Concetti introduttivi alla programmazione della board STM32F3-Discovery

Luigi Coppolino, Giovanni Mazzeo

Outline

- I Sistemi Embedded, cosa sono e a cosa servono
- I microcontrollori, la loro architettura
- Il processore ARM Cortex M4
- La board STM32F3-Discovery
- Cosa significa programmare un microcontrollore
- Tool necessari per lo sviluppo di un progetto
- Un'analisi più dettagliata di un programma per il microcontrollore

I Sistemi Embedded

- Un Sistema Embedded (SE) (o Sistema Dedicato) è un sistema di elaborazione progettato per eseguire un insieme ristretto di funzioni per applicazioni specifiche (industriali, aerospaziali, automotive, ecc.)

- Solitamente, come nel caso di un SE per il controllo treni, sono sistemi che operano in tempo reale, ovvero devono rispondere ad eventi esterni in tempi prestabiliti (*deadline*). Si parla in questo caso di Sistemi Real-Time.

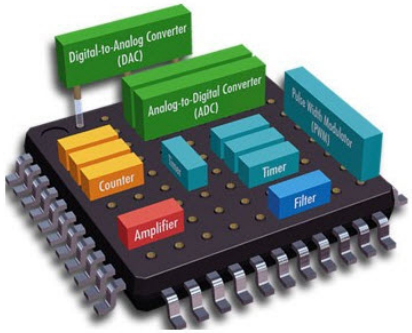
- Esempi di SE sono nel mondo che ci circonda ogni giorno: nelle lavatrici, nelle auto, nella macchina del caffè, nei cellulari.

Cos'è un Microcontrollore

- I Sistemi Embedded sono basati sui Microcontrollori. I microcontrollori sono semplicemente "computer di dimensioni ridotte" all'interno di un singolo circuito integrato. Sono utilizzati per applicazioni specifiche (*Special Purpose)*.
- Un microcontrollore, così come un computer *General Purpose*, ha una memoria, può essere programmato per qualsiasi computazione, riceve input e genera output.
- Nella maggior parte dei casi i microcontrollori possono essere dei *System-on-Chip*. Ovvero incorporano all'interno di un singolo chip tutte le unità tipiche di un calcolatore: CPU, memoria, bus, interfacce I/O, periferiche.

Embedded Systems vs General Purpose Computing

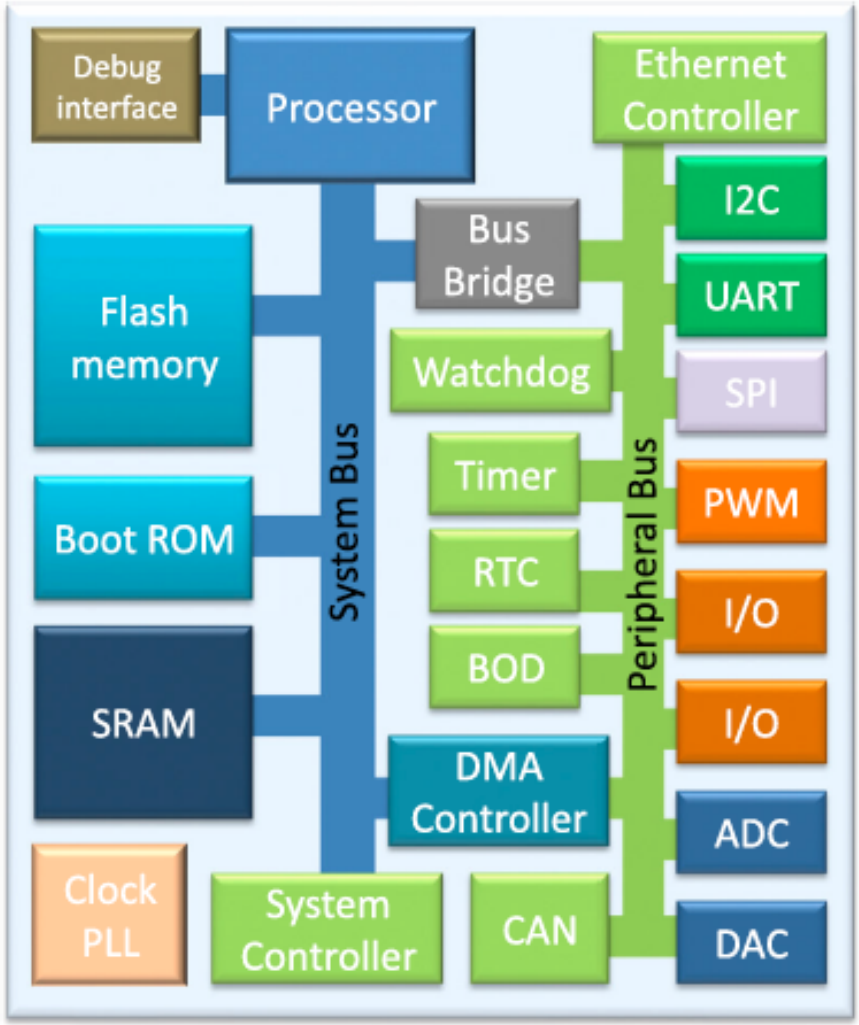
- *Embedded Systems (Special Purpose)*
  - Eseguono singole applicazioni già note in fase di sviluppo del sistema
  - Spesso hanno vincoli sul tempo di esecuzione. Non per forza, le performance devono essere alte
  - In molte applicazioni hanno *hard-contraints* sul consumo di potenza



- *General Purpose*
  - Eseguono qualsiasi tipo di applicazioni
  - Faster is always better
  - Possono essere sempre riprogrammati da un utente finale

Architettura Generale di un Microcontrollore

- Ogni microcontrollore integra:
  - Il Processore (e.g. Intel, Arm)
  - Una memoria (SRAM o DRAM)
  - Una memoria flash
  - Bus di comunicazione (e.g. Advanced Microcontroller Bus Architecture (AMBA)). Quasi sempre due bus a diverse frequenze di clock.
  - Interfacce di Comunicazione (I2C, SPI, UART, etc.)
  - ADC/DAC
  - Clock



Il Processore ARM Cortex M4

- Il processore ARM Cortex M4 e più in generale la famiglia M-series è un processore utilizzato per microcontrollori che garantisce bassi consumi di potenza con buone prestazioni

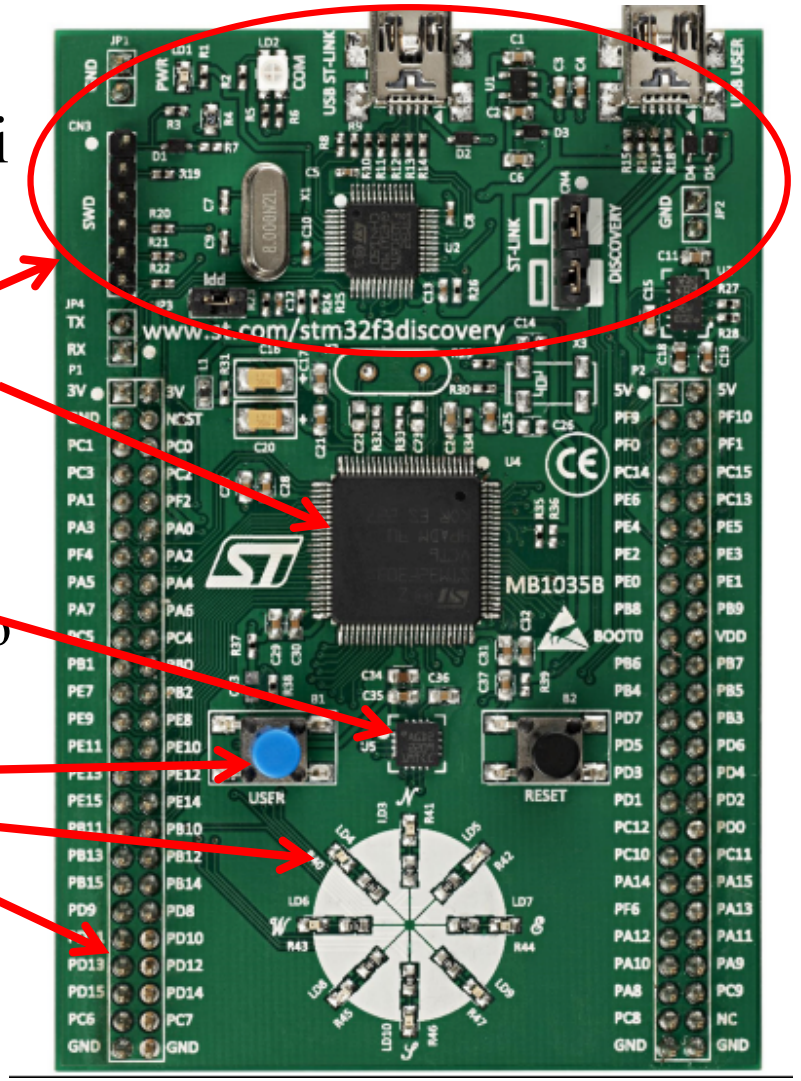
- E' un processore che implementa il set di istruzioni (ISA) Thumb a 16 bit che può essere visto come una forma compressa di un sottoinsieme dell'ARM Instruction Set (a 32 bit).

- Il processore presenta una pipeline a 3 stadi

- Gestisce le interruzioni in maniera innestata con meccanismi quale il Wake Up Interrupt Controller (WUIC) che permettono di ridurre il consumo di potenza

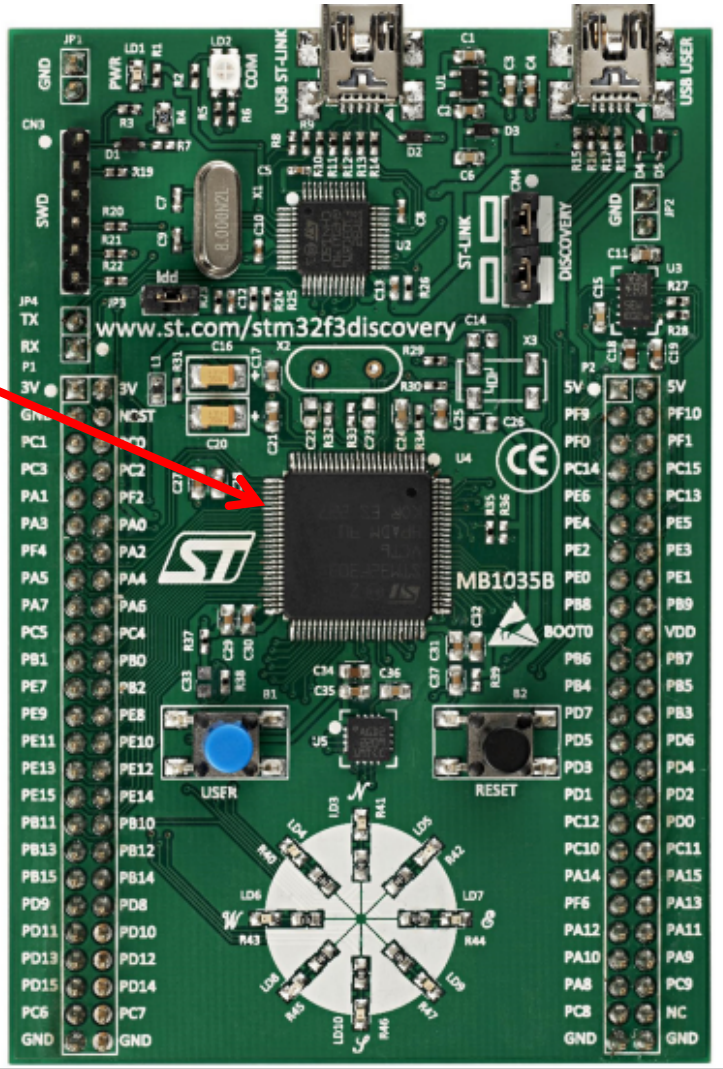
La board di sviluppo STM32F3-Discovery

- La STM32F4-Discovery è una scheda di svilluppo che contiene al suo interno:
  - Il microcontrollore STM32F4 basato sul ARM Cortex M4
  - Accelerometro 3-axis
  - ST-Link Debugger
  - Pin di interfacciamento con il mondo esterno (GPIO)
  - Due pulsanti e 8 LED



Il Microcontrollore STM32F3

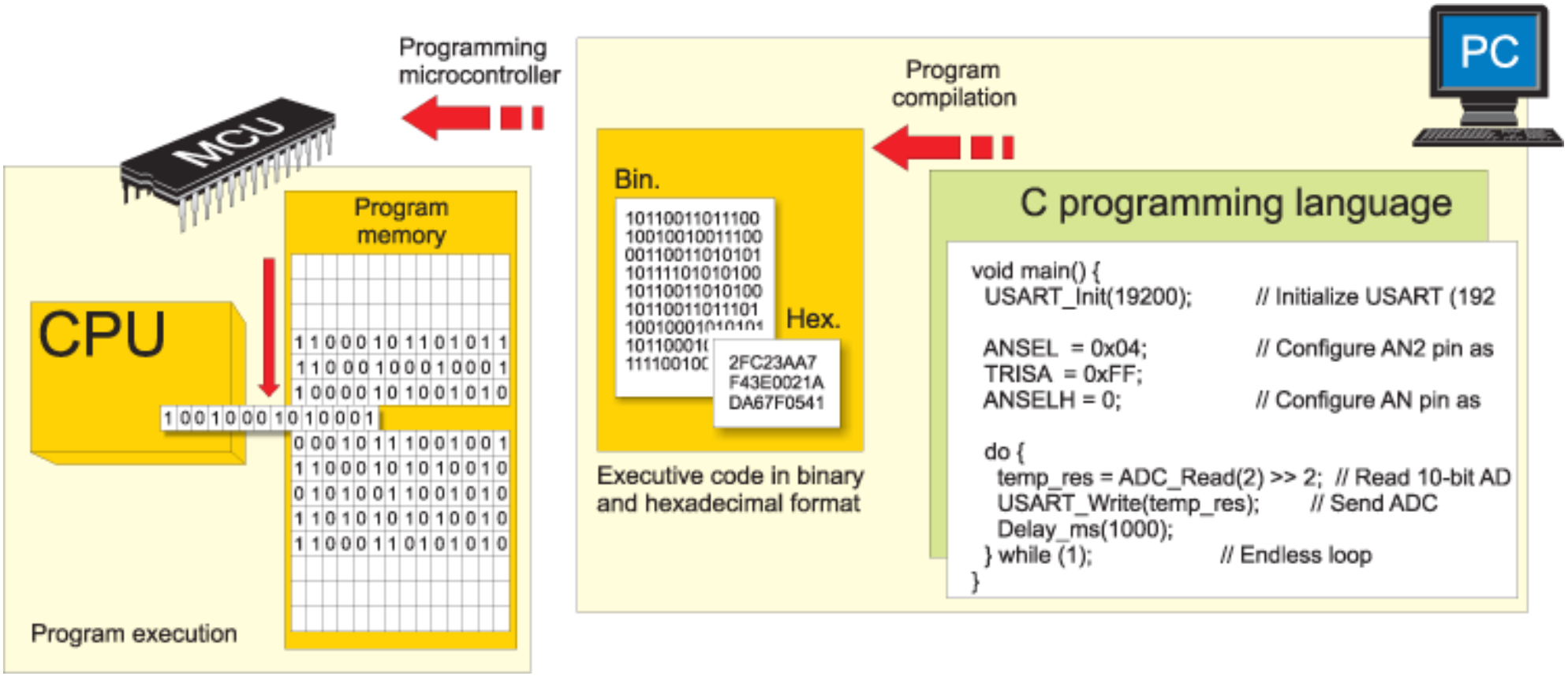
- Il microcontrollore STM32F3 presenta:
  - ARM M4 core processor 168MHz
  - 1MB Flash
  - 192KB SRAM
  - > 80 I/O Pins
  - 13 Timers
  - Serial Communications: 6 UARTs, 3 SPI, 2 I2C
  - USB OTG Controller
  - External memory controller
  - Internal DMA System
  - Ethernet Controller
  - SD Controller



Concetti di Base per la Programmazione dei Microcontrollori

- Programmare un microcontrollore significa istruirlo a fare una specifica funzione di interesse
- Per fare ciò noi scriveremo **programmi in linguaggio C** che saranno poi **compilati** (quindi tradotti in linguaggio macchina) per il nostro microcontrollore
- Il programma "tradotto" in linguaggio macchina sarà poi caricato nella **memoria Flash** del microcontrollore. Questa ha la caratteristica di mantenere la programmazione anche quando l'alimentazione al microcontrollore viene spenta
- Durante l'esecuzione la **SRAM** conterrà i dati, ovvero tutte le variabili che si utilizzano all'interno del programma

Concetti di Base per la Programmazione dei Microcontrollori



Programmare Periferiche dei Microcontrollori – 1/2

- Il codice che andremo a scrivere servirà a programmare una o più **periferiche,** che eventualmente dovranno comunicare tra loro, per portare a termine la funzione di interesse

Ad es: *si vuole istruire il microcontrollore a leggere ogni ora i valori misurati da un sensore di pressione*

- Programmare la periferica *Analog-to-Digital Converter (ADC)* affinché legga i dati misurati dal sensore
- Programmare la periferica *Timer* affinché ogni ora sia effettuata la lettura

Programmare il Timer

- Esempio semplificato di un timer TIM
- Vogliamo programmare un timer affinché conti in modo decrescente dal valore 100 al valore 0.
- Per fare ciò, dunque, dovremo:
  - Scrivere nel registro di controllo la tipologia di conteggio che si desidera (decrescente), se si vuole che il conteggio ricominci una volta terminato, ecc.
  - Scrivere nel counter register (registro di dato) il valore di partenza (100)
  - Avviare il timer scrivendo nel registro di controllo
  - Periodicamente leggere il registro di stato per controllare se il timer è arrivato al valore 0

Programmare Periferiche dei Microcontrollori – 2/2

- Programmare una periferica significa scrivere in specifici registri di memoria (ad indirizzi prestabiliti)
- Ogni periferica/unità ha il suo **set di registri dedicati** – mappati su specifici indirizzi di memoria – dal quale andrà a *leggere* per sapere come si deve comportare o andrà a *scrivere* per indicare il suo *stato*
- Ogni periferica presenta:
  - **Registri di Stato** – Un insieme di *flag* da poter leggere per conoscere lo stato della periferica
  - **Registri di Controllo** – Registri da dover scrivere per impostare il funzionamento desiderato della periferica
  - **Registri di Dato** – Un registro dal quale potremmo o leggere o scrivere dati di interesse

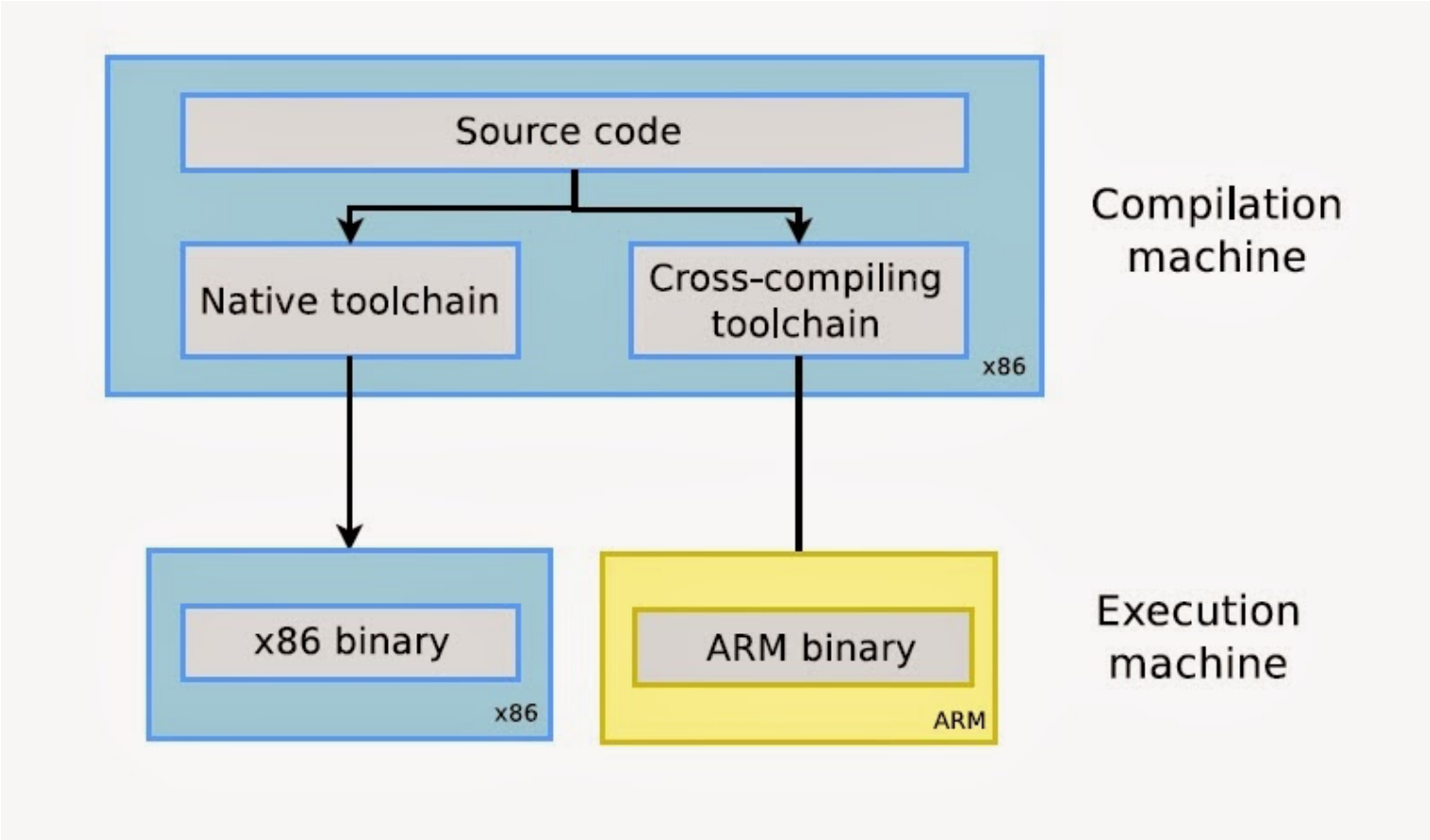
Sporchiamoci le mani: la Prima Programmazione della STM32F3

- **Obiettivo:** Si vuole programmare la scheda affinché accenda i led in modo rotatorio
- **Cosa necessitiamo:**
  - La STM32F3 discovery board
  - ARM-GCC Toolchain
  - Debugger
  - ST-LINK Drivers
  - ChibiStudio

ARM-GCC Cross Toolchain 1/2

- Il programma che scriveremo per controllare il led della scheda è scritto nel linguaggio di programmazione C
- Tale programma dovrà essere compilato per generare il codice macchina che il microcontrollore saprà eseguire
- Il compilatore nativo (e.g. gcc) genererebbe il codice macchina per l'architettura su cui si sta eseguendo la compilazione
- Noi dobbiamo compilare il codice per l'architettura specifica del microcontrollore
- A tal fine necessitiamo un Cross-Compiler il quale permette di generare un file binario eseguibile su di un'architettura diversa da quella della macchina su cui è stato lanciato il cross compiler

ARM-GCC Cross Toolchain 2/2



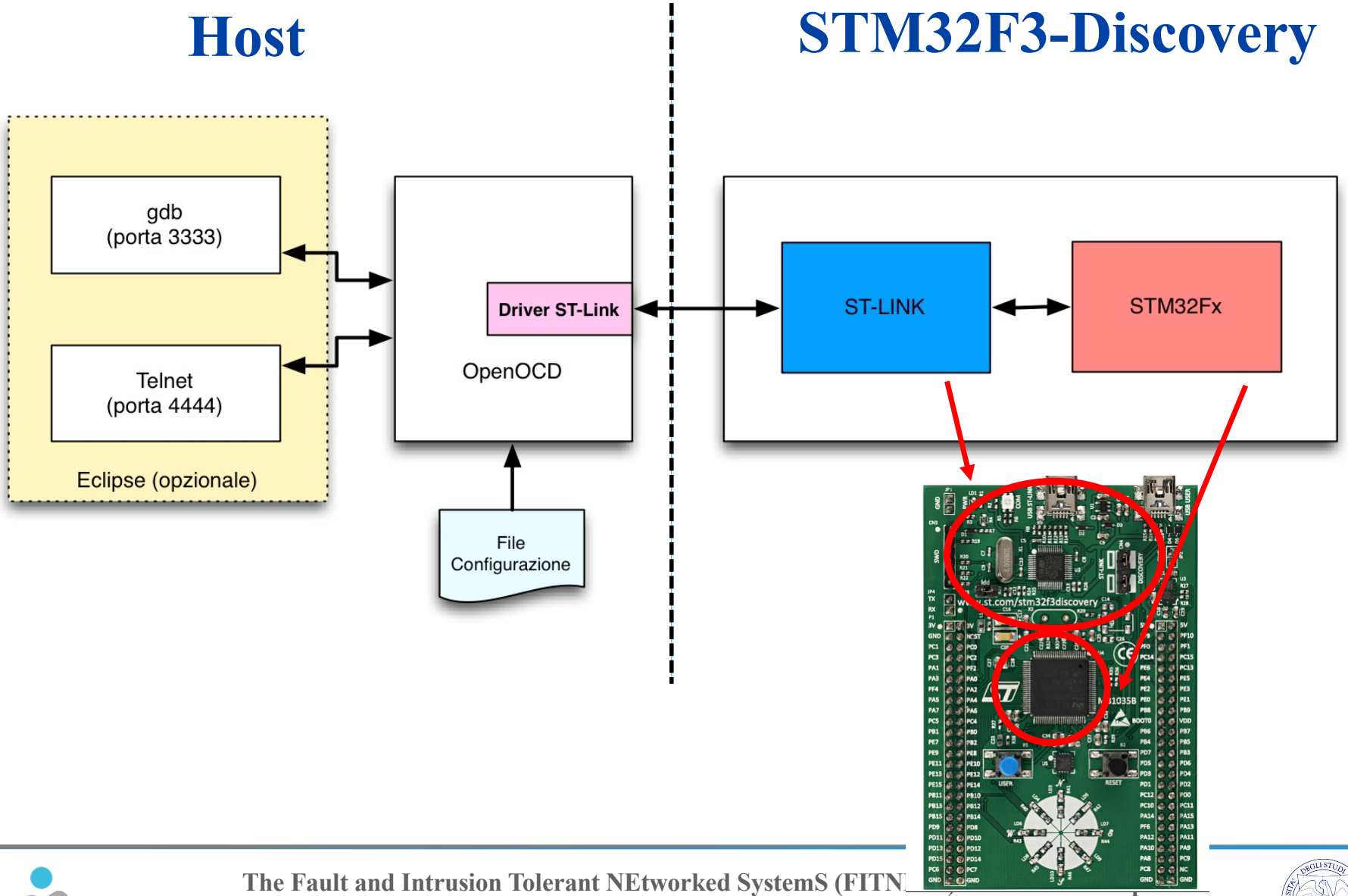
Debugger – 1/3

- Il debugger è fondamentale nello sviluppo di software complessi.
- Permette di scorrere l'esecuzione del codice "step-by-step" e di legger i valori intermedi di tutte le variabili utilizzate nel codice
- Il più famoso debugger è il GNU GDB utilizzato per verificare programmi C/C++ (https://www.gnu.org/software/gdb/)
- Si possono realizzare due tipologie di attività debugging:
  - Local - Il debugging di un programma che esegue in locale sullo stesso sistema in cui si effettua il debug
  - Remote – Il debugging di un programma che esegue su un sistema (detto *target*) differente da quello (detto *host*) su cui si esegue il debug

Debugger – 2/3

- Nel caso della STM32F3 ovviamente realizzeremo un *remote* debugging. Faremo uso di OpenOCD.
- Questo si basa su una comunicazione client/server realizzata tra la scheda e il nostro calcolatore
  - Un server OpenOCD, avente un file di configurazione specifico per la scheda su cui si vuole eseguire il programma, sarà lanciato sulla scheda e fornirà dunque informazioni al client sui valori nella memoria del microcontrollore.
  - Un client sarà lanciato e comunicherà con il server per ottenere informazioni da fornire allo sviluppatore
- OpenOCD è un debbuger universale On-Chip.
- Attraverso un driver interno interagisce con le board che utilizzano il protocollo ST-Link

Debugger – 3/3


Host
STM32F3-Discovery
gdb
(porta 3333)
Telnet
(porta 4444)
Eclipse (opzionale)
Driver ST-Link
OpenOCD
File
Configurazione
ST-LINK
STM32Fx

Driver ST-Link

- E' necessario installare i **driver** della scheda STM32F3-Discovery per permettere la corretta comunicazione della scheda con la nostra macchina host
- A tal fine, scaricare ed installare ST-Link:
  - http://www.st.com/en/development-tools/stsw-link004.html#getsoftware-scroll
- Tale eseguibile installerà oltre ai driver, un tool **ST-Link-Utility**
- Questo è utilizzato per caricare il file binario in formato .hex (senza effettuare debug), compilato con il cross compiler, all'interno della scheda

ChibiStudio – 1/2

- **ChibiStudio** è l'ambiente di sviluppo che utilizzeremo per programmare la scheda STM32F3-Discovery
- Tale ambiente **integra già al suo interno la ARM-GCC toolchain ed il debugger OpenOCD**
- ChibiStudio è basato su **Eclipse**: questo è un Integrated Development Environment (IDE) ovvero un ambiente di programmazione open source.
- Eclipse è utilizzato per programmare in diversi linguaggi di programmazione (C, C++,Java, Rust, PHP, JavaScript, …)
- Scaricare ChibiStudio da questo link:
  - https://sourceforge.net/projects/chibios/files/ChibiStudio/ChibiStudio_Preview19.7z/download

ChibiStudio – 2/2

- Scompattare l'ambiente di sviluppo in: "C:\ChibiStudio"
- L'ambiente di sviluppo necessita della **Java Runtime Environment (JRE)**
- Scaricare da questo sito l'eseguibile JRE di interesse
  - http://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html
- Eseguire dunque jre-8u121-windows-x64.exe e procedere nell'installazione della JRE
- Sarà possibile ora lanciare Eclipse dalla cartella "C:\ChibiStudio"

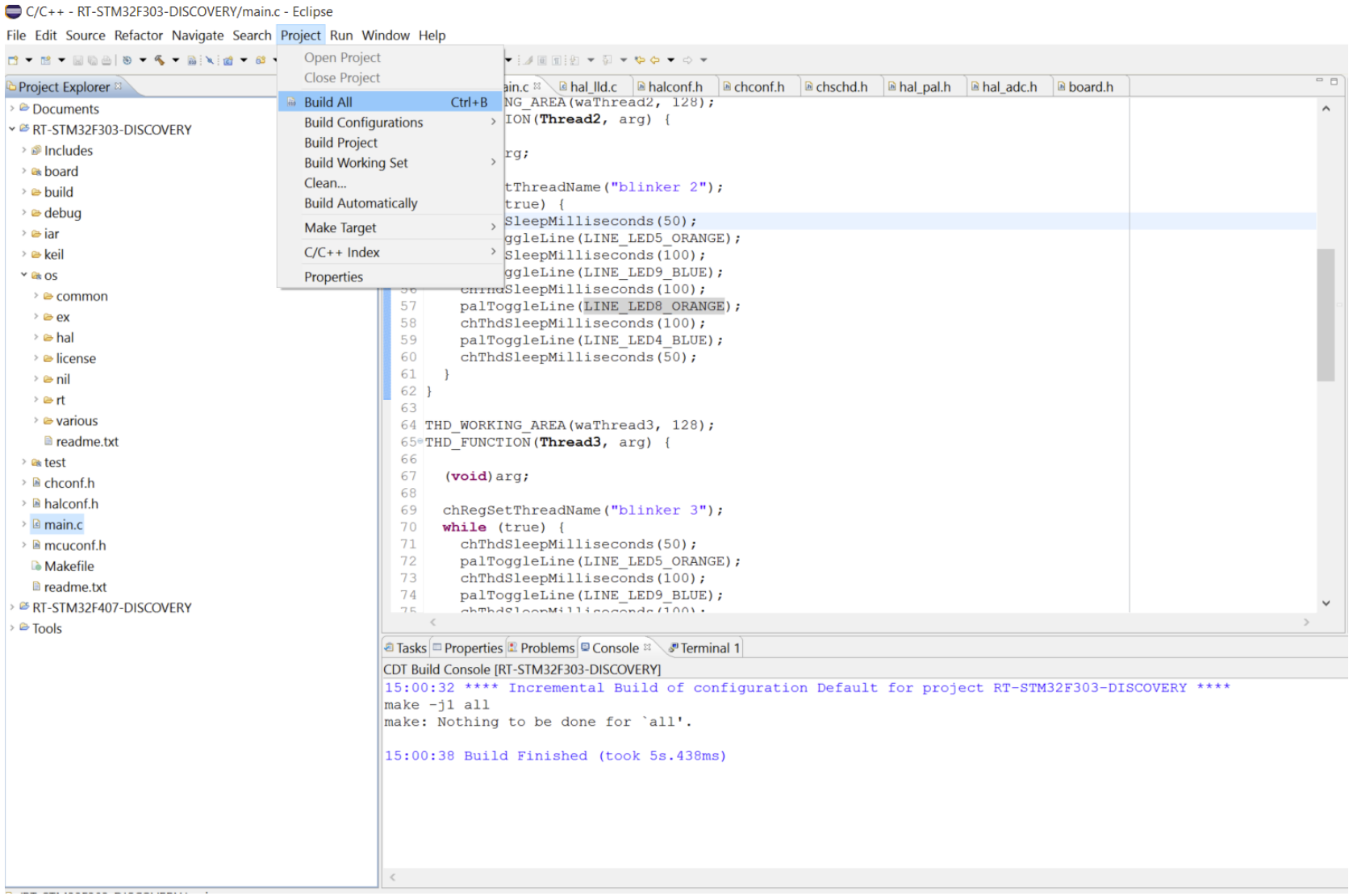
Manuali Scheda STM32F3-Discovery

- Nel corso di queste lezioni faremo uso di due manuali utili ad approfondire le diverse periferiche:
  - **Reference Manual**
    - http://www.st.com/content/ccc/resource/technical/document/reference_manual/4a/19/6e/18/9d/92/43/32/DM00043574.pdf/files/DM00043574.pdf/jcr:content/translations/en.DM00043574.pdf
  - **User Manual**
    - http://www.st.com/content/ccc/resource/technical/document/user_manual/8a/56/97/63/8d/56/41/73/DM00063382.pdf/files/DM00063382.pdf/jcr:content/translations/en.DM00063382.pdf

Creazione di un Primo Progetto: Blink Led

- Dobbiamo importare nel workspace di Eclipse un progetto di esempio che utilizzeremo per iniziare a programmare
- Aprire Eclipse. Andare in File->Import->Existing Projects into Workspace
- Browse ed aprire il percorso:
  - "C:\ChibiStudio\chibios176\demos\STM32\RT-STM32F303-DISCOVERY"

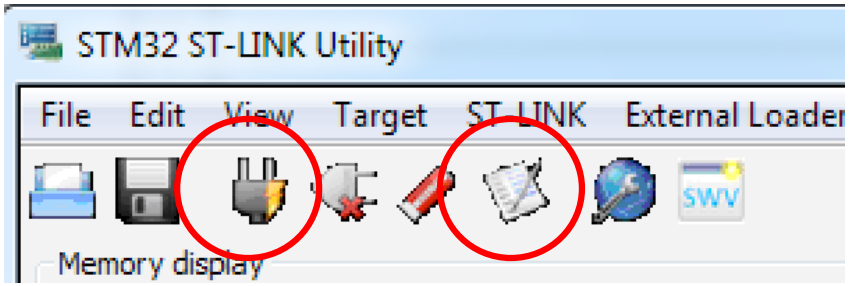
Compilazione del Programma



Caricamento del programma: ST-Link Utility

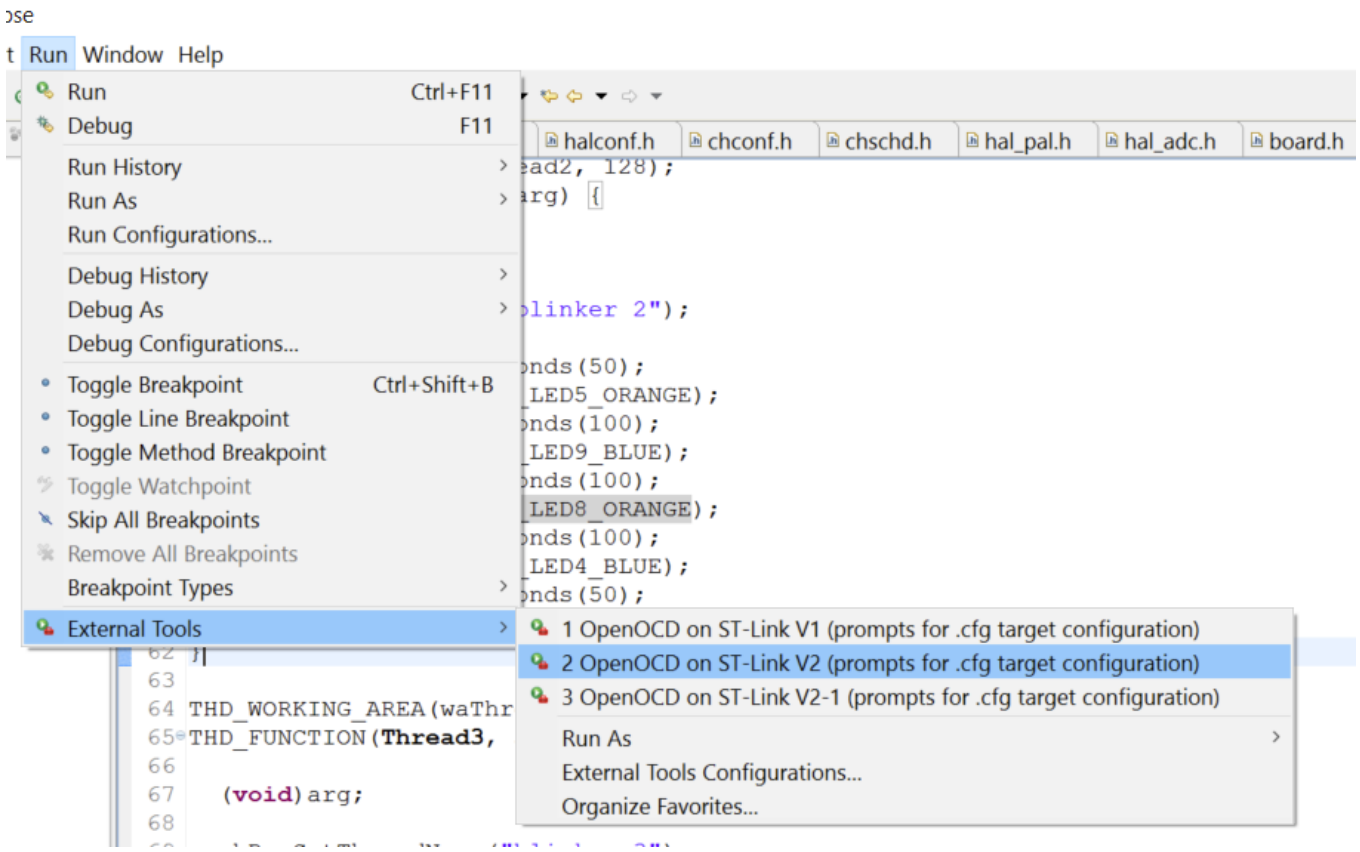
- Il metodo più semplice per “flashare” la nostra scheda è attraverso l’uso del ST-Link Utility
- Collegare la scheda, avviare **STLink Utility**, e connetterlo alla scheda cliccando sulla icona di presa elettrica.
- Aprire (in file->open) il file da caricare sulla scheda. Tale file lo si può trovare nella cartella del progetto di Eclipse sotto “**debug**”. Tale file ha estensione **.hex**
- Una volta aperto, lanciare il “program verify” tramite l’icona evidenziata.
- Scollegare e ricollegare la scheda per vedere se il led lampeggia come previsto

```
Invoking: Cross ARM GNU Create Flash Image
arm-none-eabi-objcopy -O ihex "BlinkLed.elf"  "BlinkLed.hex"
Finished building: BlinkLed.hex

Invoking: Cross ARM GNU Print Size
arm-none-eabi-size --format=berkeley "BlinkLed.elf"
   text    data     bss     dec     hex filename
   8545     160     420    9125    23a5 BlinkLed.elf
Finished building: BlinkLed.siz

14:37:26 Build Finished (took 13s.369ms)
```



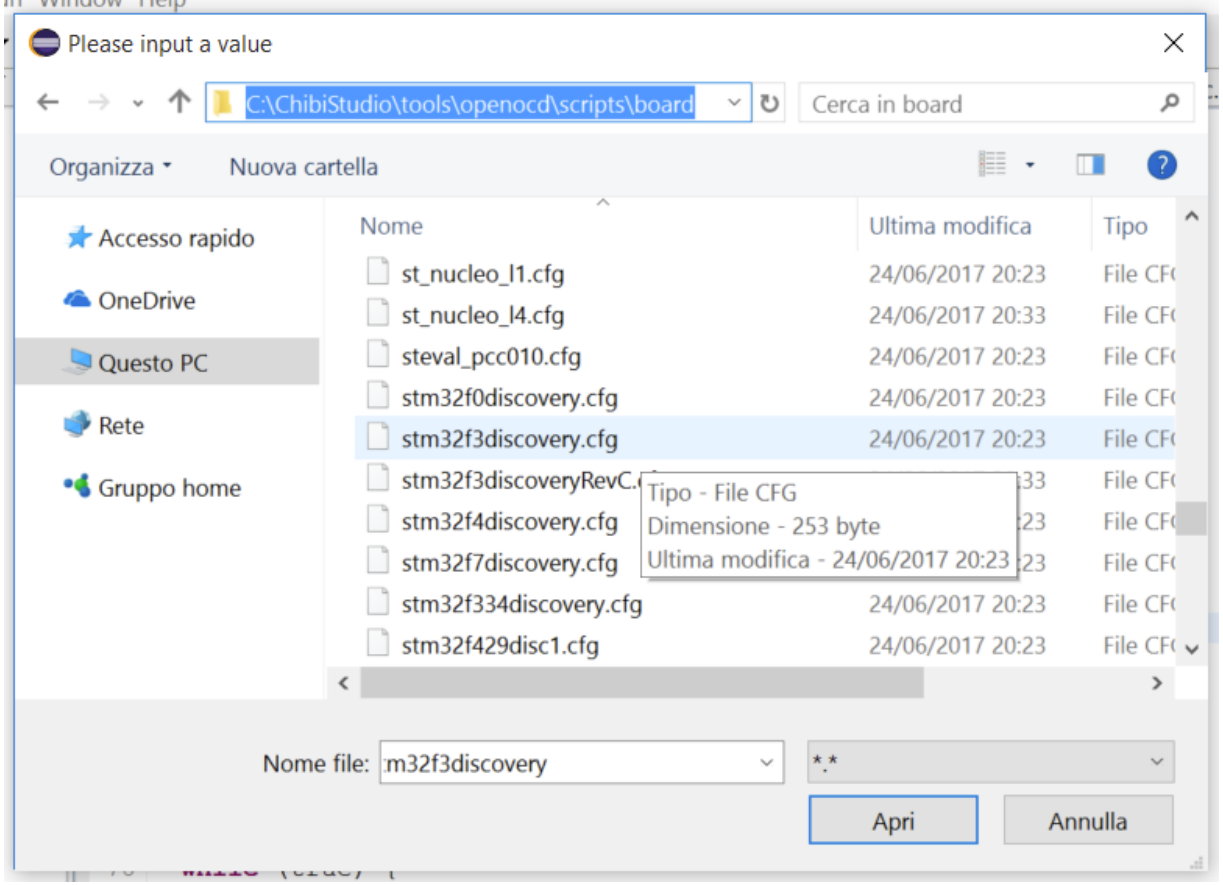
Debug del Programma

- Per effettuare il debug è necessario prima lanciare il server OpenOCD al quale passare il file di configurazione della nostra scheda



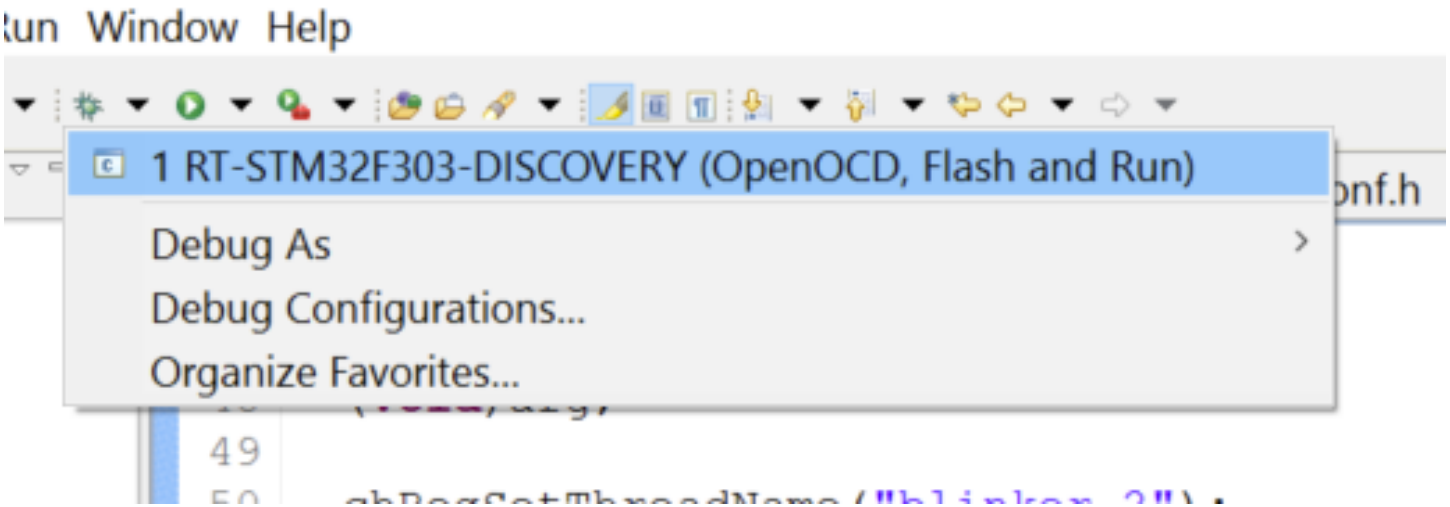
Debug del Programma

- Per effettuare il debug è necessario prima lanciare il server OpenOCD al quale passare il file di configurazione della nostra scheda



Debug del Programma

- Una volta lanciato il server, potremo lanciare il client gdb



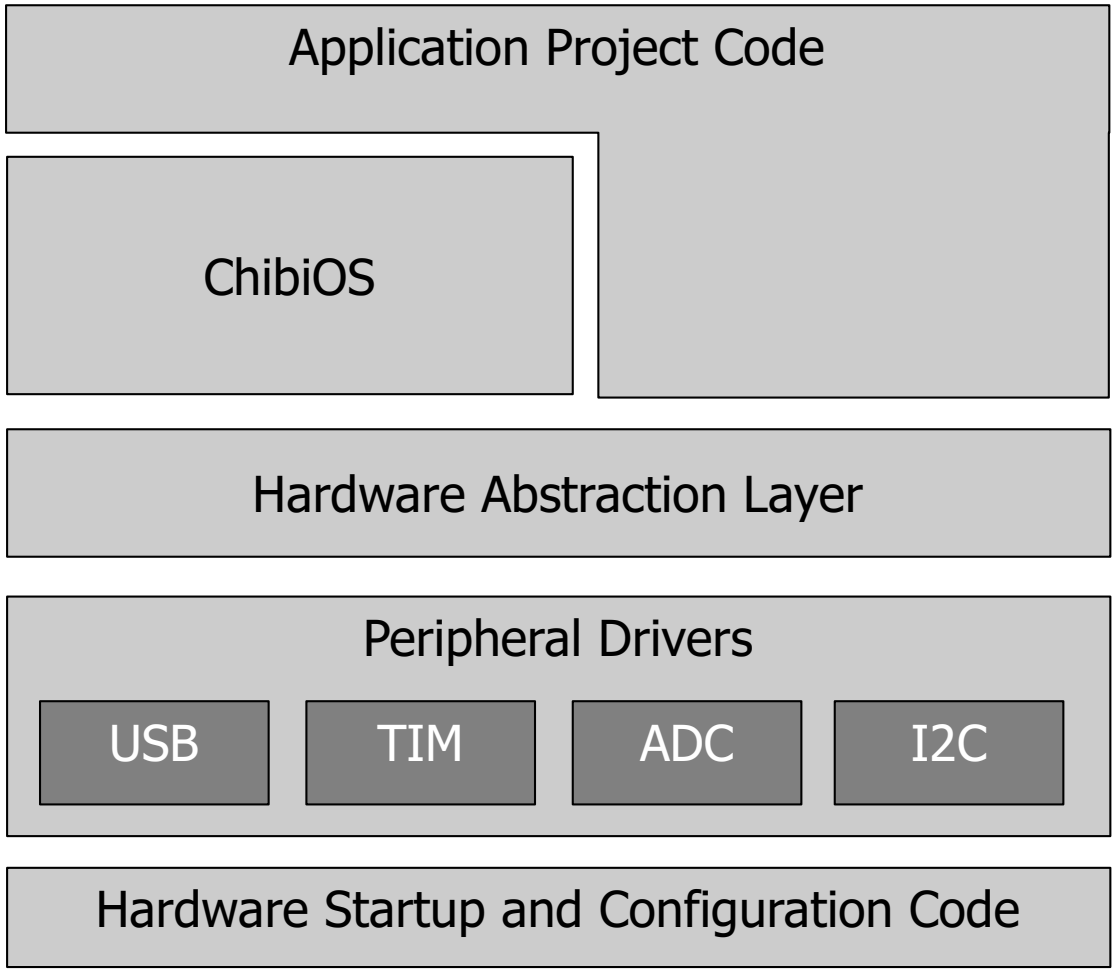
Un'occhiata più da vicino al progetto creato

- Per facilitare la vita del programmatore, il progetto appena creato con ChibiStudio porta già al suo interno un insieme di librerie che permettono di *astrarre* la programmazione delle periferiche

- Sarebbe molto più complicato dover andare a scrivere direttamente nei registri delle periferiche delle sequenze di bit

- Come accade sempre nell'informatica, il microcontrollore può essere programmato in **livelli di astrazione differenti**

Organizzazione su livelli di un progetto



Hardware Abstraction Layer

Questo livello permette di astrarre le periferiche hardware fornendo ai livelli superiori una visione delle periferiche che non dipende dal sistema su cui si sta eseguendo.

ChibiOS

- ChibiOS è un **Real-Time Operating System (RTOS)** open source per applicazioni embedded
- Offre supporto per la multi programmazione (semafori, mutex, messaggi, mailboxes) e dunque per la creazione e gestione di **thread**
- Permette l'utilizzo del File System (FS) FatFS
- Supporta lo stack TCP/IP
- Offre l'interfaccia per le periferiche montate sulla STM32F4 quali: ADC, CAN, DAC, EXT, I2C, ICU, PWM, SPI, UART, USB ed altre ancora.

I Sistemi Operativi

- Un **Sistema Operativo** è un programma che esegue su un computer interfacciandosi con l'Hardware per fornire servizi ed interfacce ad applicazioni che eseguono su quel computer.
- Una funzionalità molto importante che i sistemi operativi forniscono è il **Multi-Tasking**. Ovvero la possibilità di eseguire più programmi allo stesso tempo.
- Volendo essere più corretti, il SO schedula su un ciascun processor core un singolo thread di esecuzione dando l'impressione all'utente che fa uso del computer che i programmi siano eseguiti contemporaneamente.

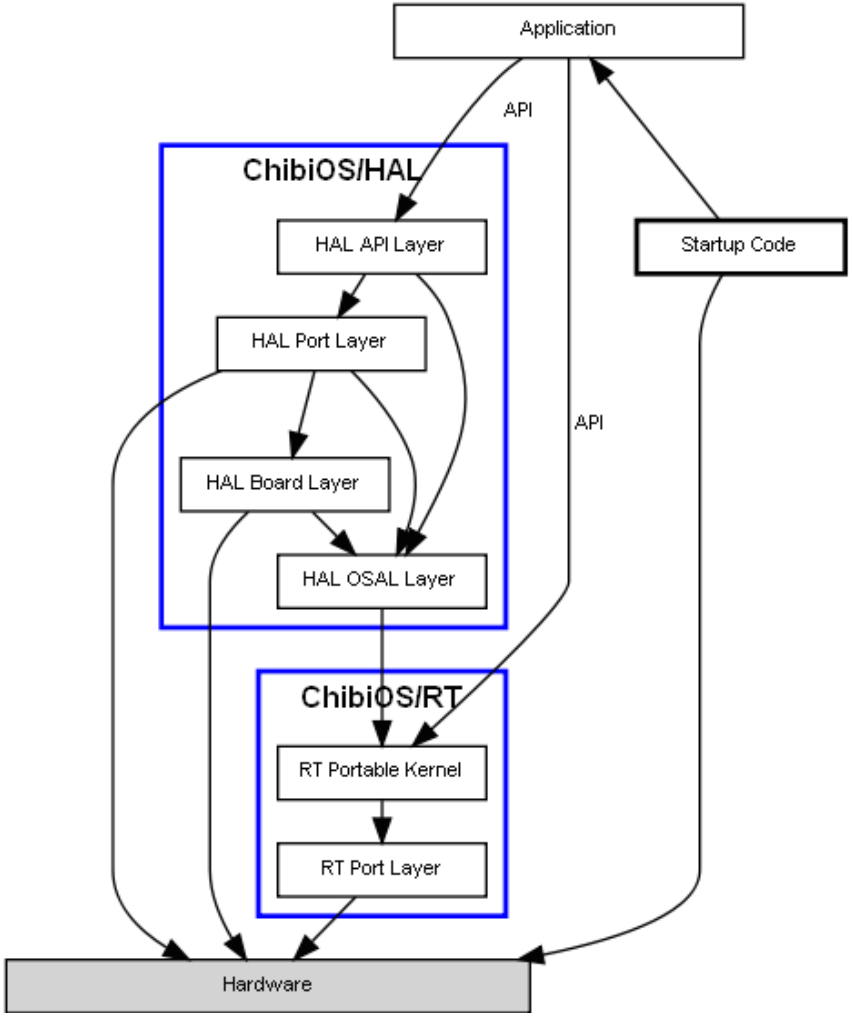
RTOS

- Un Real-Time Operating System (RTOS) è un sistema operativo utilizzato per applicazioni embedded (Sistemi Dedicati) e quindi solitamente per SoC.
- Le applicazioni embedded spesso hanno requisiti Real-Time. Ciò significa che il sistema deve rispondere rapidamente ad eventi che si verificano in tempo reale.
- Per tale motivo gli RTOS devono essere:
  - **Predicibili,** nel senso che le unità del SO (scheduler, interrupts, etc.) devono essere predicibili
  - **Deterministici,** nel senso che devono essere capaci di restituire uno stesso risultato sotto le stesse condizioni

Perché un RTOS

- Utilizzare un RTOS comporta vantaggi quali:
  - **Massimizzazione dell'uso delle risorse (e.g. multi-tasking): maggiore efficienza**
  - Semplificazione per la programmazione
  - Allocazione della memoria gestita in modo migliore
  - Maggiore protezione da errori "gravi" di programmazione

Architettura di ChibiOS



Architettura di ChibiOS

- **Applicazioni**: il codice che voi svilupperete ed andrete ad eseguire sul SO
- **ChibiOS/RT:** è il Real-Time scheduler diviso in due livelli:
  - **RT Portable Kernel** - La parte del RTOS kernel indipendente dall'architettura
  - **RT Port Layer** – E' la parte di RTOS kernel specifica per ciascuna architettura
- **HAL** – L' Hardware Abstraction Layer (HAL) offre un insieme di device drivers per le periferiche