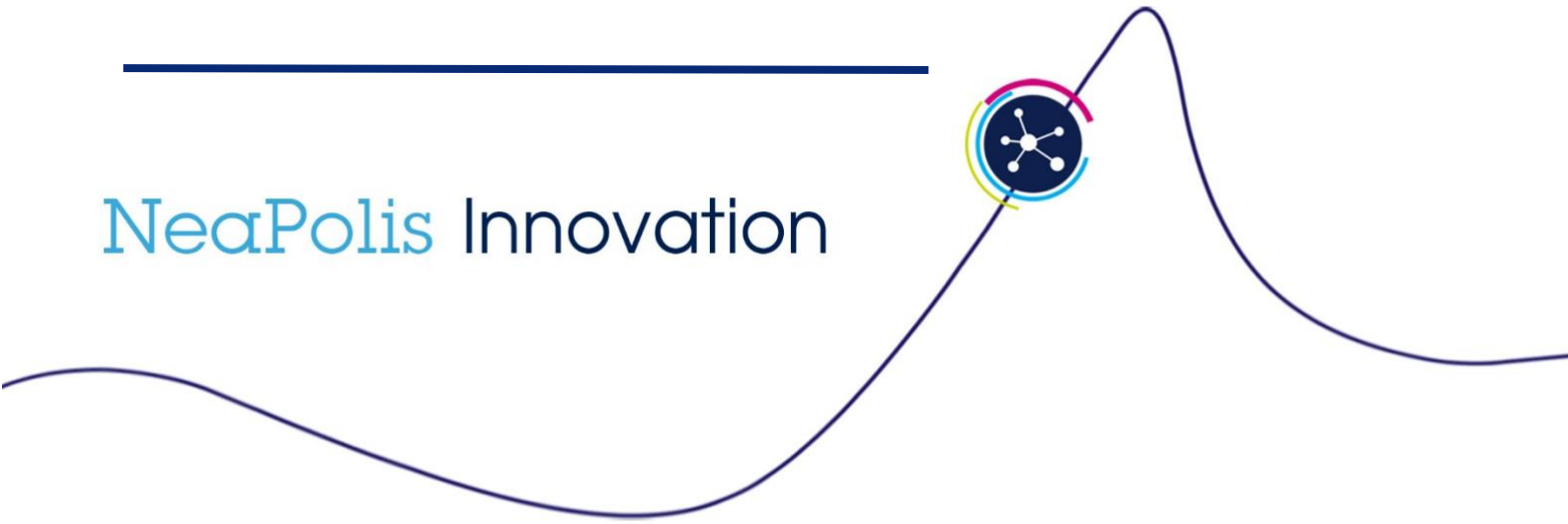
Relazione NISC2021

NeaPolis Innovation



STMicroelectronics

Autori:

Leonardo Catello – N46004862

Daiana Cipollaro – N46004941

Sommario

1. INTRODUZIONE

Il NeaPolis Innovation Summer Campus 2021 è stato svolto interamente online, attraverso la piattaforma di videoconferenze Microsoft Teams.
Il campus si è svolto dal 26 agosto 2021 al 5 settembre 2021. Le giornate prevedevano dei seminari intensivi sui microcontrollori a 32 bit e sui Sistemi Operativi Real-Time.

Dopo una prima fase preliminare di seminari e lavoro individuale, i tutor ci hanno raggruppati in team per sviliuppare un progetto. A tale scopo, ogni partecipante ha ricevuto una scheda di sviluppo e delle componenti elettroniche.

I team sono stati scelti in maniera casuale dai tutor, permettendoci di lavorare con studenti italiani e stranieri con un background completamente diverso e con maggior esperienza.

1.1. LA SCHEDA DI SVILUPPO

La scheda che ci è stata fornita è una STM32 Nucleo-64 (NUCLEO-F401RE). Si tratta di una scheda appartenente alla famiglia di microcontrollori a 32 bit basati su architettura ARM Cortex-M.

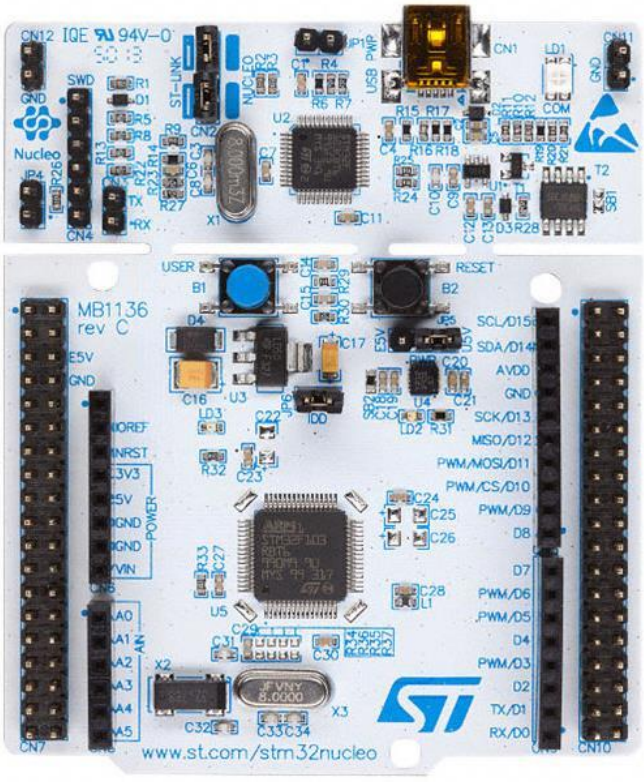


Figura 1 - STM32 NUCLEO-F401RE

1.2. L'AMBIENTE DI SVILUPPO

L'ambiente di sviluppo che abbiamo utilizzato per programmare e controllare la nostra scheda è "*ChibiStudio*": un insieme di programmi basati su Eclipse per il supporto di ChibiOS, un sistema operativo real-time molto compatto ed efficiente per progettare applicazioni embedded. Il principale linguaggio di programmazione utilizzato è stato il C.
ChibiStudio è composto da:

- Eclipse IDE;
- GNU GCC ARM Compiler & Tools;
- Open OCD Tool & scripts: software open-source per effettuare il debugging di device embedded;

- ChibiOS: fornisce una serie di librerie per sfruttare al massimo il potenziale della scheda Nucleo e per semplificare e rendere più veloce la scrittura del codice di controllo.

ChibiStudio inoltre presenta due tipologie di prospettive:

- C/C++ development: per la creazione e la gestione dei progetti, per la scrittura e la compilazione del codice e per la correzzione di errori di sintassi.
- Debug: per testare le funzionalità ed individuare comportamenti non attesi e per correggere errori funzionali.

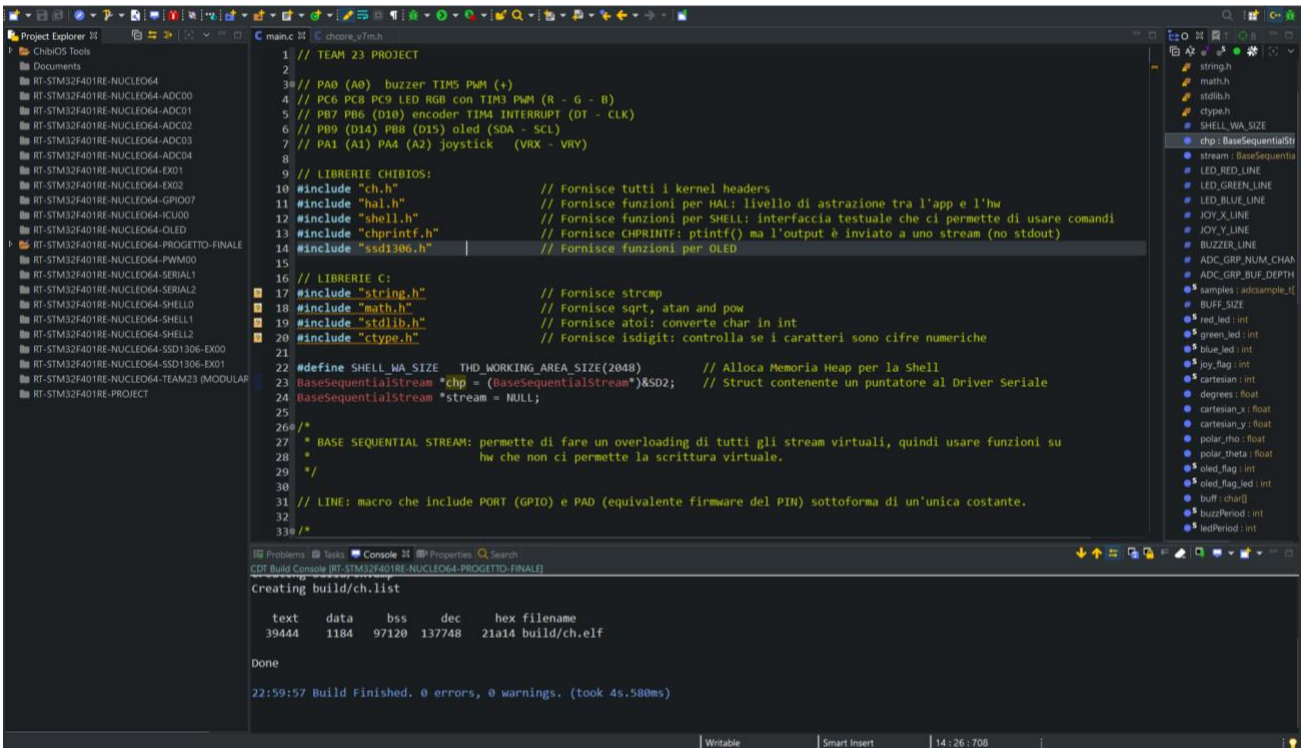


Figura 2 - ChibiStudio - C/C++ Development perspective

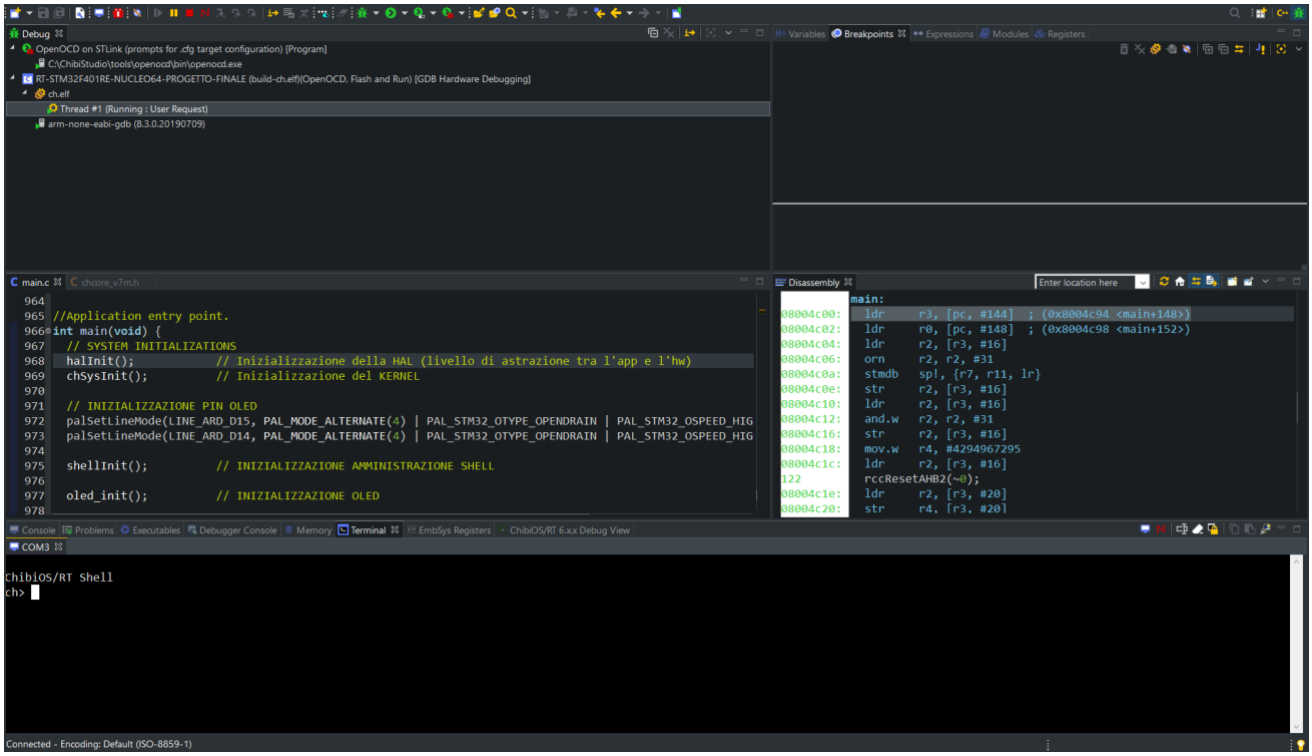


Figura 3 - ChibiStudio – Debug Perspective

1.3. L'ANATOMIA DI UN PROGETTO

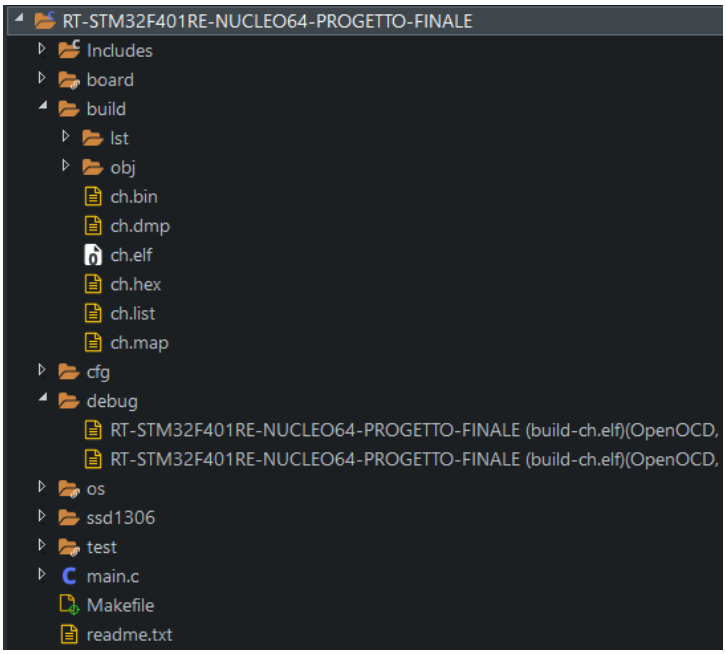


Figura 4 – struttura di un progetto

Insieme di cartelle:

- Debug folder: contiene la configurazione d'esecuzione.
- Headers di configurazione:
  - chconf.h: principali impostazioni relative al kernel;
  - halconf.h: configurazioni relative ai driver di ChibiOS ed HAL;
  - mcuconf.h: configurazioni strettamente legate al MCU
- Makefile: contiene uno script che fornisce una serie di direttive usate dal compilatore per effettuare il build iniziale del codice.
- Main: il codice sorgente, contenente l'entry point dell'applicazione.

2. STM32 GPIO

GPIO, "General Purpose Input Output", è un'interfaccia informatica hardware che *consente ai dispositivi* come i microprocessori *di interagire con un'altra periferica*.

I pin di I/O della STM Nucleo sono organizzati in gruppi di 16 elementi (da 0 a 15), e ciascuno di questi pin possono essere configurati come Input o Output, in maniera completamente indipendente.

Ogni gruppo è chiamato "Port" ed è identificato da una lettera (GPIOA, GPIOB, GPIOC…). I pin invece sono identificati dalla combinazione della lettera P, dall'indentificatore della porta (A,B,C,…) e dall'identificatore del pin (0,1,2,…15):

PF 12

2.1. GPIO – modalità di funzionamento

È possibile programmare ciascun pin in 4 differenti modalità:

- **Input Mode**, consente di campionare il livello logico di un pin.
  - Il buffer di output viene disattivato;
  - Il trigger TTL Schmitt viene acceso;
  - I pin sono continuamente campionati e conservati in una memoria che può essere acceduta e letta per ottenere lo stato del pin.
- **Output Mode**, consente di impostare il livello logico di un pin.
  - Il buffer di output viene attivato;
  - Il trigger TTL Schmitt viene acceso;
  - Può essere cambiato lo stato del pin;
  - Si può controllare lo stato elettrico del pin.
- **Analoge Mode**, consente di usare ADC o DAC.
- **Modalità alternata**, consente di assegnare un pin a una periferica di STM32.
  - STM32 è dotata di un grande numero di periferiche, connesse al GPIO attraverso dei multiplexler;
  - Ogni periferica è mappata su più di un pin per garantire una maggiore flessibilità.

2.2. GPIO – caratteristiche elettriche

Nella STM32 al microcontrollore viene fornita una tensione di 3,3V.
Lo stato GPIO alto è di 3,3V mentre quello basso è di 0V. I GPIO tollerano anche tensioni che arrivano fino a 5V e la corrente massima che un singolo pin è in grado di generare/assorbire è di circa 25 mA → la corrente totale assorbita/generata da tutti i pin non deve superare i 120 mA.

2.3. RGB led

Uno dei componenti utilizzati durante il campus è stato il led RGB, il quale è formato da 3 differenti colori a led. Il led può essere collegato ai pin d'ingresso, in serie con delle resistenze, e fornendo un segnale alto o basso (di tensione) si può rispettivamente accendere e spegnere il led.

Figura 7 – led RGB

3. ChibiOS Pal Driver

Il **PAL Driver**, Port Abstraction Layer è un driver di ChibiOS che usa il GPIO. Come suggerisce il nome, il PAL driver *fornisce un'astrazione della struttura hardware* della scheda per facilitare la progettazione del codice.

Sono presenti diversi metodi di identificazione ed il rispettivo insieme di funzioni:

- Relativi ai Pad che agiscono su in singolo I/O (port,pad)
- Relativi ai Group che agiscono su un gruppo di I/O (port, mask, offset)
- Relativi alle Port che agiscono su un'intera Port (port)
- Relativi alle Line che sono un'alternativa al Pad (X).

Configurazioni per cambiare i Pin:

- palSetPadMode(port, pad, mode)
- palSetLineMode(line, mode)
- palSetGroupMode(port, mask, offset, mode)
- palSetPortMode(port, mode)

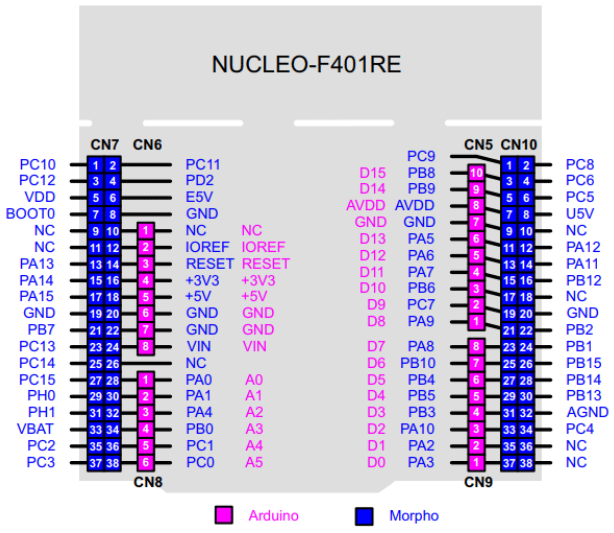


Figura 8 – Schema Pinout della Nucleo 64

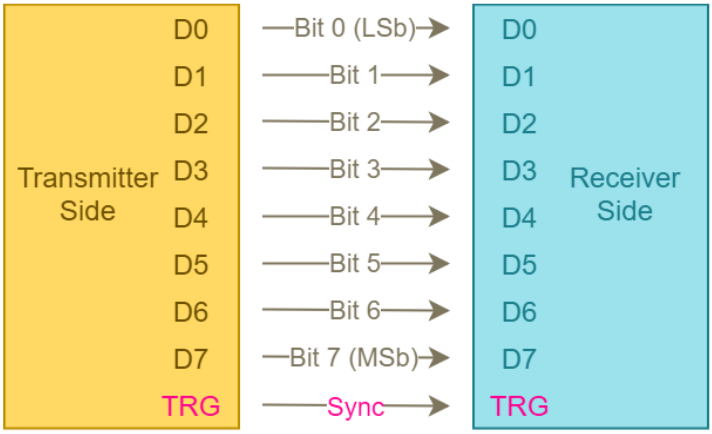
3.1. ChibiOS Pal Driver – Startup Configuration

Ogni pad deve essere opportunamente configurato prima di essere utilizzato. La configurazione per ogni pad deve seguire le specifiche e le semantiche della board.
L'*inizializzazione* iniziale è effettuata dalla funzione **halInit()**.

4. STM32 USART

Lo **UART** o Universal Asynchronous Receiver-Transmitter è un dispositivo hardware, di uso generale o dedicato, che *converte flussi di bit di dati da un formato parallelo a un formato seriale* asincrono o viceversa.

Comunicazione parallela: ogni bit viene trasmesso attraverso una linea dedicata. Richiede una linea per ogni bit più una linea per la sincronizzazione.



Comunicazione seriale: i bit vengono trasmessi attraverso una singola linea, chiamata bus. Richiede dei meccanismi di sincronizzazione più complessi.

La sincronizzazione seriale può essere: sincrona o ascincrona.
Mentre la comunicazione può essere semplice, half-duplex o full-duplex.

L'**USART** è una variante dell'UART che dispone di un clock addizionale. È una periferica progettata per implementare vari protocolli seriali.

4.1. ChibiOS Serial Driver

Un **Seriale** è un driver di ChibiOS/HAL che usa l'USART. Il driver seriale *memorizza nel buffer i flussi* di input ed output utilizzando delle code di I/O. Ed Ogni API (Application Programming Interface) del driver seriale generalmente inizia con il prefisso "sd".

Per utilizzare un seriale bisogna abilitarlo in "halconf.h": HAL_USE_SERIAL → TRUE:

```
/**
 * @brief   Enables the SERIAL subsystem.
 */
#if !defined(HAL_USE_SERIAL) || defined(__DOXYGEN__)
#define HAL_USE_SERIAL                      TRUE
#endif
```

Per utilizzare il driver invece bisogna assegnarlo ad una periferica in "mcuconf.h":

```
/*
 * SERIAL driver system settings.
 */
#define STM32_SERIAL_USE_USART1             FALSE
#define STM32_SERIAL_USE_USART2             TRUE
#define STM32_SERIAL_USE_USART6             FALSE
#define STM32_SERIAL_USART1_PRIORITY        12
#define STM32_SERIAL_USART2_PRIORITY        12
#define STM32_SERIAL_USART6_PRIORITY        12
```

Prima di utilizzarli, ogni driver seriale deve essere inizializzato e configurato. L'*inizializzazione* è effettuata automaticamente chiamando la funzione **hallinit()** nel main. La *configurazione* invece viene effettuata dall'utente utilizzato la funzione *sdStart()*. Tale funzione riceve in ingresso due parametri: un puntatore al driver che vogliamo avviare ed un puntatore alla struttura di configurazione.

4.2. Shell in ChibiStudio

Una **shell** è una interfaccia testuale che si basa su un flusso di I/O, in questo caso, offerto dal driver seriale. È uno strumento potente, che permette di creare eventi asincroni, ovvero che si verificano indipendentemente dal flusso del programma principale, inserendo dei comandi da tastiera.

Per utilizzare una Shell in ChibiStudio è necessario aggiungere il path della libreria "shell.h" all'interno del Makefile:

```
115 include $(CHIBIOS)/os/hal/lib/streams/streams.mk
116 include $(CHIBIOS)/os/various/shell/shell.mk
117
```

Per utilizzarla in maniera corretta, bisogna:

- **Allocare memoria**: bisogna definire la memoria heap in cui la Shell verrà allocata.

```
#define SHELL_WA_SIZE   THD_WORKING_AREA_SIZE(2048)          // Alloca Memoria Heap per la Shell
```

- **Definire una lista dei comandi**: è un vettore che contiene i nomi dei comandi che la Shell si aspetta di ricevere dalla tastiera e i nomi delle relative funzioni.

```
// COMANDI DELLA SHELL
static const ShellCommand commands[] = {
    {"LED", cmd_led},
    {"JOY", cmd_joy},
    {"OLED", cmd_oled},
    {"BUZZ", cmd_buzz},
    {"DIMMER", cmd_dimmer},
    {"DEMO", cmd_demo},
    {NULL, NULL}                                    // Se non viene inserito nulla, non fa nulla
};
```

- **Configurare una struttura contenete il driver seriale e il vettore dei comandi**.

```
// STRUTTURA DI CONFIGURAZIONE SHELL (interfaccia testuale che ci permette di usare comandi):
static const ShellConfig shell_cfg1 = {
    (BaseSequentialStream*)&SD2,
    commands
};
```

- **Creare un nuovo thread**, con parametri:
    - Puntatore all'heap;
    - Dimensione della Working Area;
    - Identificatore del Thread;
    - Priorità;
    - Funzione del Thread;
    - Argomenti (passati con cast a void).

```
// Normal main() thread activity, spawning shells
while (true) {
  thread_t *shelltp = chThdCreateFromHeap(NULL, SHELL_WA_SIZE, "shell", NORMALPRIO + 1, shellThread, (void*)&shell_cfg1);

  chThdWait(shelltp); // Waiting termination
}
```

La memoria allocata, non viene rilasciata automaticamente, ma è responsabilità del thread, il quale deve chiamare la funzione chThdWait() e dopo rilasciare la memoria allocata.

- **Definire comandi personalizzati**, i parametri sono:
  - Puntatore allo stream seriale;
  - Massimo numero di stringhe che la shell può ricevere ;
  - Vettore che contiene le varie stringhe che sono state inserite da tastiera.

```
// FUNZIONE OLED (usando SSD1306 OLED Display)
static void cmd_oled(BaseSequentialStream *chp, int argc, char *argv[]) {
```

5. STM32 ADC

Un **Convertitore Analogico-Digitale** *converte un segnale continuo nel tempo in una sequenza di valori discreti.* È necessario un convertitore perché i segnali discreti sono direttamente interpretabili dal microcontrollore.

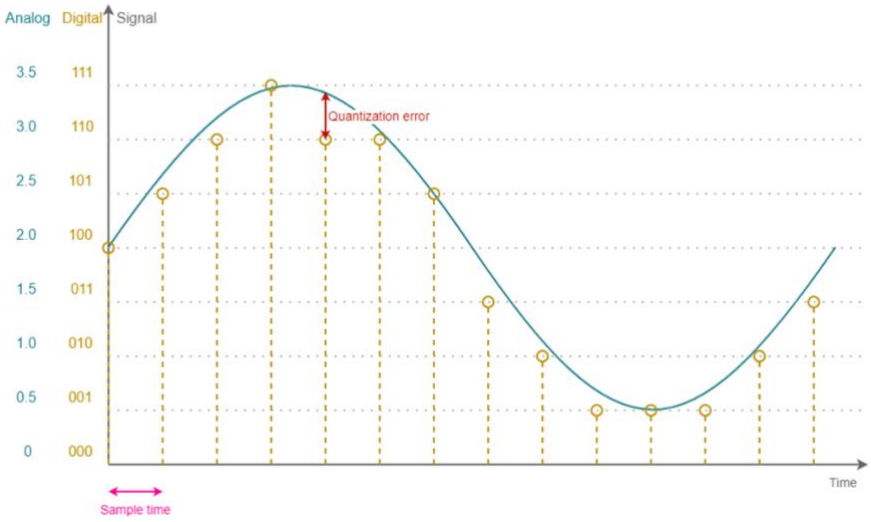


Figura 9 – grafico di un segnale quantizzato

Specifiche di un convertitore ADC:

- **FSR** (Full Scale Voltage Range), massimo range di valori analogici che può essere dato in ingresso:

$$V_{FSR} = V_{inputMax} - V_{inputMin}$$

- **Risoluzione**, numero di valori discreti che l'ADC può produrre nell'intervallo consentito di valori d'ingresso analogico. La risoluzione in termini di tensione:

$$\Delta V = \frac{V_{FSR}}{risoluzione}$$

- **Sampling rate**, cadenza di campionamento, misurato in S/s (samples per second) e i suoi multipli (kS/s, MS/s or GS/s).

In particolare, la scheda è equipaggiata con un "Successive Approximation ADC" a 12-bit, composto da:

- Sample & Hold, campionatore e interfaccia tra segnale analogico e ADC;
- Digital to Analog Converter;
- Comparatore, riceve in input due segnali e li compara;
- SAR (Successive Approximation Register).

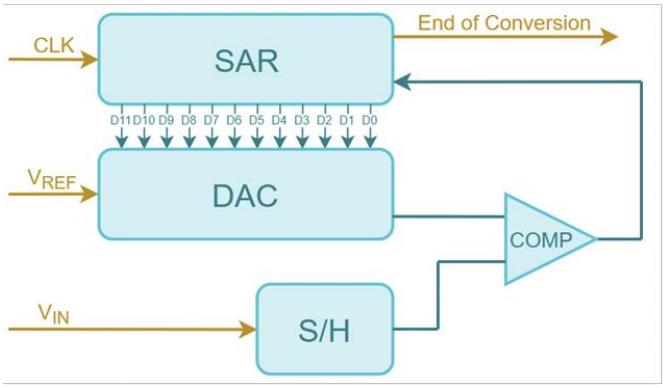


Figura 10 – Schema a blocchi dell'ADC della STM32

La SMT32 fornisce due modalità di conversioni:

- Singola conversione: l'ADC fa una sola conversione e poi si ferma.
- Continua conversione: l'ADC inizia una nuova conversione quando l'ultima è terminata.

Inoltre, ogni SMT32 fornisce molte sorgenti di input da utilizziare come canali per effettuare la conversione. In particolare, ogni canale è identificato con un numero progressivo che parte da 0. I primi 16 canali sono generalmente assegnati a fonti esterne.

5.1. JOYSTICK

Il **joystick** è un altro dispositivo utilizzato durante il campus e che è stato fornito nello starter kit.

Esso è un semplice esempio di periferica analogica: usa due potenziometri, essi variano la resistenza nel circuito e indicano la posizione del joystick rispetto agli assi X ed Y.

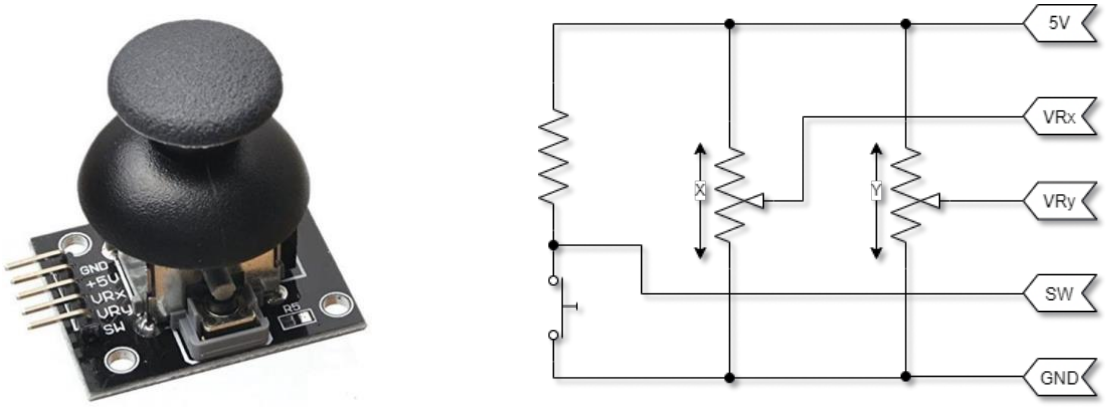


Figura 10 – joystick e schema circuitale

6. Comunicazione I2C

L'**I2C** (Inter-Integrated Circuit) è un bus di comunicazione seriale, half-duplex e sincrono. Utilizza un'architettura di tipo master-slave, con un unico master. Il segnale di clock è generato da uno degli endpoint ed è fornito agli altri tramite una linea specifica (SCL).

La comunicazione avviene tramite un'unica linea (SDA), spesso chiamata seriale a due fili. Una transazione è composta da uno o più messaggi, in cui ogni messaggio è composto da una parola di un byte più un bit (ACK/NACK) aggiuntivo.

L'intero sottosistema del driver I2C può essere abilitato mediante l'apposito campo in "halconf.h" (HAL_USE_I2C → TRUE).
Si può assegnare il driver ad una specifica periferica, modificando "mcuconf.h".

Ogni operazione del driver può essere eseguita solo se il driver è stato *configurato* correttamente.

```
// STRUTTURA DI CONFIGURAZIONE I2C:
const I2CConfig i2ccfg = {OPMODE_I2C,              // Modalità Operativa
                          400000,                  // Velocità di Clock
                          FAST_DUTY_CYCLE_2        // Duty Cycle
                          };
```

6.1. OLED DISPLAY

Un esempio di dispositivo che comunica con l'I2C è un Display OLED. Al Campus in particolare ci è stato fornito un Display OLED SSD1306:

Figura 11 – Foto dell'OLED

Successivamente alla struttura di configurazione dell'I2C, implementiamo la struttura di configurazione dell'OLED Display e la definizione del proprio Driver:

```
// STRUTTURA DI CONFIGURAZIONE OLED DISPLAY:
static const SSD1306Config ssd1306cfg = {
    &I2CD1,                                    //I2C Driver
    &i2ccfg,                                   //I2C Configuration structure
    SSD1306_SAD_0X78,                          //I2C Device address
    };

// DEFINIZIONE DRIVER OLED DISPLAY:
static SSD1306Driver SSD1306D1;
```

Configurazione dei PIN relativi all'I2C:

```
palSetLineMode(LINE_ARD_D15, PAL_MODE_ALTERNATE(4) | PAL_STM32_OTYPE_OPENDRAIN |
               PAL_STM32_OSPEED_HIGHEST | PAL_STM32_PUPDR_PULLUP);
palSetLineMode(LINE_ARD_D14, PAL_MODE_ALTERNATE(4) | PAL_STM32_OTYPE_OPENDRAIN |
               PAL_STM32_OSPEED_HIGHEST | PAL_STM32_PUPDR_PULLUP);
```

Per utilizzarlo si sfruttano dei thread ed una serie di funzioni messe a disposizione dalla libreria ssd1306, che permettono di impostare la dimensione e colori del font, figure geometriche, ecc…

7. PWM

PWM (Pulse Width Modulation) è una modulazione digitale in cui la forma d'onda cambia tra due stati con tempo di salita trascurabile e un periodo costante. In particolare, viene utilizzato per la gestione in hardware di periferiche esterne grazie a dei timer (TIM) in grado di interfacciarsi ai sensori.

Importante è il concetto di **Duty Cycle**, ovvero, il rapporto tra la durata dell'impulso positivo e l'intero periodo preso in considerazione.

Per la configurazione del driver PWM:

```
// CONFIGURAZIONE DRIVER PWM:
static PWMConfig pwmcfg = {
    10000,                                          // PRESCALER CLOCK (frequenza del TIMER in Hz) --> TIM
    200,                                            // CONTATORE (periodo PWM in tick, 1 tick = 10 ms)
    NULL,
    {
      // ARRAY DI CANALI PWM con un proprio ouput, PARAMETRO MODE definisce qual è il livello attivo (HIGH o LOW)
      {PWM_OUTPUT_ACTIVE_HIGH, NULL},
      {PWM_OUTPUT_DISABLED, NULL},
      {PWM_OUTPUT_ACTIVE_HIGH, NULL},
      {PWM_OUTPUT_ACTIVE_HIGH, NULL}
    },
    0,
    0
};
```

7.1. BUZZER

Un dispositivo che usa il PWM è il **Buzzer**, un dispositivo di segnalazione audio, che può essere meccanico o elettromeccanico. I tipici utilizzi del buzzer includono dispositivi di allarme, timer, ecc…

Figura 12 – Foto del BUZZER

La PWM gestisce in hardware il buzzer, essa tramite timer interni è in grado di prendere in ingresso un certo periodo dato dall'utente e tramite impulsi far suonare o meno il buzzer.

La PWM viene utilizzata perché in grado di cambiare il periodo in runtime, quindi, utile in caso di variazione del periodo durante l'esecuzione del programma (utile anche per i LED in modalità DYNAMIC).

Per la configurazione del Buzzer:

```
// CONFIGURAZIONE PWM DEL BUZZER:
static PWMConfig buzzpwmcfg = {
    10000,
    10000,
    NULL,
    {
      {PWM_OUTPUT_ACTIVE_HIGH, NULL},
      {PWM_OUTPUT_DISABLED, NULL},
      {PWM_OUTPUT_DISABLED, NULL},
      {PWM_OUTPUT_DISABLED, NULL}
    },
    0,
    0
};
```

8. ENCODER

L'**Encoder** è un apparato elettronico che *converte la posizione angolare del suo asse rotante in un segnale elettrico digitale* collegato da opportuni circuiti elettronici e con appropriate connessioni meccaniche. È in grado di misurare spostamenti angolari, movimenti rettilinei e circolari nonché velocità di rotazione e accelerazioni.

Figura 13 – Foto dell'ENCODER

L’Encoder può essere utilizzato per vari scopi. Nel nostro progetto ha avuto il ruolo di **Dimmer**, un regolatore elettronico utilizzato per controllare la potenza assorbita da un carico. È quindi in grado di controllare l’intensità luminosa del LED.

9. IL PROGETTO

Come anticipato nella parte iniziale di questa relazione, dopo una prima breve fase preliminare, in cui i sono state fornite tutte le nozioni più importanti, i tutor della ST ci hanno diviso in gruppi per poter lavorare su un progetto.

Erano 4 i possibili progetti da scegliere, la nostra scelta è ricaduta sull’implementazione di una test suite: un’interfaccia che riceva in ingresso (mediante la shell) ed elabora una serie di comandi.

Nel documento di specifica del progetto che ci è stato assegnato, erano indicati i seguenti requisiti funzionali:

- **TS-REQ01-1**: LED [RED|GREEN|BLUE] [STATIC|DYNAMIC]
  Se STATIC → [ON|OFF] altrimenti DYNAMIC → [100÷1000] (ms).
- **TS-REQ01-2**: JOY [XY|POLAR]
  Una volta attivato vengono visualizzati:
  - se in XY: le coordinate cartesiane relative alla posizione del joypad.
  - se in POLAR: le coordinate polari relative alla posizione del joypad.
- **TS-REQ01-3**: OLED [LED|JOY]
  Una volta attivato, lo stato del LED o del JOYPAD viene stampato sul display OLED.
- **TS-REQ01-4**: BUZZ [ON|OFF] [PERIOD]
  Viene riprodotto un segnale acustico, con periodo espresso in multipli interi del secondo.
- **TS-REQ01-5**: DIMMER [ON|OFF] [RED|GREEN|BLUE]
  Una volta attivato, il led indicato deve avere l'intensità controllata dall'encoder.
- **TS-REQ02-1**: DEMO
  Implementa una ruota dei colori con display OLED e il BUZZER. Il buzzer emette un tono quando viene raggiunto uno dei tre colori primari e la percentuale dei singoli color dominanti (contemporaneamente) viene visualizzato sul display.

9.1. BOARD E COLLEGAMENTI

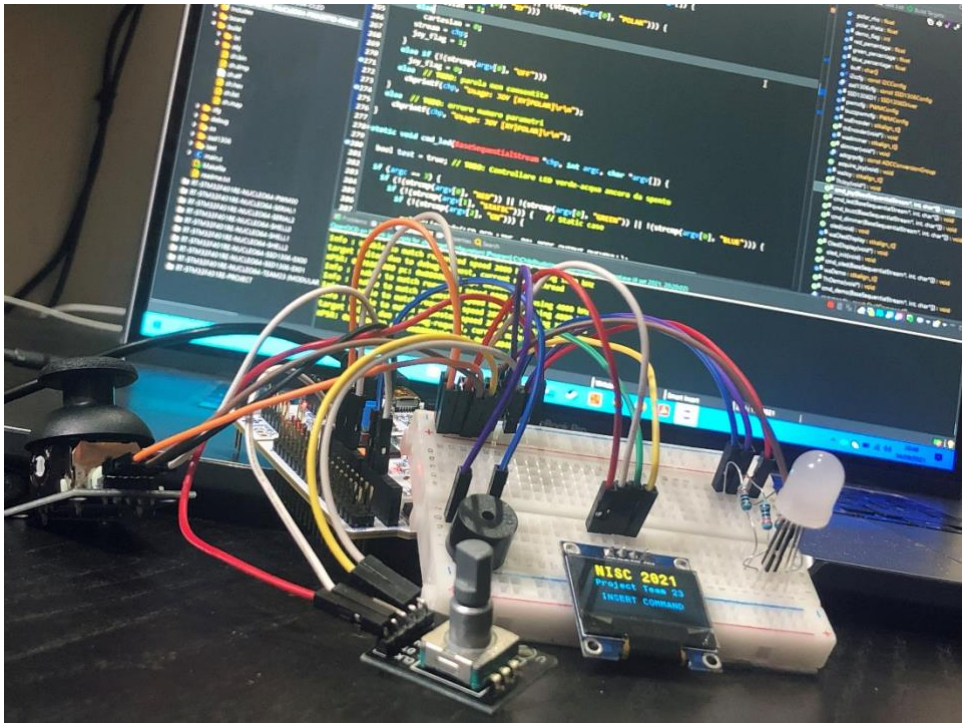


Figura 13 – Foto della scheda con tutte le periferiche collegate

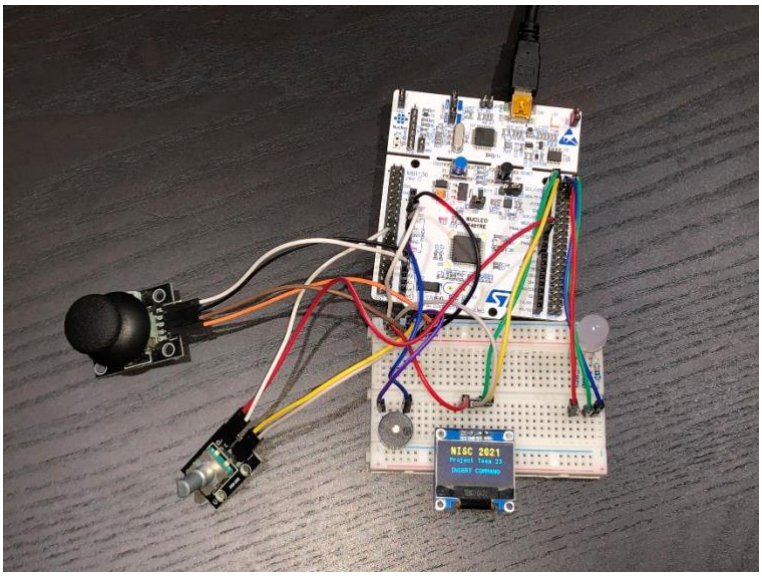


Figura 14 – Foto della scheda dall'alto