MCU for IoT

Corso Cyber Physical Systems

GIUSEPPE TIPALDI s4141435

Novembre 2019

UNIVERSITÀ
DEGLI STUDI
DI GENOVA

INDICE:

1 INTRODUZIONE:

Nell'ultima decade il mercato dei microcontrollori è letteralmente esploso trovando un'ampia applicazione in diversi settori chiave quali automotive, robotica e IoT. Le diverse architetture disponibili e i moduli hardware che integrano li hanno resi estremamente flessibili quanto potenti.

E' sufficiente una breve ricerca su uno store online per rendersi conto di quanto sia ampia oggi l'offerta relativa ai microcontrollori sia il numero di competitors, solo per citarne qualcuno abbiamo Cypress Semiconductor Corporation, Texas Instruments, STMicroelectronics, NXP Semiconductors e altre svariate industrie TOP del settore.

Inoltre il successo di alcuni prodotti ha dato vita a delle mode (il successo globale di arduino ne è un valido esempio) con le quali si sono diffusi centinaia applicazioni.

Questa monografia sarà suddivisa in due parti, la prima sarà relativa l'ambiente di sviluppo dell'applicazione mentre la seconda parte sarà incentrata sullo sviluppo di piccola applicazione dimostrativa.

2 Development board

Per questo monografia è stata usata una NUCLEO-F401RE prodotta dalla STMicroelectronics, basata su stm32f401 (cortex arm m4 a 84MHz) dispone di 512Kbytes di memoria flash e di 92Kbytes di memoria entrambi integrati nel SoC.

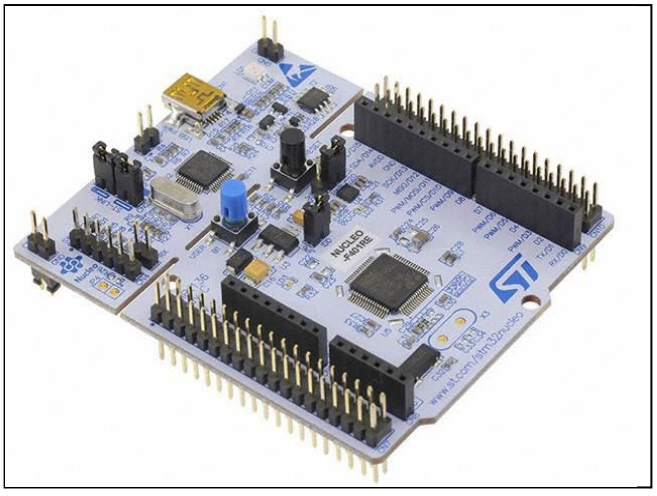

FIGURA 1: board di sviluppo

Ciò che ne rende agevole lo sviluppo è la presenza on-board di un debugger, quest'ultimo semplifica concretamente l'esperienza generale poichè permette di flashare comodamente il microcontrollore; ci fornisce inoltre una comodissima porta COM virtuale. La linea USART2 dell'mcu è collegata al debugger, pertanto

attraverso l'interfaccia virtuale del debug si ha accesso direttamente al microcontrollore.

Caratteristiche tecniche:

MCU	STM32F401RE
Family	ARM Cortex-M4
Vendor	ST Microelectronics
RAM	96Kb
Flash	512Kb
Frequency	up to 84MHz
FPU	yes
Timers	11 (2x watchdog, 1 SysTick, 6x 16-bit, 2x 32-bit [TIM2])
ADCs	1x 12-bit
UARTs	3
SPIs	4
I2Cs	3
RTC	1
Vcc	2.0V - 3.6V
Datasheet	Datasheet
Reference Manual	Reference Manual
Programming Manual	Programming Manual
Board Manual	Board Manual

3 CODING:

Questa parte sarà maggiormente incentrata sulla comprensione degli ambienti di sviluppo che ritengo fondamentali. Diversi sono gli aspetti che passano in secondo piano durante lo studio e l'approccio iniziale con un microcontrollore, tra questi c'è proprio l'ambiente di sviluppo sebbene questa scelta venga in generale giustificata con la necessità di prendere confidenza con le funzionalità e il prodotto.

Programmare un microcontrollore è un qualcosa di ben più profondo del toggle della GPIO associata ad un led.

In generale il primo approccio all'utilizzo di un microcontrollore sarà sempre una successione delle seguenti fasi:

- Scelta di una demo board,
- Installazione dell'IDE proprietario,
- Uso dei vari esempi che generalmente il produttore fornisce a corredo della demo board.
- Applicazione dei tutorial e videotutorial associati all'IDE

Questo è l'approccio classico attraverso cui tutti siamo costretti a passare. Dal mio punto di vista rimanere ancorati all'IDE proprietario ha dei vantaggi iniziali che svaniscono nel momento in cui si usano periferiche di un altro vendor (ad esempio transceiver, touchpad ecc) oppure si va realizzare un'applicazione più avanzata nella quale è necessaria una maggiore ottimizzazione del codice compilato.

Infine si pensi all'eventualità in cui si e costretti a sostituire l'hardware con quello di un altro vendor, la portabilità del codice scritto diventa un problema non banale in una misura che dipende della complessità dell'applicazione.

Escluso il caso di un approccio iniziale, come conseguenza dei limiti esposti è in generale consigliabile nel momento in cui si va a sviluppare un'applicazione complessa usare un ambiente di sviluppo che sia:

- flessibile nel senso che supporti più architetture di diversi produttori.
- attivo, ovvero sia attualmente in continuo sviluppo.
- sostenuto da una community

Questo ambiente è chiaro che non possa essere fornito dal produttore (in quanto è contro la sua logica commerciale) perchè sulle demo board ognuno usa il proprio hardware e generalmente l'IDE proprietario non ha interesse a supportare l'hardware

di un concorrente.

Nella progettazione concreta di una dispositivo l'uso di un componente a discapito di un altro è determinato quasi esclusivamente dal suo prezzo unitario. Il suo costo avrà una rilevanza sempre maggiore al crescere dei volumi.

3.1 IDE PROPRIETARIO:

STMicroelectronics così come fanno i suoi competitors mette a disposizione per il proprio hardware l'IDE *STM32Cube,* discutere dell'IDE proprietario non è nello scopo di questa monografia.

In generale l'aspetto vincente della programmazione di un MCU attraverso l'uso dell'IDE proprietario a mio avviso, consiste nella facilità con cui si configura l'hardware.

3.1 NON IDE:

Da diverso tempo per svariati motivi ho abbandonato Windows passando al mondo open source usando Debian e in seguito Fedora. Per lavoro e per passione nel tempo ho preso confidenza con i vari tool di sviluppo quali git, makefile, cross-compilatori, sia con linux-kernel e il relativo ambiente user-space.

Lavorando e seguendo il mondo embedded è nata la voglia di provare a fornire una piccola documentazione sulla mia esperienza con la demo board della STM

Questo capitolo è stato intitolato il NON IDE, perchè tutto lo sviluppo avviene dal terminale installando di volta in volta esclusivamente il necessario. L'editor di testo che ho usato (anch'esso funzionante da terminale) è vim.

3.2 LibOpenCM3

LibOpenCM3 è una libreria open source a basso livello per l'architettura arm cortex m3 (oggi supporta vari cortex). Inizialmente era nata esclusivamente per gli stm32 ma è stata evoluta e il supporto esteso anche ad altri prodotti non STM.

E' sufficiente osservare nell'ordine lo stato del progetto su GITHUB, il wiki ed infine la documentazione delle API per farsi un'idea dello stato della libreria e le sue potenzialità.

Mi sono posto l'obiettivo di creare una piccola libreria che contenga il minimo indispensabile per iniziare:

1. Setup del sistema:
   Ipotizzando di disporre di una distribuzione linux vergine è necessario installare i tool di sviluppo necessari per gestire il progetto, compilarlo, ed infine scaricarlo sul microcontrollore.

   Nel mio caso, data l'uso di Fedora 30:

```
$ sudo dnf update
$ sudo dnf install arm-none-eabi-binutils-cs arm-none-eabi-newlib \
arm-none-eabi-gcc-cs arm-none-eabi-gcc-cs-c++
$ sudo dnf install git-core make stlink make patch vim
```

   snippet 1: setup del sistema

2. Checkout del codice:
   Consiglio di prendere visione dei repository che il team di sviluppo di LibOpenCM3 mette a disposizione. Oltre il principale che consiste nei fatti nella libreria, sono rilevanti il repository **libopencm3-examples** e **libopencm3-template**.
   Nella mia esperienza di lavoro con questa libreria, ho eseguito un fork del progetto di template rinominandolo _stm32_app. Successivamente ho adeguato con vai commit il codice alle mie esigenze rendendolo univoco per la board NUCLEO-F401RE

   Procediamo al clone del mio repository _stm32_app:

```
$ git clone https://github.com/Ciusss89/_stm32_app.git
$ cd _stm32_app
```

   snippet 2: checkout _stm32_app

3. Base-Code:
   Il progetto denominato ***_stm32_app*** presente sul mio profilo github è un fork del progetto template di libopencm3 nel quale è stato implementata la seguente struttura:

   a. Directory *app*: contiene il main e il Makefile. Questi ultimi due file rappresentano il cuore del programma che verrà compilato e scritto sul dispositivo. Al suo interno è anche presente una directory sia i vecchi main che mi sono per sviluppare e testare questa piccola bozza di libreria.

b. Directory *common-code*: contiene i moduli per scrivere sullo stdio e gestire alcuni degli 11 timers presenti sulla nucleo. L'idea è provare ad espanderla nel tempo.
c. Directory *libopencm3*: una volta effettuato il checkout del codice essa risulta vuota. Qui risiede la libreria e va inizializzata come definito nel file read.me

4. Utilizzo:
   La Nucleo-F401RE possiede oltre all'mcu un chip dedicato esclusivamente al debug e alla programmazione del suo microcontrollore. Una delle 3 porte di comunicazione seriale (nello specifico la UART2) è collegata al chip di debug (commercialmente noto come STLink).

   Connettendo la nucleo al proprio pc attraverso il cavo usb, il chip di debug creerà nei device linux la porta di comunicazione */dev/ttyACM0*

   Usando un qualsiasi terminale e instaurando una comunicazione con la porta di comunicazione precedente, avremo (una volta programmato) un feedback visivo dello stato del microcontrollore.

5. La libreria LibOpenCM3 è diffusa, ottimizzata per l'hardware STM32, dispone di molteplici esempi ed applicazioni. Lo svantaggio che emerge con l'uso di questa libreria sta nell'approccio iniziale che in funzione del proprio background di competenze personali può risultare complicato è in alcune situazioni sicuramente complesso.

   E' necessario comprendere di volta in volta cosa serve, come va configurato ed infine usato.

   Questa fase iniziale è decisamente dispendiosa in termini di tempo, tuttavia gli esempi messi a disposizione sono molti.

6. LibOpenCM3 è un progetto ancora attivo e mantenuto, ma lo sviluppo è ha subito un rallentamento consistente negli ultimi due anni. Sono subentrati sulla scena a partire dal 2015 altri progetti più interessanti che hanno una struttura modulare e supportano più dispositivi e funzionalità

3.3 Real Time Operative System

Nonostante l'incremento prestazionale degli mcu, i dispositivi IoT rimangono oggetti dalle risorse hardware estremamente limitate. In particolare questi non hanno le risorse sufficienti per permettere il funzionamento di sistemi Linux-Like come OpenWRT (distribuzione linux diffusissima per hardware embedded) e suoi derivati, o di altri sistemi operativi più complessi

Negli ultimi anni sono diversi le soluzioni open source che propongo sistemi real time per IoT. Tra queste i progetti di maggiore successo sono RIOT, Contiki, Zephyr, mbed OS, FreeRTOS, e TinyOS.
In generale un sistema operativo IoT per definirsi tale deve costruito attorno a punti cardine quali l'astrazione dell'hardware, il supporto networking e il power management. Il tutto deve funzionare in presenza di risorse hardware limitate.
Le complessità che un sistema real time deve gestire emergono con con facilità dal confronto tra un sistema basato su microcontrollore (mcu) e un system on a chip (SoC), risulta un fattore di 10^6 di memoria in meno, di 10^3 capacità computazionali inferiori e infine 10^3 in meno di consumi (quest'ultimo è un aspetto positivo).

Un sistema operativo per IoT deve incontrare il giusto bilanciamento tra i seguenti punti chiave.

1. Performance: è prioritario avere un'efficiente gestione della memoria finalizzata a garantire un'elevata reattività e consumi energetici ridotti.
2. Networking: un sistema IoT avrà bisogno nel caso più semplice di una connessione verso internet, scambiare messaggi e interagire con i suoi simili nei casi più complicati. Serviranno quindi gli stack di comunicazione ad esempio un 802.3 (classico ethernet) o 802.11/15 (WiFI/BLE).
3. Flessibilità: dovrà essere sviluppato per essere flessibile nel senso di indipendenza dall'hardware sia per integrarsi con altre librerie e software di terze parti.
4. Sicurezza & Privacy: negli ultimi anni sono stati innumerevoli gli attacchi lanciati da botnet di tipo ddos. Gli attacchi erano lanciati da sistemi IoT non aggiornati e con carenze relative alla sicurezza. Questo aspetto avrà sempre più rilevanza nel tempo in quanto è tra quelli che più facilmente si trascura per risparmiare tempo e denaro.

3.4 RIOT-OS

RIOT-OS è il sistema operativo per IoT che ho scelto per sviluppare l'applicazione presentata nel capitolo successivo:

"*RIOT is an operating system designed for the particular requirements of Internet of Things (IoT) scenarios. These requirements comprise a low memory footprint, high energy efficiency, real-time capabilities, a modular and configurable communication stack, and support for a wide range of low-power devices. RIOT provides a microkernel, utilities like cryptographic libraries, data structures (bloom filters, hash tables, priority queues), a shell, various network stacks, and support for various microcontrollers, radio drivers, sensors, and configurations for entire platforms, e.g. TelosB or STM32 Discovery Boards.*"

Come accennato, nell'ultima decade abbiamo assistito alla diffusione capillare di piattaforme hardware in grado di soddisfare tutte le molteplici richieste del mercato; in termini prestazionali sia di costo. Dal punto di vista del software si è assistito ad un processo di standardizzazione e astrazione dell'hardware. E' in questa ottica che ha trovato spazio e successo soluzioni come:

- Windows: il sistema operativo per più diffuso (vincolato inizialmente all'architettura x86).
- OpenWrt: il sistema operativo per dispositivi router/ap/nas.
- Android: il sistema operativo per dispositivi multimediali.
- RIoT-OS: si candida a essere il sistema operativo per dispositivi embedded.

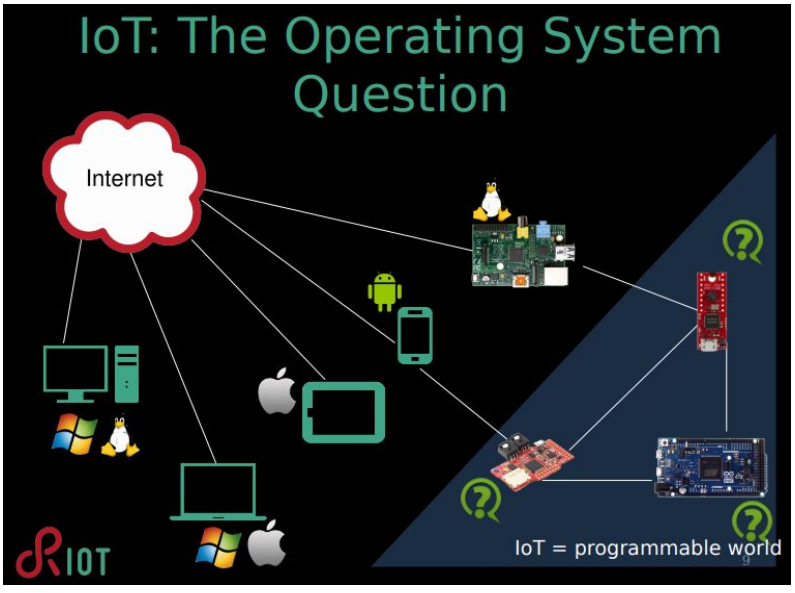

FIGURA 2: RIOT-OS SLIDE

RIOT implementa i più rilevanti standard open source ed è stato sviluppato seguendo lo standard ANSI C, C++. Questo permette all'intero progetto di avere una notevole flessibilità ed efficienza, lo stesso codice può essere compilato per un mcu a 8-bit come quello usato da arduino ma anche per un mcu a 16-bit come la serie texas instruments MSP430 o ancora per un potente ARM a 32-bit.

Il vantaggio che comporta l'avere un buon livello di astrazione lo si intuisce se si pensa per esempio all'uso di una periferica hardware come il blocco adc.
Nella scrittura dell'applicazione non mi è richiesto di scendere a basso livello per configurare il blocco adc poiché questo è uno dei compiti eseguiti dal driver della periferica.

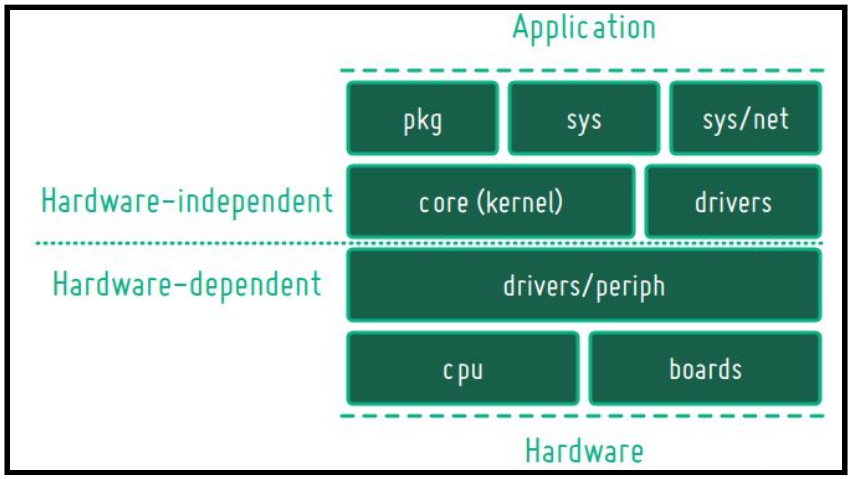

FIGURA 3: RIOT-OS ABSTRACTION

Questo approccio permette di supportare un numero considerevole di piattaforme hardware, di fatti esiste un canale di sviluppo dedicato all'estensione della compatibilità RIOT-OS boards. Un'altro punto di forza di RIOT-OS consiste nella presenza di uno stack network completo:

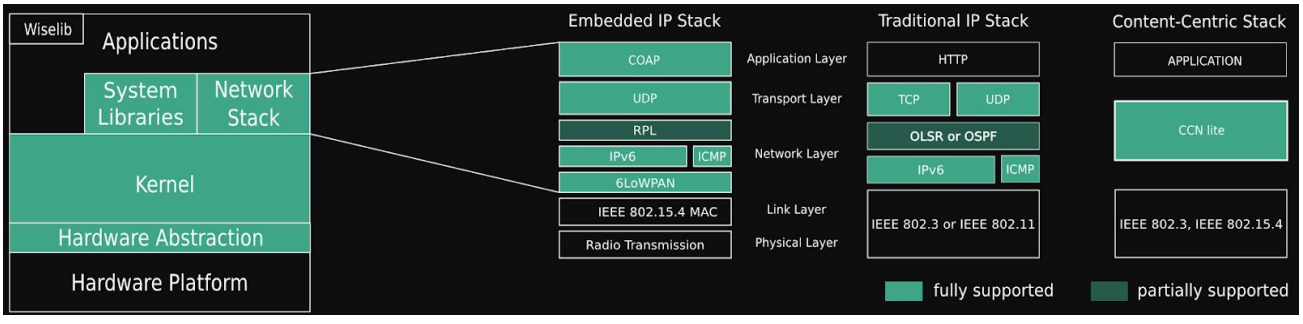

FIGURA 4: RIOT-OS network stack

Per ulteriori approfondimenti su RioT-OS rimando alla documentazione presente online e alle pubblicazioni disponibili nella bibliografia.

4 Energy Meter:

L'applicazione dimostrativa realizzata è un energy meter. L'obiettivo che mi sono posto è stato quello di costruire l'hardware sia di progettare il software necessario a misurare i consumi elettrici di un elettrodomestico.

Nell'ambito di un impianto elettrico civile, avremo un sistema monofase a corrente alternata a 50Hz. Normalmente la potenza che il gestore dell'energia fornisce ad una abitazione è pari 3,2KW. Superata questa soglia dopo qualche minuto il contatore elettrico interviene interrompendo la fornitura.
La misura del consumo elettrico istantaneo prevede l'acquisizione di del valore di corrente e tensione istantanea. Ricordo che per un circuito in regime sinusoidale la potenza sarà mediata sul suo periodo, i fasori di tensione e corrente non saranno mai in fase ma sfasati a causa della componente reattiva degli utilizzatori. Ricordando il teorema di BOUCHEROT si definisce:

- Potenza Attiva: potenza dissipata dalla parte resistiva dell'utilizzatore in regime sinusoidale, si misura in Watt [W].

$$P = V\ I\ cos\ \phi$$

  L'angolo di sfasamento ϕ tra tensione e corrente dovrebbe in norma essere prossimo a 0.95. Gli utilizzatori che hanno un'elevata reattanza (qualsiasi elettrodomestico dotato di un motore) deve essere rifasato, di norma questo viene fatto con un reattanza capacitiva che compensa quella induttiva.
  La potenza attiva è quella che ci interessa misurare perché quella che paghiamo. Essa viene trasferita sull'utilizzatore ed in esso trasformata nel tempo in energia termica, meccanica o altre forme di energia.

- Potenza Apparente: consiste nel prodotto dei valori efficaci di tensione e corrente senza considerare il loro sfasamento, si misura in VoltaAmper VA.

$$S = V I$$

- Potenza Reattiva: quando la potenza apparente è maggiore di quella attiva significa che abbiamo un carico fortemente sfasato (oltre che non a norma).
  Questo contributo di potenza è continuamente "palleggiata" tra generatore ed utilizzatore, non è associata ad alcun effetto pratico al contrario della potenza attiva. Si misura in VoltAmper Reattivi (VAR)

$$Q = V\ I\ sen\ \phi$$

In conclusione, siamo interessati alla sola misura della sola potenza attiva. Una sua accurata misura richiede quindi l'acquisizione di tensione, corrente e l'angolo di sfasamento tra i due fasori; nella pratica l'errore commesso approssimando l'angolo di sfasamento come una costante pari a 0,90 è trascurabile.

4.1 Hardware

Per la valutazione della potenza attiva è necessario eseguire una misura della corrente di linea e della tensione di linea. In questo progetto si è realizzata la sola misura della corrente di linea tralasciando quella della tensione, ciò nonostante l'applicazione software è stata redatta supponendo di disporre di entrambe le misure.

In letteratura esistono diverse soluzioni, tra le disponibili ho scelto la più semplice ed economica per dedicare maggiori risorse allo sviluppo dell'applicazione.

Ho utilizzato un classico *current transformer* (CT), ovvero una applicazione del principio che sta alla base del trasformatore. Il primario è costituito dal filo della fase; Quando in esso scorre una corrente, il flusso da questa generato viene raccolto e guidato nel toroide (che costituisce il circuito magnetico) che lo contorna.
A sua volta il flusso indotto, induce una corrente nel circuito secondario costituito da un filamento metallico che ha decine di centinaia di avvolgimenti attorno al toroide. La corrente indotta viene scalata di un fattore pari al rapporto di trasformazione, è in generale presente anche un termine di sfasamento.

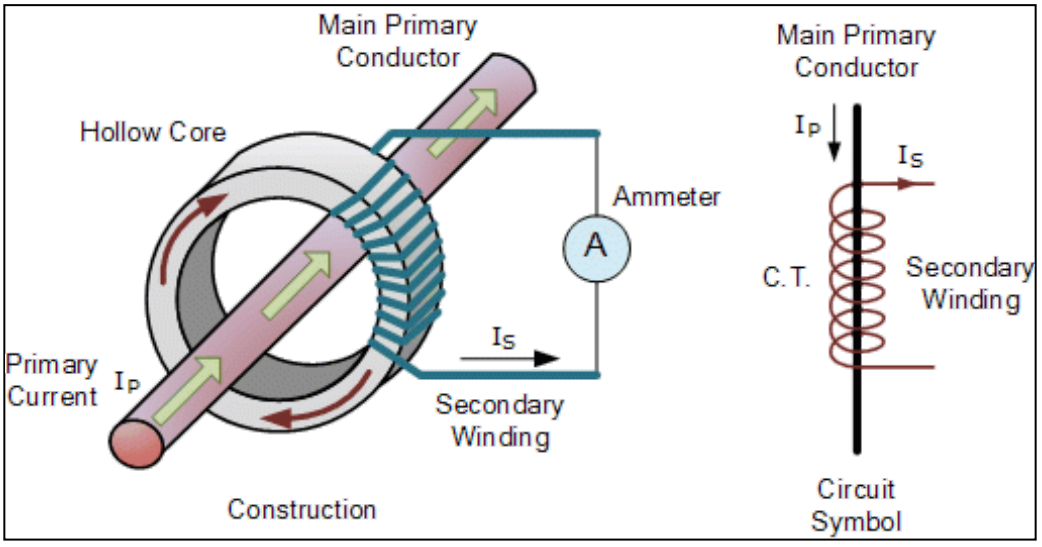

FIGURA 5: Principio fisico

Per procedere al campionamento del valore di corrente del secondario è necessario trasformare questa corrente in una tensione equivalente. Il modo più semplice e immediato di farlo cosiste nell'utilizzo di un resistore (I legge di Ohm), in alternativa qualora fosse necessario maggiore accuratezza e precisione nella misura si deve ricorrere a soluzioni basate sul concetto dell'amplificatore di transimpedenza. Per realizzare la sonda che ho usato, sono sufficienti 4 componenti oltre il CT:

1. Due resistori (R_A e R_B) servono a polarizzare il CT a metà della tensione di alimentazione. Il loro valore è di poco conto, è sufficiente che siano uguali. Nel mio caso ho usato due resistori da 10KΩ.
2. Un resistore di carico (R_{BURDEN}) per il circuito secondario del CT. Questo è l'unico elemento che va dimensionato con più attenzione. Il resistore deve essere scelto in modo tale da rappresentare in modo corretto tutta la dinamica di interesse (espressa in tensione).
3. Un condensatore elettrolitico (C_1) nell'ordine del μF, io ne ho usato uno da 1μF è consigliabile stare nell'intorno delle decina di microfard, non eccederei oltre i 22μF.

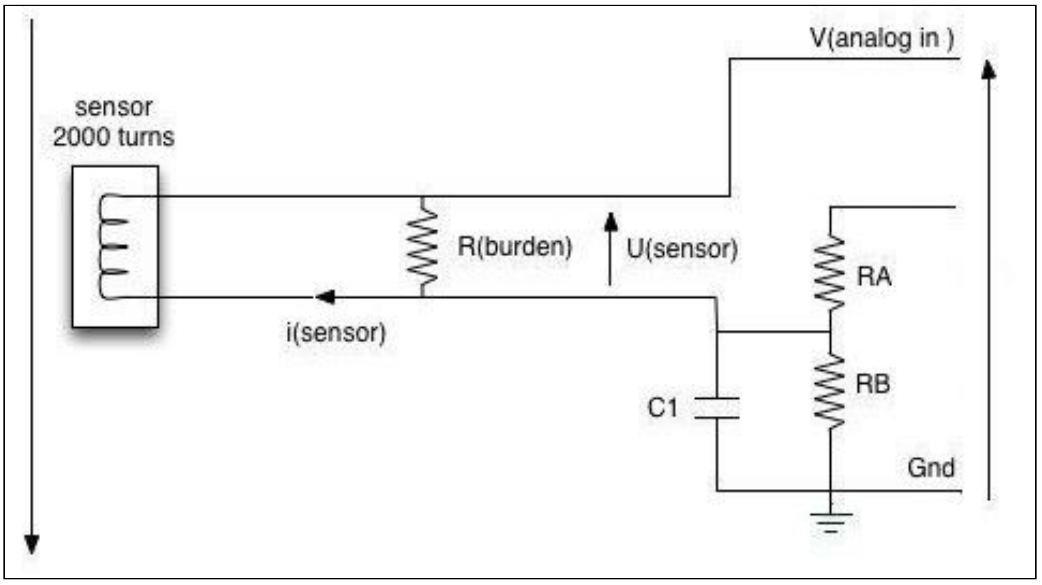

FIGURA 6: schema elettrico

La regola per dimensione il resistore di carico è banale, va tenuto in considerazione il rapporto di trasformazione del CT, il valore di polarizzazione desiderato ed infine la massima dinamica. Nel mio caso, ho usato un current transformer YHDC TA1020

Di seguito è rappresentato la natura e le caratteristica dei segnali che andremo a campionare:

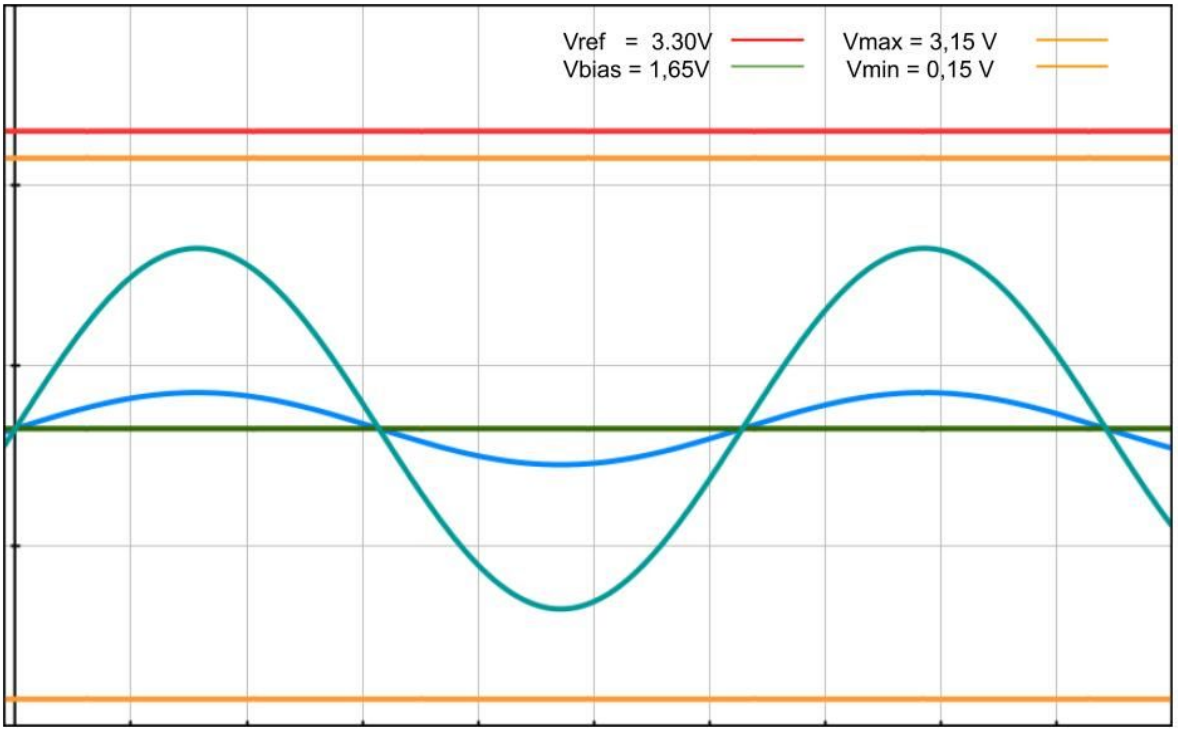

FIGURA 7: segnali

L'ADC del microcontrollore ha una dinamica d'ingresso massima compresa tra 0 e 3,3 V. Il resistore di carico per il current transformer è stato scelto in modo da:

- Evitare di sfruttare l'intera dinamica dell'adc. Nonostante l'stm32 usi un adc rail-to-rail ho deciso di limitare la dinamica dai 3,3V (V_{REF} in rosso) potenziali a 3V. La tensione sinusoidale generata avrà quindi un picco inferiore al più pari a 0.15V ed uno superiore al più di 3.00V (riferimenti arancioni).
- La tensione sinusoidale generata, voglio sia centrata a metà dell'alimentazione, quindi intorno a 1,65V (V_{BIAS} in verde scuro)

Nel precedente grafico, In blu e in ciano sono rappresentate due ipotetiche sinusoidi corrispondenti a due diversi valori di carico.

In conclusione, il resistore di carico viene dimensionato nel seguente modo:

1. $I_{PRIMARIO_{MAX}} = I_{RMS} * \sqrt{2} \rightarrow Per\ ipotesi\ I_{RMS} = 7A$
2. $I_{SECONDARIO_{MAX}} = \frac{I_{PRIMARIO_{MAX}}}{CT_{RATIO}} \rightarrow CT_{RATIO} = 1000$
3. $R_{BURDEN} = \frac{1,5}{I_{SECONDARIO_{MAX}}} \rightarrow R = 151\ \Omega$

Il valore 151 non è un valore di resistenza commerciale, consiglio di ottenere tale valore suddividendo il lavoro tra due resistori, per esempio usandone uno da 100Ω in serie ad un trimmer da 100Ω regolando quest'ultimo per valore più preciso.

L’utilizzo del trimmer permette di avere sufficiente gioco nel momento in cui volessi cambiare la massima corrente misurabile oltre che facilitare il dimensionamento.

Nella figura che segue è rappresentata un'anteprima finale del circuito elaborata dal software fritzing.

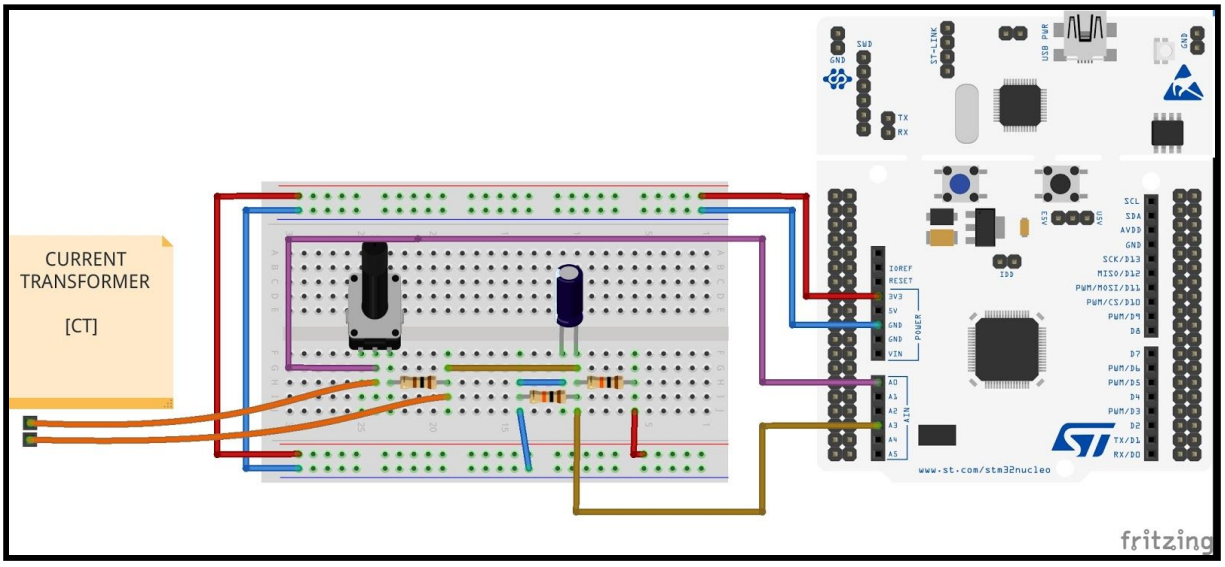

FIGURA 8: schematic preview

Di seguito troviamo la realizzazione sperimentale del prototipo, la prolunga ha sul cavo di fase il current transformer :

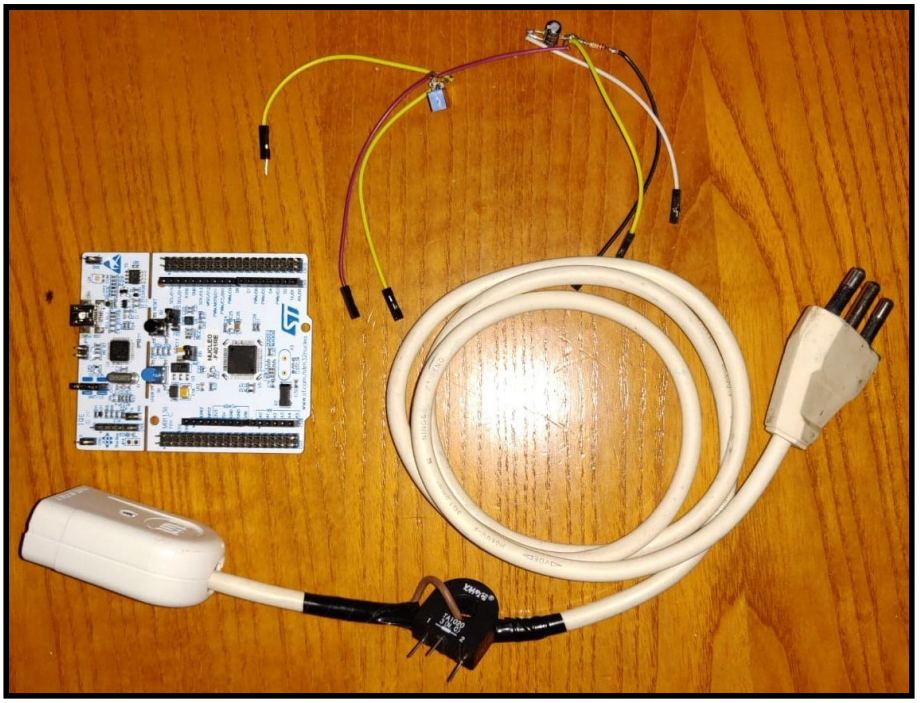

FIGURA 9: prototipo

4.2 EnergyMeter APP

L'energy meter (em) è una semplice un'applicazione real time avviabile da shell basata su riot-os. E' costituita da due thread, il primo dedicato al campionamento dei valori di tensione e corrente mentre il secondo è un tread di log.

L'applicazione è stata sviluppata come un nuovo applet shell di riot-os, pertanto essa sarà visibile nella lista dei comandi disponibili da shell. La versione minimale della shell è costituita dai comandi *reboot* e *ps* ai quali si è aggiunto il nuovo *em*

```
Run `help` to see a list of all available commands

> help
help
Command              Description
---------------------------------------
em                   em - energy meter application
reboot               Reboot the node
ps                   Prints information about running threads.
```

snippet 3: shell di riot os in esecuizione sull'mcu

Descrizione dell'applicazione:

1. Codebase: dopo aver effettuato il checkout del codice mediante git, il codice sarà così suddiviso.

   topdir:

```
~/mcu/_riot_app (master) $ tree -L 1
.
├── bin
├── energy_meter
├── LICENSE
├── main.c
├── Makefile
├── media
└── README.md

3 directories, 4 files
```

snippet 4: code preview dell'app

FILE:	DESCRIZIONE
bin	Contiene i binari prodotti dal processo di compilazione tra questi

	abbiamo il firmware che sarà effettivamente scritto sull'mcu.
energy_meter	Contiene i sorgenti dell'applicazione em
LICENSE	Contiene il file di licenza, questo progetto sua la licenza GNU AGPLv3
main.c	definisce e aggiunge il nuovo applet per la shell di riot
Makefile	file per il processo di compilazione.
media	media: direcotry di supporto per il file README.md
README.md	file in markdown

energy_meter:

```
~/mcu/_riot_app/energy_meter (master) $ tree -L 1
.
├── core.h
├── ct1020.h
├── em.h
├── main.c
├── Makefile
└── measure.c

0 directories, 6 files
```

snippet 5: code preview dell'app

FILE:	DESCRIZIONE
core.h	Come il nome suggerisce è il file l'header principale, al suo interno sono presenti le API, le due strutture dati e le varie costanti.
ct1020.h	Descrive le caratteristiche del current transformer YHDC TA1020
em.h	Definisce l'entita energy meter per il precedente main.
main.c	Costituice l'applicazione in oggetto.
Makefile	file per il processo di compilazione.
measure.c	Contiene le funzioni di inizializzazione e di sampling.

Per comprendere nel dettaglio i *Makefile* consiglio di fare riferimento alla documentazione di Riot-OS, dalla quale si può apprezzare l'efficacia e la pulizia di una struttura modulare ed astratta.
Il main file della topdir è molto semplice, esso definisce ed aggiunge all shell di riot-os il nuovo USEMODULE energy meter.

2. Energy Meter, main file:

   Il main è abbastanza semplice, è composto da due funzioni di inizializzazione e da due tread:

   I. Funzioni di inizializzazione, la prima si occupa del setup della parte relativa al current transformer, la seconda configura il blocco adc.
   II. Vengono avviati nel seguente ordine il tread per il sampling e il tread per il logging. La lora esecuzione non sarà mai interrotta.

   Il programma si completa stampando nella shell i valori del struttura *em_realtime.* Terminato il programma i tread rimangono attivi in background, rilanciando nuovamente l'applicazione constata la presenza dei due tread verranno verrà stampato direttamente il contenuti della struttura *em_realtime.*

   pseudocodice:

```
ct_sensor_setup;

IF (adc_setup < 0)
        fail;

IF (pid_sampling == -1)
        IF (bias_check < 0)
               fail;

IF (pid_sampling == -1)
        thread_create(em sampling);

IF (pid_sampling > 0 && pid_logging == -1)
        thread_create(em log minute);

print_data;

return 0;
```

snippet 6: psedudocodice main

3. Energy Meter: tread sampling

   L'unico elemento di complessità consiste nel tread di sampling. Sussiste il problema del corretto campionamento di un segnale sinusoidale con frequenza pari a 50Hz (periodo di 20 msec).

   Ricordando Nyquist, è necessario campionare ad una frequenza almeno pari a due volte la frequenza massima presente nel segnale originale.

In questa particolare applicazione poiché non ho bisogno di ricostruire il segnale campionato, la regola di Nyquist non rappresenta una forte sottostima della reale frequenza minima di campionamento (non ho un filtro ricostruttore).

L'idea che ho implementato nella soluzione consiste nel determinare la frequenza di campionamento automaticamente in relazione al numero di campioni desiderati (si osservi la costante *SAMPLE_UNIT*) per costante di tempo.

Ottenute le misurazioni di tensione e corrente, è necessario calcolare il valore quadratico medio del set di dati acquisiti. Si è implementata in forma algoritmica la seguente formula.

$$V_{RMS} = \sqrt{\frac{1}{N} \sum_{i=1}^{12} v[i]^2}$$

La soluzione realizzata è la più semplice ma non la migliore, in letteratura esistono implementazioni più fini che mirano a minimizzare l'uso delle risorse hardware sia il rumore che inevitabilmente affligge la misura.
Il thread di sampling, ogni secondo esegue un loop dalla durata esatta di 20msec nel quale vengono acquisiti 12 campioni di corrente e 12 di tensione.
Il thread di sampling salva i valori correnti nella struttura statica *em_realtime.*

4. Energy Meter: tread logging
   Il thread di logging si occupa di leggere ogni secondo i valori correnti presenti nella struttura statica *em_realtime*, per poi inserirli in un buffer. Ogni minuto viene eseguita la media di questi valori ed i risultati pubblicati nella struttura statica *em_realtime.*

5. Energy Meter: debug
   All'interno dell'applicazione è presente una modalità di debug, la costante VERBOSE è definita dal file core.h

```
#define VERBOSE       1U       /* VALUES: [0,1,2,3,4] */
```

snippet 7: file energy_meter/core.h

Al crescere del valore vengono stampate di volta in volta più informazioni, si osservi che il livello 4 corrompe il loop di acquisizione poiché la system call *printf* è molto dispendiosa in termini di risorse.

4.3 Modalità sviluppo:

Questa ultima parte fornisce le informazioni necessarie per continue lo sviluppo e il miglioramento dell'applicazione.

1. Setup del sistema:
   La fase di preparazione è uguale a quella già eseguita per LibOpenCM3, inoltre è riportata nel file *READ.ME.*
   Ho lavorato con Fedora che fa uso del gestore pacchetti *dnf* il quale è diverso da *apt* che è usato da debian (e ubuntu).

```
$ sudo dnf update
$ sudo dnf install arm-none-eabi-binutils-cs arm-none-eabi-newlib \
arm-none-eabi-gcc-cs arm-none-eabi-gcc-cs-c++
$ sudo dnf install git-core make stlink make patch vim
```

snippet 8: setup sistema

2. Checkout del codice sorgente:
   Anche questa parte è riportata nel file *READ.ME*, sarà necessario ottenere i sorgenti di RIOT e app che ho sviluppato:

```
$ mkdir app
$ cd app
$ git clone https://github.com/RIOT-OS/RIOT.git
$ git clone https://github.com/Ciusss89/_riot-os_app.git
```

snippet 9: checkout sorgenti

3. Inizializzazione:
   Una volta scaricato il codice sorgente attraverso git è necessario selezionare la release con la quale si intende lavorare (git di default sceglie master). Io ho sviluppato l'app usando la release *2019.10-branch* di riot

```
cd RIOT/
git checkout <LATEST_RELEASE>
cd ../_riot-os_app
git checkout v1.0
```

snippet 10: setup sorgenti

4. Clean, Compile, Flash:
   Ottenuto il codice e dopo averlo inizializzato è sufficiente digitare il il seguente comando "*make clean all*" nella direcotory dell'applicazione per compile il

progetto e creare l'eseguibile da flashare sulla nucleo.

```
~/mcu/_riot_app (master) $ make clean all
Building application "riot-os_app" for "nucleo-f401re" with MCU "stm32f4".
"make" -C /home/giuseppe/mcu/RIOT/boards/nucleo-f401re
"make" -C /home/giuseppe/mcu/RIOT/boards/common/nucleo
"make" -C /home/giuseppe/mcu/RIOT/core
"make" -C /home/giuseppe/mcu/RIOT/cpu/stm32f4
"make" -C /home/giuseppe/mcu/RIOT/cpu/cortexm_common
"make" -C /home/giuseppe/mcu/RIOT/cpu/cortexm_common/periph
"make" -C /home/giuseppe/mcu/RIOT/cpu/stm32_common
"make" -C /home/giuseppe/mcu/RIOT/cpu/stm32_common/periph
"make" -C /home/giuseppe/mcu/RIOT/cpu/stm32f4/periph
"make" -C /home/giuseppe/mcu/RIOT/drivers
"make" -C /home/giuseppe/mcu/RIOT/drivers/periph_common
"make" -C /home/giuseppe/mcu/RIOT/sys
"make" -C /home/giuseppe/mcu/RIOT/sys/auto_init
"make" -C /home/giuseppe/mcu/RIOT/sys/div
"make" -C /home/giuseppe/mcu/RIOT/sys/isrpipe
"make" -C /home/giuseppe/mcu/RIOT/sys/newlib_syscalls_default
"make" -C /home/giuseppe/mcu/RIOT/sys/pm_layered
"make" -C /home/giuseppe/mcu/RIOT/sys/ps
"make" -C /home/giuseppe/mcu/RIOT/sys/shell
"make" -C /home/giuseppe/mcu/RIOT/sys/shell/commands
"make" -C /home/giuseppe/mcu/RIOT/sys/stdio_uart
"make" -C /home/giuseppe/mcu/RIOT/sys/tsrb
"make" -C /home/giuseppe/mcu/RIOT/sys/xtimer
"make" -C /home/giuseppe/mcu/_riot_app/energy_meter
   text         data          bss          dec          hex      filename
  25280          508         6228        32016         7d10
/home/giuseppe/mcu/_riot_app/bin/nucleo-f401re/riot-os_app.elf
```

snippet 11: building

La compilazione è un processo veloce, si completa in una manciata di secondi.
Una volta compilato ed ottenuto il file eseguibile (*.elf) è sufficiente digitare il comando "*make flash*" e collegare mediante cavo usb la nucleo board.

```
~/mcu/_riot_app (master) $ make flash
/home/giuseppe/mcu/RIOT/dist/tools/openocd/openocd.sh flash
/home/giuseppe/mcu/_riot_app/bin/nucleo-f401re/riot-os_app.elf
### Flashing Target ###
Open On-Chip Debugger 0.10.0
Licensed under GNU GPL v2
For bug reports, read
      http://openocd.org/doc/doxygen/bugs.html
hla_swd
Info : The selected transport took over low-level target control. The
```

```
results might differ compared to plain JTAG/SWD
adapter speed: 2000 kHz
adapter_nsrst_delay: 100
none separate
srst_only separate srst_nogate srst_open_drain connect_deassert_srst
srst_only separate srst_nogate srst_open_drain connect_assert_srst
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : clock speed 1800 kHz
Info : STLINK v2 JTAG v28 API v2 SWIM v18 VID 0x0483 PID 0x374B
Info : using stlink api v2
Info : Target voltage: 3.260646
Info : stm32f4x.cpu: hardware has 6 breakpoints, 4 watchpoints
    TargetName         Type       Endian TapName            State
--  ------------------ ---------- ------ ------------------ ------------
 0* stm32f4x.cpu       hla_target little stm32f4x.cpu       reset
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
adapter speed: 1800 kHz
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x0800050c msp: 0x20000200
auto erase enabled
Info : device id = 0x10006433
Info : flash size = 512kbytes
target halted due to breakpoint, current mode: Thread
xPSR: 0x61000000 pc: 0x20000046 msp: 0x20000200
wrote 32768 bytes from file
/home/giuseppe/mcu/_riot_app/bin/nucleo-f401re/riot-os_app.elf in
1.165904s (27.447 KiB/s)
target halted due to breakpoint, current mode: Thread
xPSR: 0x61000000 pc: 0x2000002e msp: 0x20000200
target halted due to breakpoint, current mode: Thread
xPSR: 0x61000000 pc: 0x2000002e msp: 0x20000200
verified 25788 bytes in 0.244432s (103.029 KiB/s)
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
Info : Unable to match requested speed 2000 kHz, using 1800 kHz
adapter speed: 1800 kHz
shutdown command invoked
Done flashing
```

snippet 12: flashing

RIOT fa uso del tool *openocd* per flashare l'mcu. Un feedback relativo all'operazione lo si ottiene grazie al led *LD1,* da rosso diventa verde nel momento in cui la programmazione si coclude positivamente.

Anche in questa situazione emerge come riot-os è stato ben pensato, i comandi sono semplici ed intuitivi.

5. Apertura della comunicazione verso la Nuleo-F401RE

   La comunicazione con la nucleo board può avvenire usando un tool di RIOT, oppure in alternativa usano il comando *picocom*. La differenza sta nel fatto che il tool di RIOT aggiunge un time stamp all'output generato dell'mcu.

   Di seguito l'outup del comando "*make term"*

```
~/mcu/_riot_app (master) $ make term
/home/giuseppe/mcu/RIOT/dist/tools/pyterm/pyterm -p "/dev/ttyACM0" -b "115200"
Twisted not available, please install it if you want to use pyterm's JSON capabilities
2019-11-23 11:02:46,528 # Connect to serial port /dev/ttyACM0
Welcome to pyterm!
Type '/exit' to exit.
help
2019-11-23 11:02:55,208 # help
2019-11-23 11:02:55,211 # Command              Description
2019-11-23 11:02:55,214 # ---------------------------------------
2019-11-23 11:02:55,218 # em                   em - energy meter application
2019-11-23 11:02:55,222 # reboot               Reboot the node
2019-11-23 11:02:55,227 # ps                   Prints information about running threads.
```

snippet 13: riot-os term

   Segue l'output del comando "*picocom -b 115200 /dev/ttyACM0 --imap lfcrlf*"

```
~/mcu/_riot_app (master) $ picocom -b 115200 /dev/ttyACM0 --imap lfcrlf
picocom v3.1

port is        : /dev/ttyACM0
flowcontrol    : none
baudrate is    : 115200
parity is      : none
databits are   : 8
stopbits are   : 1
escape is      : C-a
local echo is  : no
noinit is      : no
noreset is     : no
hangup is      : no
nolock is      : no
send_cmd is    : sz -vv
receive_cmd is : rz -vv -E
imap is        : lfcrlf,
omap is        :
emap is        : crcrlf,delbs,
logfile is     : none
initstring     : none
exit_after is  : not set
```

```
exit is        : no

Type [C-a] [C-h] to see available commands
Terminal ready


> main(): This is RIOT! (Version: 2020.01-devel-728-g5a705-2019.10-branch)
RIOT on a nucleo-f401re board, MCU stm32f4
> help
Command              Description
---------------------------------------
em                   em - energy meter application
reboot               Reboot the node
ps                   Prints information about running threads.
>
```

snippet 14: picocom term

Consiglio di premere il tasto reset dopo aver aperto la comunicazione.

Attraverso questi semplici passi si è dimostrato come dopo aver ottenuto il codice è possibile compilarlo ed eseguirlo sulla NUCLEO-F401RE.

4.4 Demo:

In questo ultimo paragrafo verrà eseguita una dimostrazione usando come utilizzatore un asciugacapelli da 1800W. Si osserverà come la corrente misurata vari in funzione del suo utilizzo:

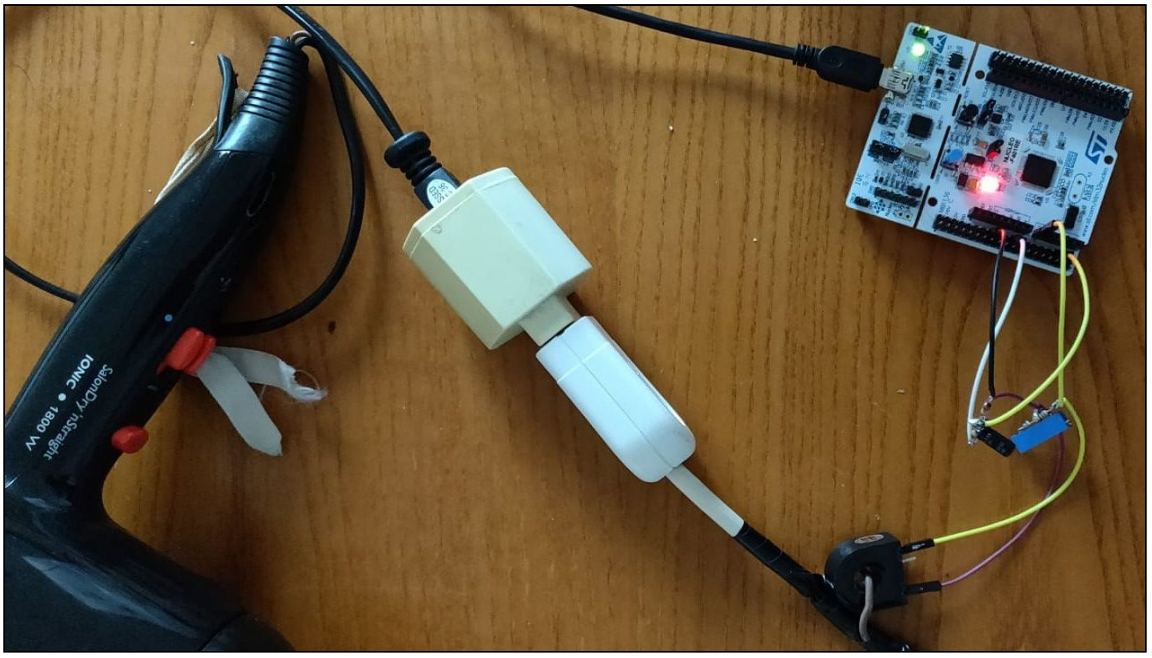

FIGURA 10: demo

Ricordo che il setup attuale permette di misurare una corrente massima al più pari a 7A, per modificare questo limite si deve agire sul trimmer. Quest'ultimo va configurato in modo tale da ottenere il valore di resistenza calcolato (tale valore viene stampato nella modalità di debug 1).

Temp	Speed	Corrente misurata [A]
off	low	0.497
low	low	0.887
low	max	3.887
max	max	6.314

Usato l'asciugacapelli attraverso la prolunga predisposta per la misura ecco i valori realtime misurati:

```
> main(): This is RIOT! (Version: 2020.01-devel-728-g5a705-2019.10-branch)
RIOT on a nucleo-f401re board, MCU stm32f4
> em
Starting EnergMeter service...
[*] Energy Measuring: sampling has started
[*] Energy Measuring: log minute has started
Current 0.000A
Voltage 0.000V
> em
Starting EnergMeter service...
[!] Energy Measuring: sampling already started, pid=3
[!] Energy Measuring: log minute already started, pid=4
Current 0.497A
Voltage 230.000V
> em
Starting EnergMeter service...
[!] Energy Measuring: sampling already started, pid=3
[!] Energy Measuring: log minute already started, pid=4
Current 0.887A
Voltage 230.000V
> em
Starting EnergMeter service...
[!] Energy Measuring: sampling already started, pid=3
[!] Energy Measuring: log minute already started, pid=4
Current 3.887A
Voltage 230.000V
> em
Starting EnergMeter service...
[!] Energy Measuring: sampling already started, pid=3
[!] Energy Measuring: log minute already started, pid=4
Current 6.314A
```

```
Voltage 230.000V
```

snippet 15: demo

Infine si osservi l’applicazione compilata con la modalità di debug livello 1, in questa condizione vengono stampate tutte le informazioni di inizializzazione:

```
> em
[###] DEBUG LEVEL=1
Starting EnergMeter service...
[*] CT sensor setup:
        RMS MAX current: 7A
        Max primary peak current: 9.899495A
        Max secondary peak current: 0.009899A
        Burden resistor: 151.522873Ω
[*] ADC setup:
        ADC bits: 12
        ADC bias offset: 2048
        ADC scale factor: 0.000733
        ADC sampling frequency: 600HZ
        ADC gets [12] sample each 1666 usec
[*] Calibration loop:
        ADC(3) bias boundary: [1.61-1.69]V
        Bias readed voltage: 1.6382V
[*] Energy Measuring: sampling has started
[*] Energy Measuring: log minute has started
Current 0.000A
Voltage 0.000V
```

snippet 16: debug

5 BIBLIOGRAFIA:

-	Descrizione	url
01	Main page LibOpenCM3	link
02	LibOpenCM3 wiki page provides by github	link
03	LibOpenCM3 tutorials, examples on github	link
04	LibOpenCM3 documentation	link
05	Main page RIOT-OS	link
06	GitHub RIOT-OS	link
07	RIOT-OS wiki page provides by github	link
08	RIOT-OS board branch	link
09	RIOT-OS documentation	link
10	Paper: A Memory-Efficient True-RMS Estimator in a Limited-Resources Hardware	link
11	Paper: The RIOT Approach to Ubiquitous Networking for the IoT	link
12	IEEE: A real-time kernel for wireless sensor networks employed in rescue scenarios	link
13	SUMIT: SOFIE Secure and Open Federation of IoT systems	link
14	Paper: RIOT: an Open Source OS for Low-end Embedded Devices in the IoT	link
15	AN: RMS Calculation for Energy Meter Applications Using the ADE7756	link
16	Custom APP LibOpenCM3	link
17	Custom APP RIOT-OS	link