

**TOR VERGATA**  
UNIVERSITÀ DEGLI STUDI DI ROMA

## MAK1: Test bench hardware for (mini)Codas

28.7.2021 - report V\_1

**TESTING (SLOW) CONTROL ARCHITECTURE (CODAS) for Protosphera (DTT) on a small experimental setup (part of MAK1).**

[DTT WBS, WP: System Integration, WBS: TBD] *Subtask 5: CODAS prototyping for ProtoSphere*

**Initial hardware setup: mimicking the central solenoid control loop**

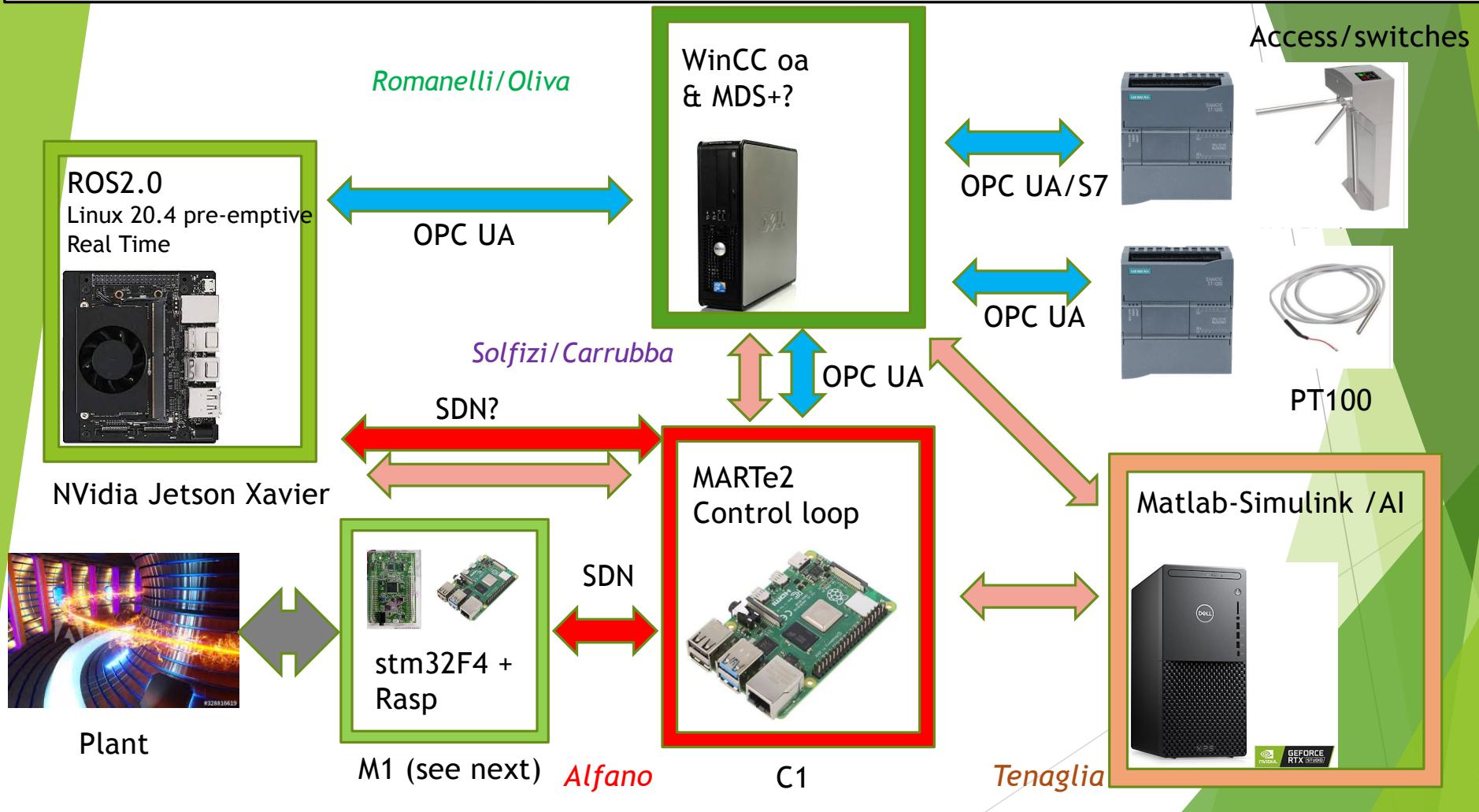
- Iron core transformer with primary loop powered by a current amplifier shield
- current amplifier shield with MOSFET (IRFB4310) driven by IR2104 + micro
- 24V 1000A (hermetic) lead battery
- magnetic sensors (TBD)
- Temperature sensors (2 x PT100)
- 2 x PLC Siemens s7-1200 sampling temperature and magnetic sensors (and other digital switches)
- PC desktop with SCADA Software (WINCCOA)
- PC desktop with MATLAB/LINUX to deploy control code into a MARTe 2.0 architecture (TCV-like...)

**CODAS SOFTWARE:**

Create on WINCCOA a software interface to handle and visualize plant parameters resembling the main feature of Tokamak CODAS:

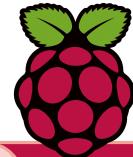
- Prototype the Vacuum plant;
- Experimental Sequence UI;
- Parameter settings and main State Machine;
- Diagnostics (Langmuir probes, cameras, Hall sensors, diamonds);
- Data analysis and visualization

## MAK 1 proposed architecture



## Sistema di Controllo ad alto livello

Sistema di controllo ad alto livello del sistema



### MARTE2 Sistema di controllo centrale

C1

MARTE2

Riceve richieste da ESDN e le attua usando EMP.  
Invia e riceve i dati dalla STM32F4

SDN

## M1: Coil Current Controller

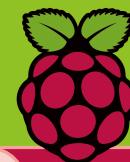
STM32F4  
DAC e attuatore



Current Drive

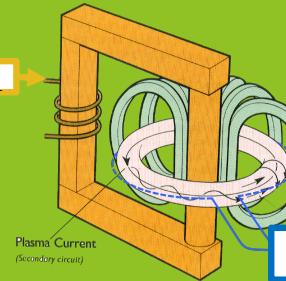
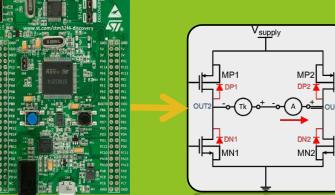
## Trasformatore (CS coil)

Campionamento a  
2Khz della STM32F4

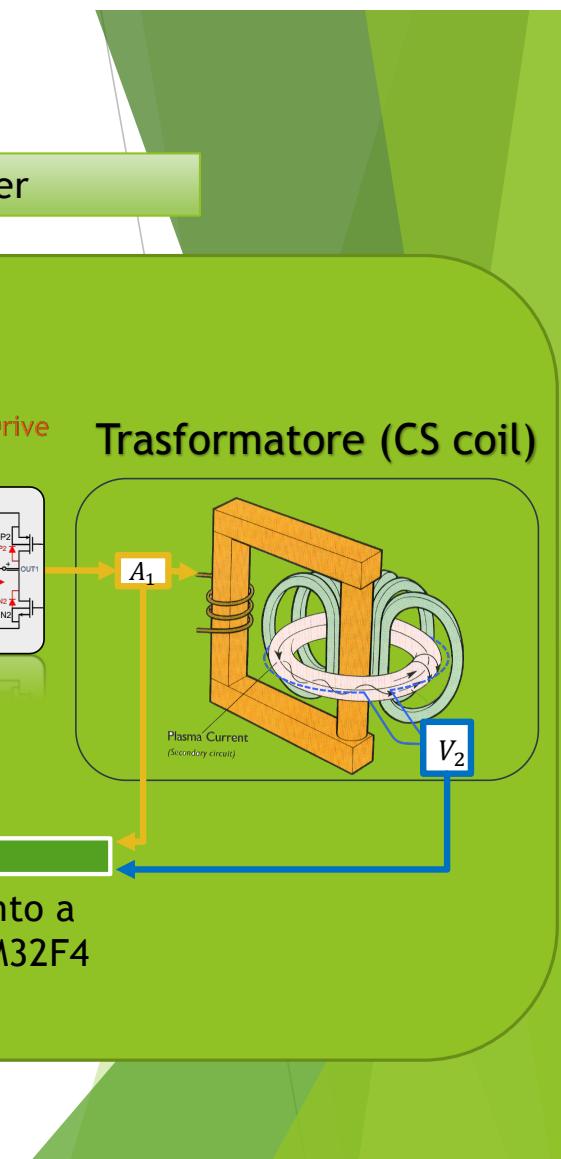


Sistema di Comunicazione a  
pacchetti basato su USB  
EMP

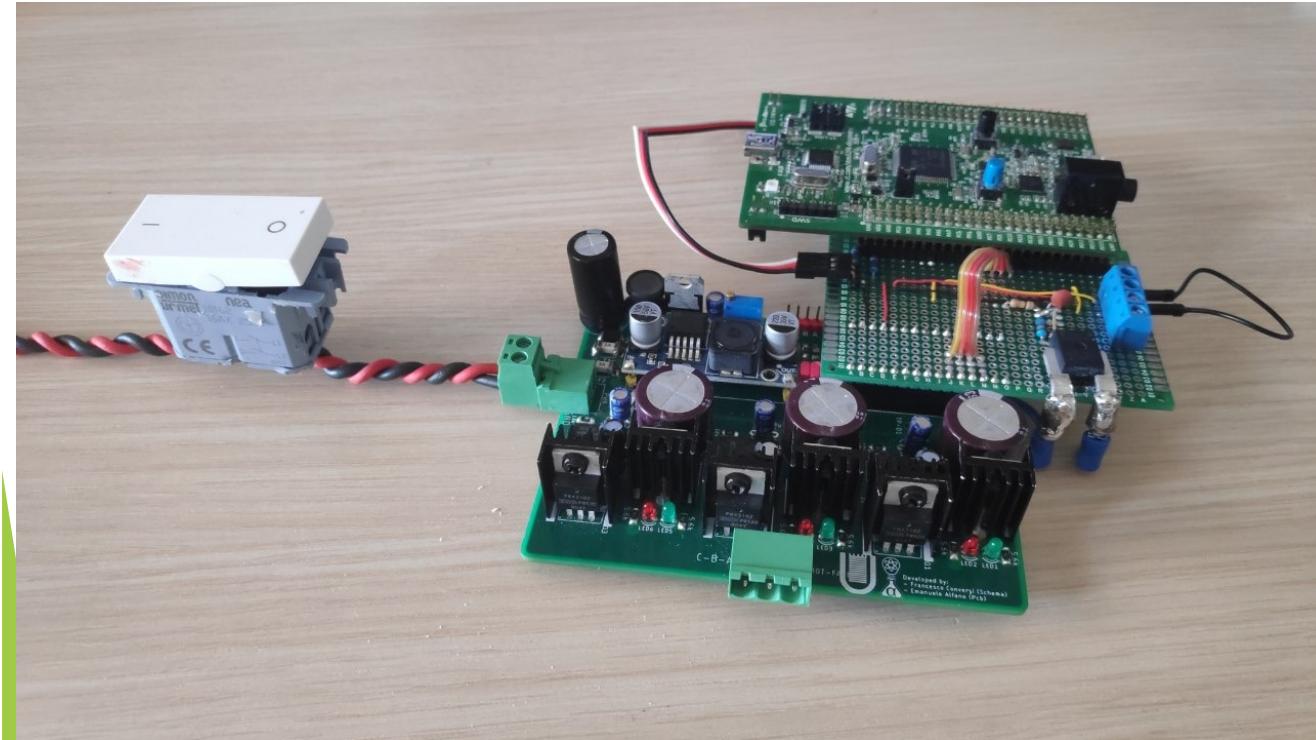
EMP



V<sub>2</sub>



## Prototipo attuale



### Sensori:

Sensibilità Corrente:

50mA

Massima Corrente Misurabile:

$\pm 100A$

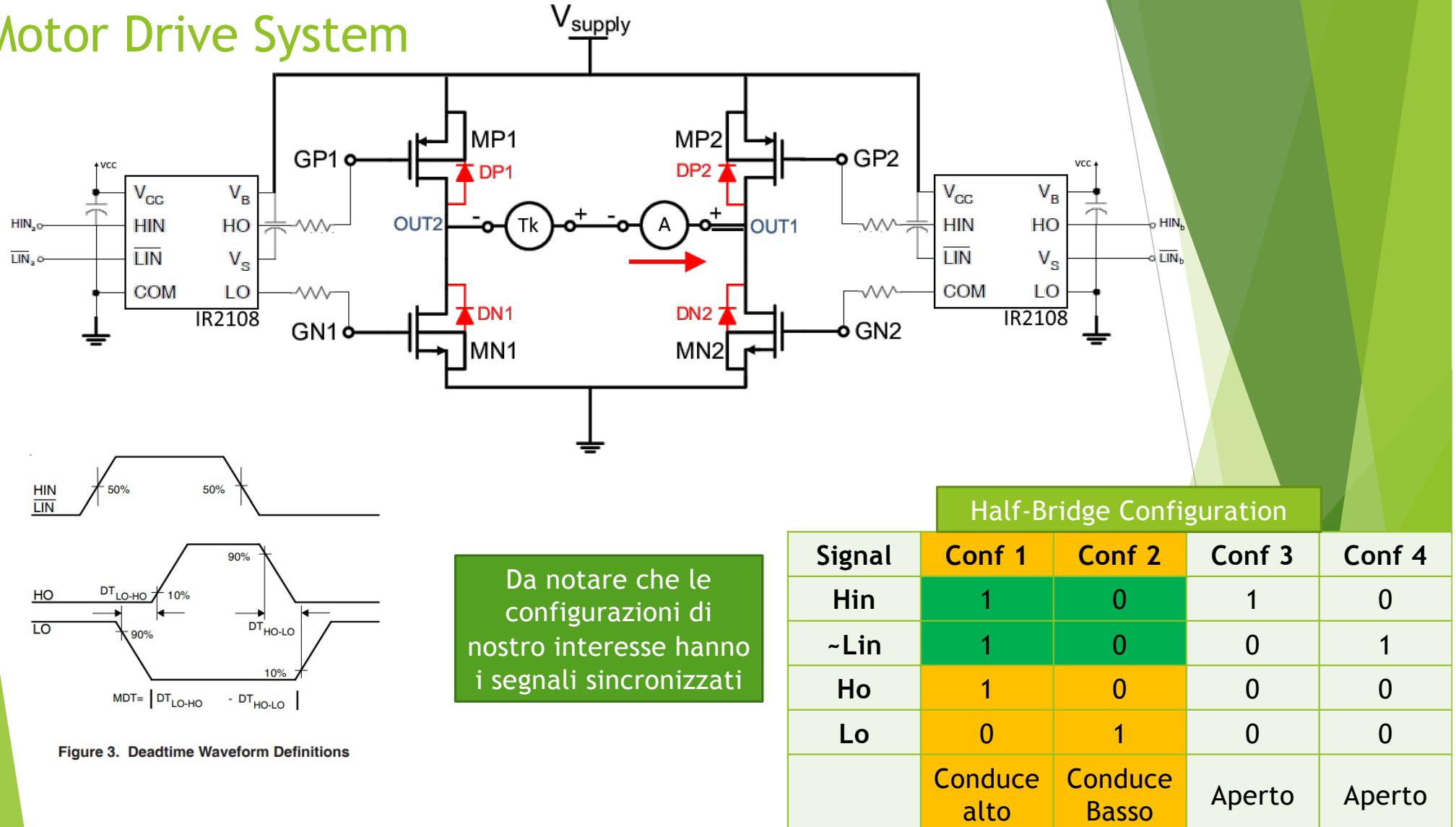
Sensibilità Tensione Secondario:

0,7mV

Massima Tensione Secondario :

1,5V

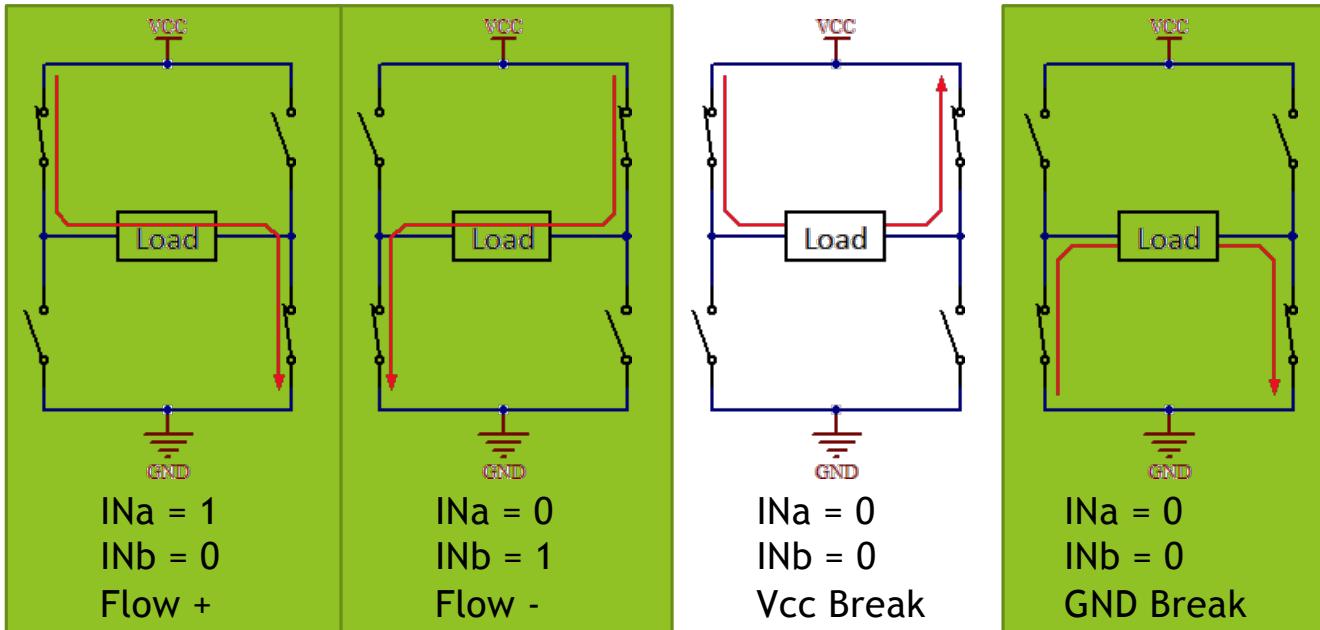
# Motor Drive System



# H-Bridge Possible Configuration

Essendo i segnali di nostro interesse sincronizzati, ci riferiremo a essi:

- lato 1 come INa
- lato 2 come INb



Queste 3 Configurazioni sono di nostro interesse per controllare la corrente, senza mai aprire il circuito, **evitando** così di aggiungere delle dinamiche non lineari al sistema

# Funzione logica di Controllo

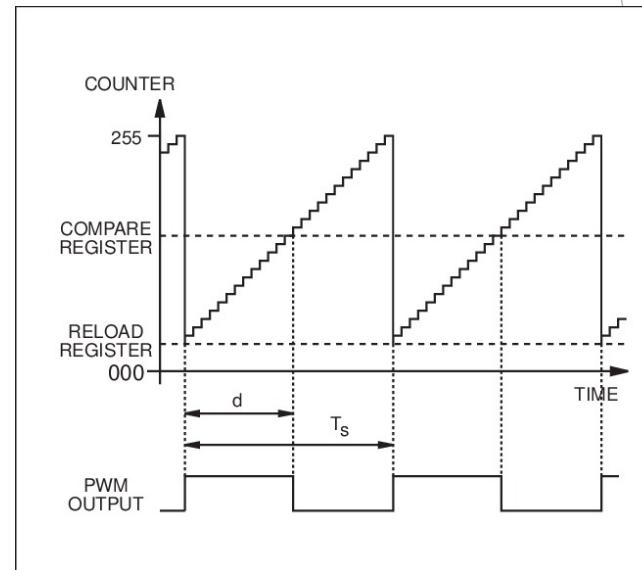
Controllando il ponte Ponte-H usando un segnale di PWM e una direzione di marcia, è necessario trovare una funzione booleana  $F(PWM, Dir) \rightarrow (in_a, in_b)$ :

| PWM | Dir | INa | INb |
|-----|-----|-----|-----|
| 1   | 0   | 1   | 0   |
| 1   | 1   | 0   | 1   |
| 0   | 0   | 0   | 0   |
| 0   | 0   | 0   | 0   |



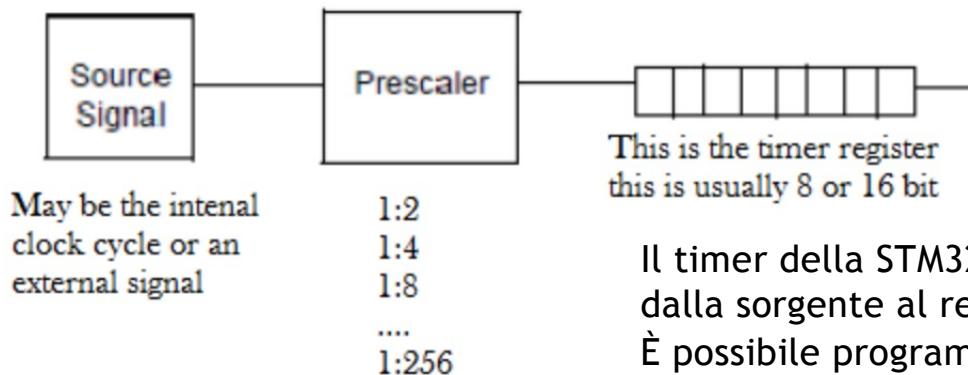
$$INa(Pwm, dir) = Pwm \cdot \overline{dir}$$
$$INb(Pwm, dir) = Pwm \cdot dir$$

Nell'attuare gli output, il valore logico del PWM è osservabile dallo schema, ed è stato impostato opportunamente scegliendo i valori dei registri.

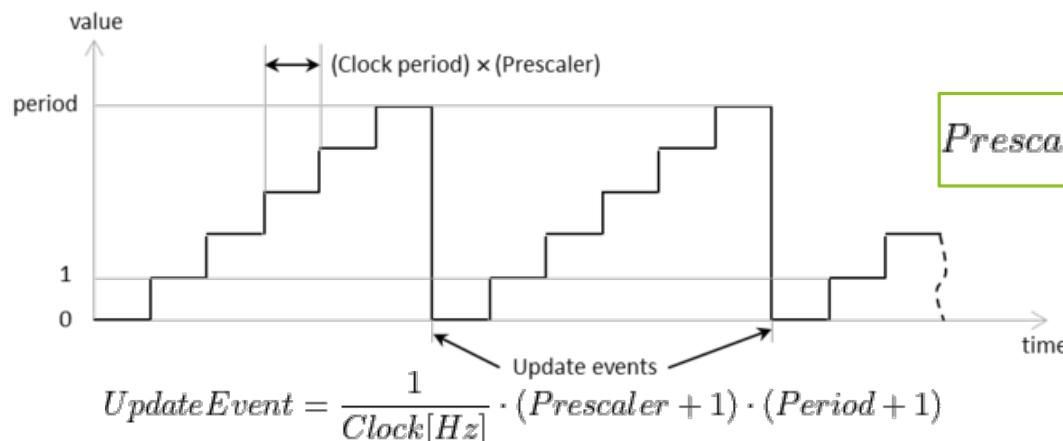


# Il timer di una STM32F4

Timer Block Diagram



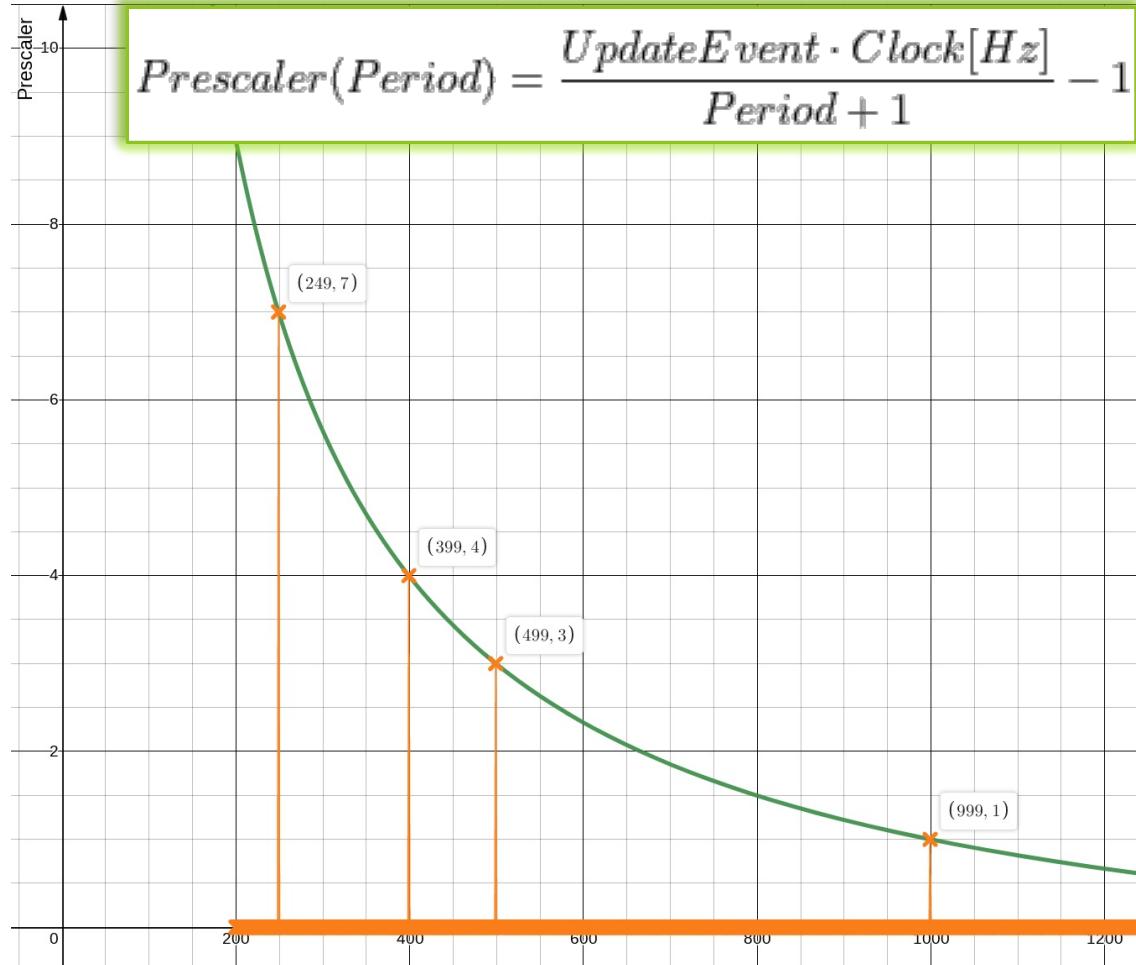
Il timer della STM32F4 ha una struttura a cascata dalla sorgente al registro contatore.  
È possibile programmare sia il Prescaler che il periodo regolando opportunamente i registri:



$$Prescaler(Period) = \frac{UpdateEvent \cdot Clock[Hz]}{Period + 1} - 1$$

# Generare il PWM alla corretta Frequenza

Testando la funzione solo per numeri interi, e cercandone le soluzioni intere esce fuori:



$$c_{lk} = \frac{(168 \cdot 10^6)}{2}$$

$$c_{lk} = 84\ 000\ 000$$

$$t_{des} = \frac{1}{42 \cdot 10^3}$$

$$t_{des} = 0.0000238095238095$$

$$S_{ampleTime} = \frac{1}{2 \cdot 10^3}$$

$$S_{ampleTime} = 0.0005$$

## Current Drive Setting

A corollario di quanto detto, è stato possibile impostare il timer3 della STM32F4 affinché venga generato un PWM da 42Khz, con un pulse-width da [0:999], ovvero una finezza di controllo dello 0,1%.

Ciò impostando i registri a:

Prescaler = 1

LimitCounter = 999



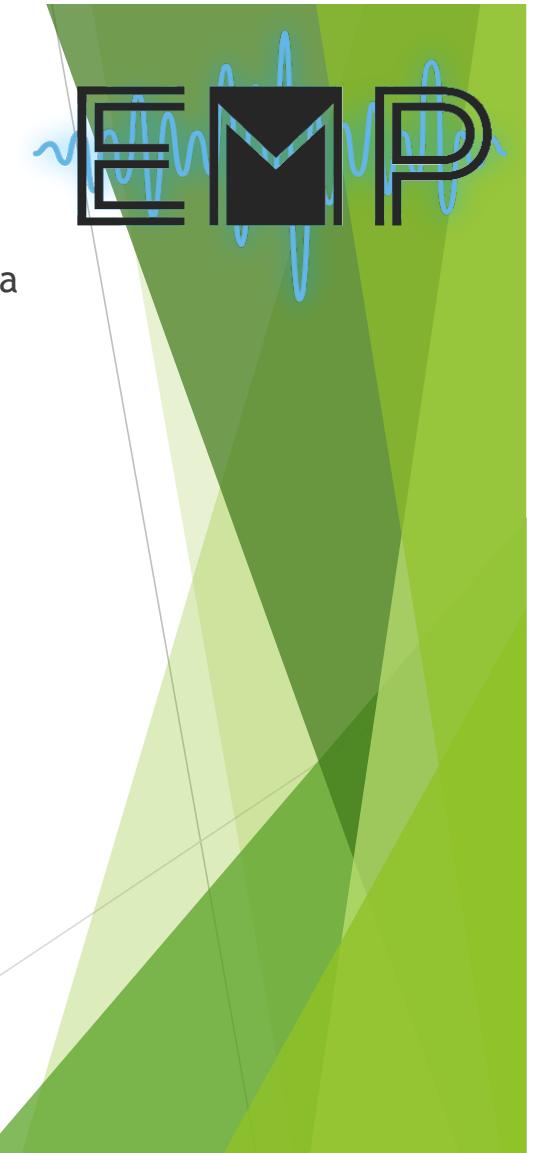
# EMP: Embedded Message Pack

<https://github.com/Alfystar/EMP>

# Scopo, Key feature, limiti di EMP

1. EMP nasce con l'obiettivo di standardizzare un protocollo e creare una libreria C++ basata su classi Template, che permettesse di automatizzare e standardizzare tutto il lavoro di programmazione necessario all'invio e alla ricezione di dei pacchetti dal formato Pré-Concordati tra 2 Device connessi Peer2Peer (Nessuna pretesa di networking).
2. In aggiunta a questa richiesta, EMP è stato pensato per permettere il trasporto, attraverso lo stesso mezzo, di pacchetti di tipologia e dimensione diversa all'interno della stessa libreria, evitando al contempo di inviare per ogni pacchetto più byte dello strettamente necessario.
3. In fine, per renderla adatta a un uso 'Streaming', essa non necessita di nessuna fase di sincronizzazione iniziale, e in caso di fallimento nella trasmissione, è in grado di scartare il pacchetto in maniera trasparente all'utilizzatore.

Lavori di controllo di flusso, e riempimento dei pacchetti sono TUTTI a carico degli utilizzatori della libreria.



# Protocollo di comunicazione

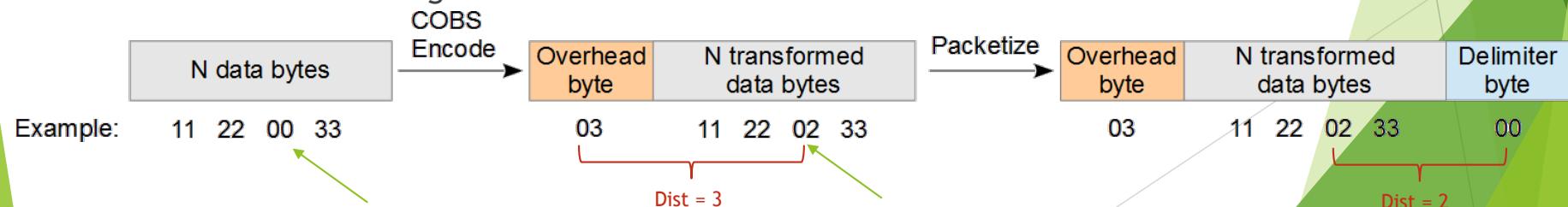
Iniziamo partendo dal protocollo di comunicazione che permette l'invio di pacchetti diversi, e senza fasi di negoziazione, alla base delle sue potenzialità:

## Consistent Overhead Byte Stuffing (COBS)

Esso è un algoritmo per codificare i byte di informazione, che risulta al tempo stesso **efficiente**, **riutilizzabile**, e un **metodo non ambiguo** per definire i **data pack frame** di un pacchetto.

COBS trasforma una stringa arbitraria di byte, ciascuno dei quali ha un Range di valori da [0:255] in una stringa di byte dove però ogni byte va da [1:255].

Avendo eliminato tutti i possibili zeri dal pacchetto, il byte zero diventa un buon candidato per essere usato come terminatore di stringa rendendo il pacchetto COBS-Encoded non ambiguo.



# Conseguenze del protocollo

Avendo ogni pacchetto **Self-Delimited**, diventa possibile scambiare pacchetti diversi tra loro lungo lo stesso stream, a patto che il destinatario sia capace, leggendolo, di determinare il tipo del pacchetto, questo problema è banalmente risolvibile o guardando la semplice *lunghezza della stringa* (se i pacchetti concordati hanno tutti una lunghezza diversa) o aggiungendo all'inizio della trasmissione un *type byte*, ovviamente pre-concordato.

La Codifica aggiunge 2 byte extra alla comunicazione fissa, e ha costo  $O(n)$  sia in codifica che decodifica.

## Vantaggi

1. Pacchetti **Self-Delimited**
2. Canale **Multi-Packet ready**
3. Protocollo **Senza Negoziazioni**
4. All'utilizzatore è richiesto solo di **Pre-Concordare** il formato del pacchetto con l'altro lato dello stream
5. Prerequisiti implementativi minimale (mezzo di comunicazione a bytes di tipo peer2peer asincrono)

## Svantaggi

1. Aggiunge 2 byte fissi
2. Richiede  $O(n)$  elaborazione sia in codifica che decodifica
3. ~~Codice scritto in C++~~



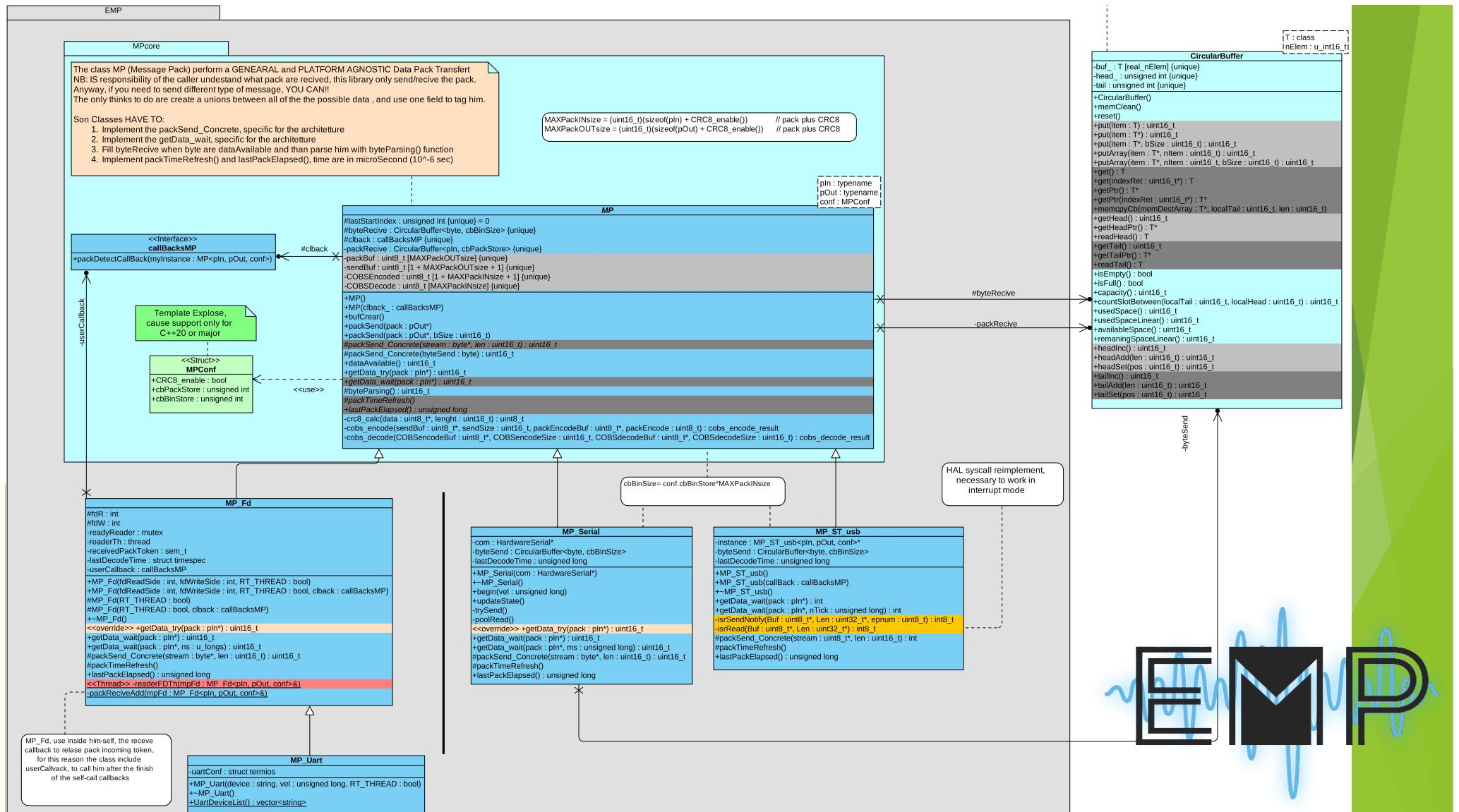
## Pack Check, CRC8

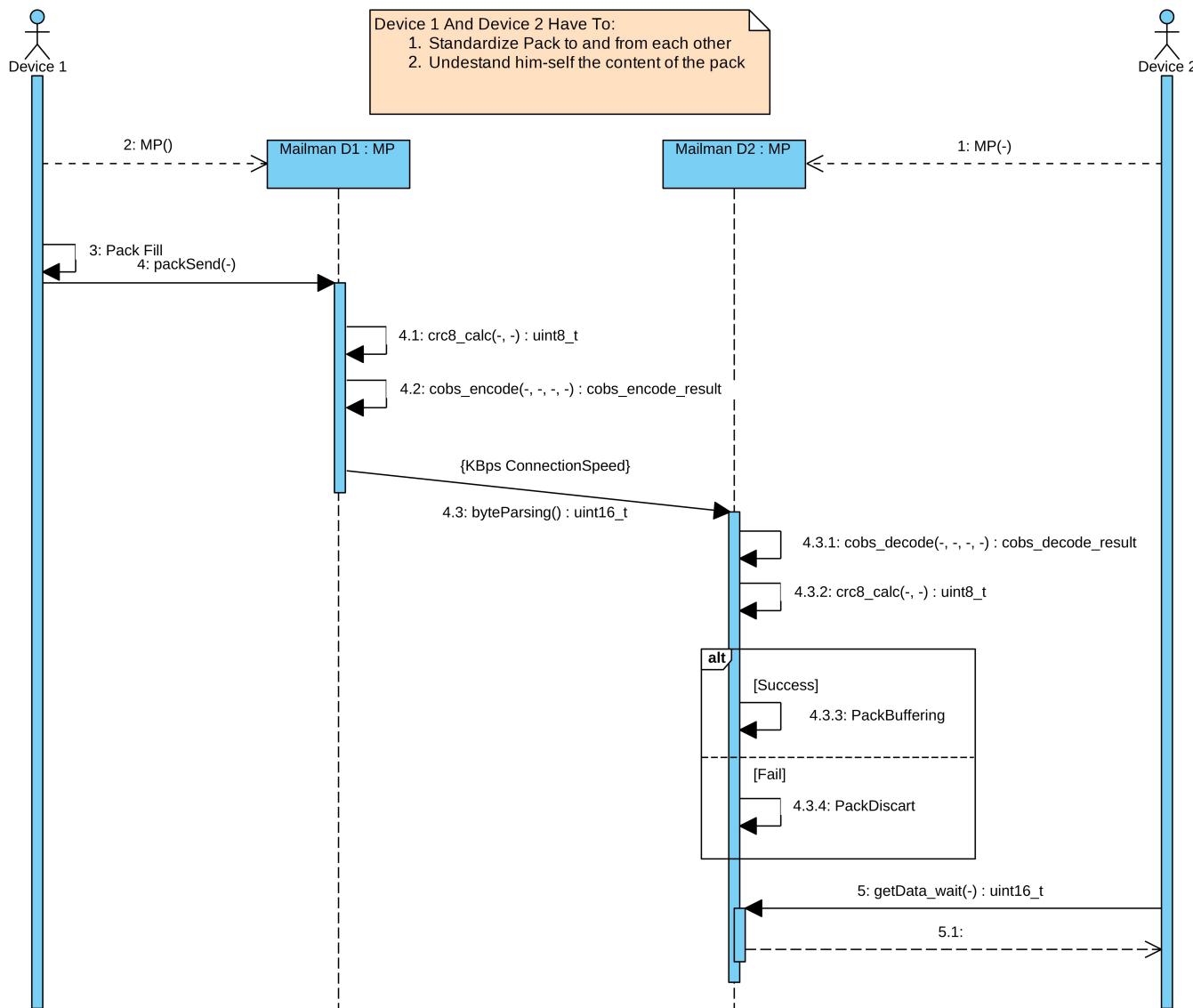
Per aumentare ulteriormente i campi d'uso e garantire un minimo di correttezza sul pacchetto arrivato, la libreria include anche un CRC8 calcolato sullo stream dei dati (non ancora COBS-Encoded) e inviato come parte dello stream (quindi COBS-Encoded) aggiunto alla fine del pacchetto.

Questa features deve essere attiva o disattivata da entrambi i lati della libreria, ed è data come assunto Pre-concordato anch'essa. La libreria, se attiva, è in grado di capire se il pacchetto ha subito degli errori (ovviamente nei limiti del CRC8) e in tal caso scarta il pacchetto ricevuto.

L'aver usato COBS come sistema di codifica per la trasmissione, garantisce che la decodifica debba avvenire solo nei byte compresi tra 2 zeri, e se questa decodifica presenta un errore, toglie ogni ambiguità sul da farsi: il pacchetto viene scartato e si attende un successivo 0 mentre si memorizzano i byte ricevuti nel frattempo.







# EMP Device Support

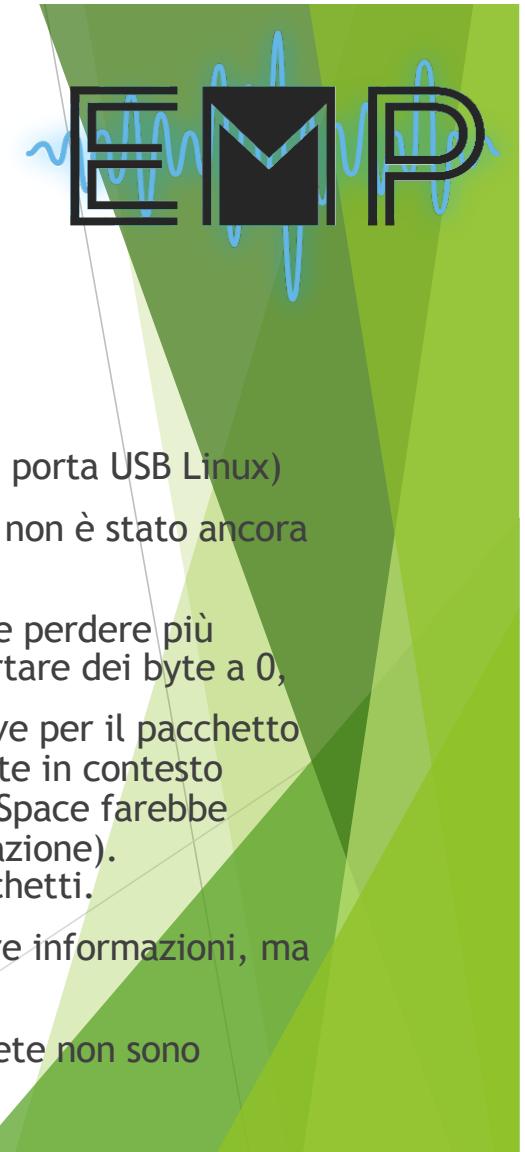
La libreria è stata scritta e testata per funzionare con :

- ▶ Linux Socket File Descriptor (per esempio pipe)
- ▶ Linux UART (Sia USB che UART, usando la libreria termios)
- ▶ Arduino Serial Class (Hardware Serial)
- ▶ STM32 USB Device (Attuale record medio 0,5ms PTS (pack time space) tra STM32 e porta USB Linux)

Anche se in linea di principio questa libreria è estendibile per funzionare tramite SPI, non è stato ancora sviluppato nulla a tal riguardo per una serie di motivazioni:

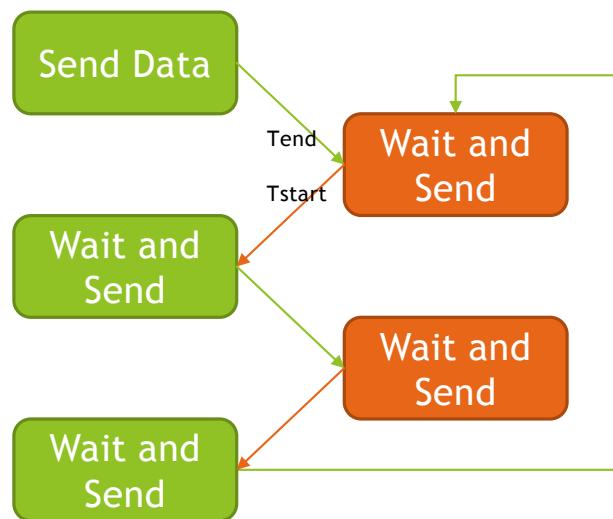
1. La comunicazione è sincrona, e in caso di pacchetti non simmetrici, uno dei 2 deve perdere più tempo del necessario ad inviare 0, caricando di lavoro l'altro device che deve scartare dei byte a 0,
2. Il master della comunicazione **DEVE** far durare la comunicazione tanto quanto serve per il pacchetto più lungo dei 2, il che richiede una fase di contrattazione, (Particolarmente pesante in contesto Linux, dove il driver SPI è kernel-Space, e per comunicare i dati alla libreria User-Space farebbe eseguire in poco tempo molti cambi di contesto,(pesante overhead nella comunicazione). Questo limite rende difficile scendere con semplicità sotto il ms di tempo tra pacchetti.
3. Esiste la possibilità che lo Slave nella comunicazione abbia la necessità di trasferire informazioni, ma deve attendere il permesso del master per farlo.

Discorso simile per altri protocolli di comunicazione come I2C, in cui i membri della rete non sono paritari.



# Speed test

Gli attuali test di laboratorio, replicabili senza problemi prendendo il codice dal repository, consistono nel seguente path di comunicazione:



Usando questo schema, il Round Trip time medio tra i 2 estremi è stato:

- ▶ Linux  $\leftrightarrow$  Arduino  $\approx 7,5$  ms
- ▶ Linux  $\leftrightarrow$  STM32 USB  $\approx 0,4$  ms

Con pacchetti asimmetrici da 14byte e 16byte

Gli attuali test sul prototipo della scheda di controllo, hanno raggiunto la capacità di leggere Linux  $\leftrightarrow$  STM32 USB  $\approx 0,5$  ms come sample rate (deciso precedentemente da interrupt di timer), e un rate di controllo del PWM alla stessa frequenza, includendo CRC8.



Grazie per l'attenzione

