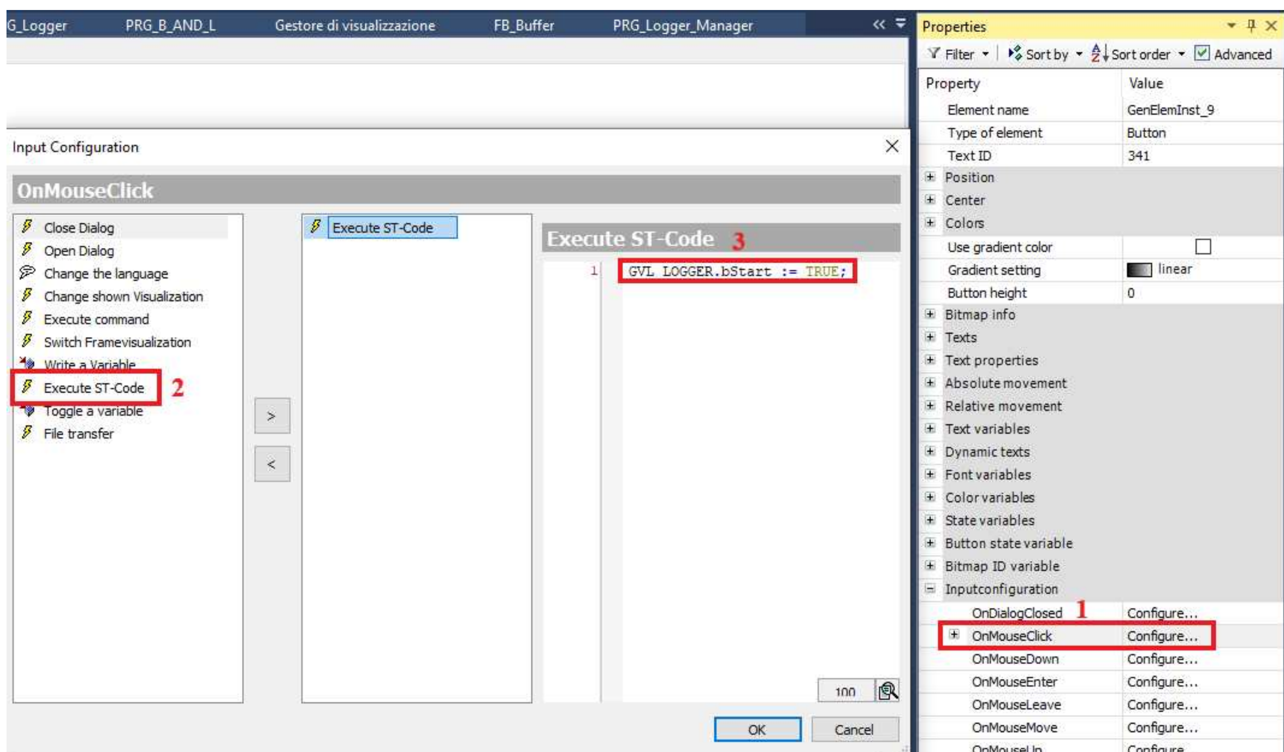


Figura 51: Procedimento per collegamento bottone con relativa GVL

10 Conclusioni ed eventuali sviluppi futuri

Di seguito si riporta una tabella riassuntiva dei risultati ottenuti in termini di prestazioni. In rosso sono stati evidenziati i tempi ciclo più lunghi di 1 ms mentre in giallo quelli al limite ed in verde quelli al di sotto di tale tempo. Inoltre per la quinta versione di data logger sono stai messi anche i tempi misurati durante l'analisi delle prestazioni per 100 e 200 canali che erano stati omessi nel sottocapitolo 7.5.

		Primo picco	Secondo picco	Regime
Prima verisone	100 canali	6.31 ms	-	4.44 ms
	200 canali	-	-	-
	300 canali	-	-	-
Seconda verisone	100 canali	14.57 ms	-	0.74 ms
	200 canali	-	-	-
	300 canali	-	-	-
Terza versione	100 canali	0.93 ms	-	0.74 ms
	200 canali	-	-	-
	300 canali	-	-	-
Quarta versione	100 canali	0.39 ms	0.23 ms	0.21 ms
	200 canali	0.59 ms	0.55 ms	0.40 ms
	300 canali	0.77 ms	0.81 ms	0.58 ms
Quinta versione	100 canali	0.39 ms	-	0.20 ms
	200 canali	0.57 ms	0.54 ms	0.38 ms
	300 canali	0.76 ms	0.79 ms	0.57 ms

Figura 52: Tabella riassuntiva dei risultati ottenuti

In conclusione, l'analisi dei risultati ottenuti conferma con soddisfazione che tutti i requisiti, sia funzionali che non funzionali, stabiliti all'inizio del progetto sono stati pienamente soddisfatti come mostrato nei capitoli precedenti. Il risultato che dal punto di vista progettuale è stato il più soddisfacente è il superamento del numero di canali minimo da registrare in simultanea ogni millisecondo, passando da un aspettativa iniziale minima di 30 canali/ms a 300 canali/ms.

L'ottenimento di tutti i requisiti, sia funzionali che non funzionali, rappresenta un risultato significativo e conferma al gruppo Loccioni la possibilità di poter tentare questa nuova strada per il data logging nei loro banchi test.

Guardando al futuro, i risultati positivi ottenuti possono fornire spunti per ulteriori ricerche e sviluppi come:

1. Rendere il Data Logger configurabile da C# ed eventualmente implementare delle operazioni di post process, esempio:
    - Compressione dei file dopo un determinato periodo. Ad esempio dopo 5 anni si potrebbe non aver più bisogno di un file in cui si hanno i valori campionati a 1 ms di un sensore ma potrebbe essere sufficiente avere un andamento generale del segnale con un campionamento di 100ms. Si potrebbe quindi creare una routine in cui si vanno a ripulire i file da tutti i campionamenti superflui, anche in ottica di alleggerimento della memoria)
    - Media di un determinato campionamento
    - Unione di più file segmentati

2. Aggiungere le calibrazioni:
    - **LIN**: funzione che si occupa di trasformare l'insieme dei punti RAW (in generale 2^n dove n rappresenta la risoluzione del modulo di acquisizione) in grandezze elettriche (ad esempio Ampere o Volt)
    - **ING**: funzione che si occupa di trasformare le grandezze elettriche in grandezze "ingegneristiche", ad esempio data una certa corrente o voltaggio la trasforma in °C, bar, N e via dicendo.
3. Gestione trigger START AND STOP: ovvero non è l'utente che decide quando il Data Logger deve iniziare a loggare i dati ma è l'occorrenza di un certo evento, ad esempio, inizio a loggare quando la corrente supera una certa soglia oppure inizio a loggare quando l'albero motore supera i 3000 giri/min .
4. Per velocizzare la scrittura su file si potrebbe pensare di implementare una scrittura in binario anziché una scrittura diretta su file *.csv .
5. Sample rate diverso per ogni canale: Dare la possibilità all'utente di poter selezionare un tempo di campionamento diverso in base al canale. Questo perché avrebbe poco senso campionare un sensore di temperatura con la stessa frequenza di un sensore di corrente, infatti per ovvie ragioni fisiche la corrente varia molto più velocemente della temperatura e a parità di frequenza di campionamento (ad esempio 1 ms) verrebbero generati molti dati inutili.
6. Un sistema che permetta la ricostruzione dei file segmentati.

Sitografia

[3] Beckhoff. *DUTs*. URL: https://infosys.beckhoff.com/english.php?content=../content/1033/tc3_plc_intro/2530071691.html&id=.

[4] Beckhoff. *Function block csvMemBufferWriter*. URL: https://infosys.beckhoff.com/english.php?content=../content/1033/tcplclib_tc2_utilities/34979467.html&id=.

[5] Beckhoff. *Function block file close*. URL: https://infosys.beckhoff.com/english.php?content=../content/1033/tcplclib_tc2_system/30972939.html&id=.

[6] Beckhoff. *Function block file open*. URL: https://infosys.beckhoff.com/english.php?content=../content/1033/tcplclib_tc2_system/30977547.html&id=.

[7] Beckhoff. *Function block file writer*. URL: https://infosys.beckhoff.com/english.php?content=../content/1033/tcplclib_tc2_system/30986763.html&id=.

[8] Beckhoff. *GVLs*. URL: https://infosys.beckhoff.com/english.php?content=../content/1033/tc3_plc_intro/2530218891.html&id=.

[9] Beckhoff. *Logger Beckhoff standard*. URL: https://infosys.beckhoff.com/english.php?content=../content/1033/tf4100_tc3_controller_toolbox/245475339.html&id=.

[10] Beckhoff. *POUs*. URL: https://infosys.beckhoff.com/english.php?content=../content/1033/tcplccontrol/925240075.html&id=3072943758336742557.

[11] Beckhoff. *Visualization*. URL: https://infosys.beckhoff.com/english.php?content=../content/1033/tc3_plc_intro/3523377803435966657l.html&id=.

[12] GeeksforGeeks. *Buffer doppio*. URL: https://www.geeksforgeeks.org/double-buffering/.

[13] spiegato.com. *Buffer Circolare*. URL: https://spiegato.com/che-cose-un-buffer-circolare.

[14] Scott Whitlock. *TwinCAT Measurement project*. URL: https://www.contactandcoil.com/twincat-3-tutorial/the-scope-view/.

[15] Wikipedia. *Standard IEC_61131-3*. URL: https://en.wikipedia.org/wiki/IEC_61131-3.

Articoli

[1] Beckhoff. «1 TwinCAT 3 Overview IT». In: *Corso IT-TR30000-3003 Base* ().
[2] Beckhoff. «2 TwinCAT_RealTime_IT». In: *Corso IT-TR30000-3003 Base* ().

Ringraziamenti

Alla fine di questo elaborato, mi sembra doveroso dedicare uno spazio per ringraziare tutte le persone che, con il loro supporto e la loro guida, mi hanno aiutato in questo lungo periodo universitario con i suoi alti e bassi.

In primis vorrei ringraziare il mio relatore, il Prof. Simone Fiori, grazie per essere stato sempre presente e disponibile per qualsiasi tipo di chiarimento, e per avermi aiutato, con molta pazienza nella stesura di questa tesi.

La mia gratitudine va anche al Dott. Magini, mio tutor presso il gruppo Loccioni. Grazie per avermi guidato con estrema serietà e disponibilità durante il mio tirocinio formativo nonostante i tuoi numerosi impegni all'interno dell' impresa. Un grazie anche per aver arricchito le mie conoscenze tecniche.

Ringrazio inoltre il Dott. Luca Mazzuferi per avermi affidato questo progetto offrendomi così una preziosa prospettiva pratica, aiutandomi a collegare le teorie apprese in università alla realtà lavorativa.

Desidero inoltre estendere un sentito ringraziamento al gruppo Loccioni, l'azienda che mi sta accompagnando oramai da 7 anni. Il vostro sostegno e la vostra collaborazione sono stati fondamentali per la mia crescita personale e professionale. Grazie per avermi accolto nel vostro team e per avermi offerto un ambiente di lavoro stimolante e formativo. Durante questi anni, ho avuto l'opportunità di apprendere da professionisti competenti e di mettere in pratica le mie conoscenze teoriche. Il vostro supporto è stato fondamentale per sviluppare le mie competenze pratiche e per comprendere meglio il funzionamento di un'azienda. Sono grato per le esperienze che ho vissuto con voi e per tutto ciò che ho imparato.

Desidero ringraziare di cuore anche i miei cari amici. Anche se potreste non esservene resi conto, la vostra presenza è stata fondamentale per ricaricare le batterie nei momenti in cui l'università sembrava più dura. Grazie per tutte le risate e i momenti passati insieme.

Un grazie anche ai miei compagni di viaggio in questa triennale. Alessandro, Gregorio e Nicolò grazie per tutti i momenti condivisi in queste sessioni, i film, le cene, gli aperitivi e tanto altro che hanno fatto sembrare questi anni meno lunghi e pesanti.

Grazie anche ai miei parenti per essere qui oggi e per avermi dedicato del tempo per festeggiare insieme questo risultato.

Un grazie anche ai nonni per avermi cresciuto e per tutto il tempo trascorso insieme. Grazie a quelli paterni per avermi trasmesso la bellezza delle cose semplici e dei lavori manuali, grazie anche a nonno Dino per essere stato sempre al mio fianco come un amico. So che sareste stati fieri di me oggi. A te nonna invece, visto che ho ancora la fortuna di averti, dico grazie per avermi accudito sin da bambino e per tutti i pranzetti che mi hai preparato al rientro da scuola.

Ora mi prendo queste ultime righe per ringraziare la mia famiglia. Spero che risultato questo possa ripagare anche in minima parte tutto quello che avete fatto per me in questi 23 anni per rendermi la persona che sono oggi. Grazie per avermi donato come regalo più grande quello della serenità e del coraggio per affrontare anche i percorsi più difficili e di avermi dato la possibilità di fare molte esperienze privandovi, per primi, di qualcosa.
Un grazie anche ad Edoardo per tutti i momenti leggeri passati insieme nel periodo di lockdown e per tutto l'affetto che mi ha dimostrato in questi anni.

Infine vorrei ringraziare me stesso con la speranza e l'augurio che questo sia solo un nuovo punto di partenza.