Sistemi embedded: teoria e pratica

Alexjan Carraturo e Andrea Trentini

A. Carraturo, A. Trentini, Sistemi Embedded: Teoria e Pratica

Ledizioni

ISBN:

Tutti i marchi ed i loghi appartengono ai legittimi proprietari: marchi di terzi, nomi di prodotti, nomi commerciali, nomi corporativi e società citati possono essere marchi di proprietà dei rispettivi titolari o marchi registrati di altre società e sono stati utilizzati a puro scopo esplicativo.

La stesura del testo, l'impaginazione e la realizzazione delle immagini è stata eseguita sfruttando programmi rilasciati con licenze libere. Per tutto il resto consultare le note di licenza.

Questo testo è rilasciato in versione digitale con licenza Creative Commons '`Attribuzione - Non Commerciale - Condividi allo stesso modo`', versione 3.0, Italiano. Per maggiori dettagli sulla licenza consultare il sito *https://creativecommons.org/licenses/by-nc-sa/3.0/it/legalcode*. Per utilizzo di versioni stampate o per riproduzioni non digitali dell'opera, rivolgersi alla casa editrice Ledizioni di Milano.

Indice

Indice **v**

1 Introduzione **1**
1.1 I Sistemi Embedded 4
1.2 Struttura del testo 9

2 Concetti generali **11**
2.1 Sistemi “monoprogrammati” e “multiprogrammati” 11
2.2 Conversione AD e DA 14
2.3 Multiplexing 16
2.4 Controlli automatici 18
2.5 Modo di pensare 26
2.6 Real Time 30
2.7 Licenze Software 32

3 Richiami di elettronica **41**
3.1 Richiami sui principi 42
3.2 Forme d’onda 50
3.3 Componenti di base 53
3.4 PWM (Pulse Width Modulation) 67
3.5 Semiconduttori 68
3.6 Strumenti di misura 78
3.7 Montaggio fisico 85
3.8 Sensori 87
3.9 Attuatori 89
3.10 MEMS 90

4 Architetture Embedded **91**
4.1 Instruction Set Architecture 91
4.2 ARM 96
4.3 MIPS 98
4.4 ARC 99
4.5 AVR 100
4.6 Xtensa 101
4.7 Esempi pratici 102
4.8 Implementazione Hardware 114

5 Memorie, I/O e comunicazione **119**
5.1 Memorie 119
5.2 Bus e periferiche 126

6 Il sistema operativo **147**
6.1 Introduzione 147
6.2 I sistemi GNU/Linux 151
6.3 Root Filesystem 156
6.4 Bootloader 161
6.5 La fasi di boot 164

7 Configurazione GNU/Linux **167**
7.1 Ambiente di sviluppo 168
7.2 Buildroot 188
7.3 Dispositivo di memoria 196
7.4 Esempio di configurazione della SD 200

8 FreeRTOS **203**
8.1 Introduzione 203
8.2 Caratteristiche 204
8.3 Concetti di base 205
8.4 Usare FreeRTOS 208
8.5 Gestione della memoria HEAP 210
8.6 Task 212
8.7 Queue 215
8.8 Timer 218
8.9 Interrupt 220
8.10 Semafori 221
8.11 Esempio: LED Blink 222

9 Arduino e Wiring **225**
9.1 I punti di forza 227
9.2 I punti di debolezza 228
9.3 Software: struttura di uno *sketch* 229
9.4 Il linguaggio 230

10 Rete e protocolli **249**
10.1 Topologie di rete 252
10.2 Protocolli 253

A Ambiente di Test **263**
A.1 Linux Test Environment 263

B Esempi pratici **269**
B.1 Abilitare un LED tramite GPIO su Linux 269
B.2 Scrittura su I2C 272
B.3 Una "termo-ventola" con Arduino 273

Bibliografia **277**

Capitolo 1

Introduzione

Come l'avvento della metallurgia e la costruzione di armi e utensili a partire dal II millennio A.C. ha caratterizzato un avanzamento nelle possibilità e nella qualità della vita per le popolazioni, o come l'invenzione della macchina a vapore ha portato al cambiamento totale sulla produzione di beni nella cosiddetta rivoluzione industriale del diciottesimo secolo, anche le scoperte nel campo dell'ingegneria, nell'elettronica e nell'informatica del ventesimo secolo hanno introdotto l'umanità in quella che è stata definita l'età digitale o l'età dell'informazione[13]. È giusto ricordare che i fondamenti teorici dell'informatica, così come i primi calcolatori meccanici, sono assai precedenti alla tecnologia digitale; basti pensare al lavoro di precursori quali ad esempio Blaise Pascal nel diciassettesimo secolo, con la costruzione della *Pascalina*[1]) o al lavoro di Charles Babbage (con la progettazione e il tentativo di costruzione della omonima macchina differenziale) e di Ada Lovelace, a buon titolo considerata la prima "programmatrice" della storia[2].

Detto ciò, si tende a far coincidere l'inizio dell'era dei computer con le "macchine" basate sul concetto di *"Universal Com-*

[1] La *Pascalina* o macchina di Pascal, inventata dal omonimo matematico e filosofo francese nel 1642 fu uno dei primi calcolatori meccanici in grado di fare somme e sottrazioni.

[2] Ad Ada Lovelace si fa ricondurre il primo algoritmo pensato per essere processato da un elaboratore.

puting Machine" espresso da Alan Turing a partire dal 1936[68].

Dalla fine della seconda guerra mondiale in poi, la rapida evoluzione tecnologica ha mutato radicalmente l'immaginario legato al mondo degli elaboratori, trasformandoli da oggetti di enormi dimensioni e appannaggio di pochi esclusivi centri di calcolo a strumenti di dimensioni minuscole che permeano vari aspetti della vita comune.

Se da principio si identifica l'universo informatico con un unico grande settore di studio multidisciplinare, con contributi matematici, fisici ed ingegneristici, con l'andare dei decenni si è sempre di più differenziato in molti settori distinti, specifici per la tecnologia, la tipologia e l'ambito applicativo. Se all'inizio si poteva pensare al computer come l'immenso elaboratore, oggi parlando di computer si può pensare ad una enorme varietà di strumenti, siano essi computer da casa, telefoni, tablet, termostati, centraline di auto, centri di calcolo, server web, strumenti di pagamento elettronico, produzione industriale, sistemi biomedicali e tanti altri ancora. La sempre maggiore differenziazione dal mondo desktop/workstation ha reso necessario la presenza di professionisti specializzati e qualificati.

In questo testo si presenteranno gli aspetti legati alla macrocategoria tecnologica nota come *Embedded Computing*, offrendo una panoramica basilare sulle conoscenze necessarie per studiare e lavorare in questo settore in modo semplice e sintetico. L'*Embedded Computing* è di per se estremamente multidisciplinare in quanto unisce aspetti legati alla fisica, alla matematica e all'ingegneria ad una serie di pratiche ed usi derivati da lunghe e consolidate esperienze industriali.

Data la enormità di queste discipline prese singolarmente, lo scopo del libro è quello di offrire una percorso didattico ideato per studenti di corsi di ingegneria o informatica che dispongano già di basi tecniche, conoscenza e pratica d'uso del computer, esperienze pregresse con la programmazione imperativa ed il funzionamento basilare dei sistemi operativi. Per quanto possibile, si è cercato di mantenere una semplice soglia di accesso ai contenuti, ma alcuni capitoli non potranno essere compresi appieno senza le necessarie conoscenze pregresse. È inoltre da sottolineare che, nonostante sia stato dato un notevole spazio

a tematiche legate all'hardware, il destinatario ideale del testo è lo sviluppatore di software per sistemi embedded, che possiede le necessarie competenze legate allo sviluppo, e che vuole sfruttare le potenzialità offerte dall'hardware embedded. Non è scopo di questo testo invece affrontare l'argomento da un punto di vista squisitamente ingegneristico, quindi per coloro che sono intenzionati a progettare componentistica hardware.

Il linguaggio utilizzato per la stesura è in molti casi di natura tecnica e talvolta gergale. Tale scelta stilistica è fatta di proposito per favorire l'inserimento del lettore all'interno del gergo specifico, facilitandone le future interazioni con documentazioni e letteratura settoriale oltre che per mantenere il testo sintetico e senza perdita di intuitività. In alcuni casi infatti, porre spiegazioni e traduzioni di terminologia specifica può risultare controproducente, sia dal punto di vista della comprensione che da quello della correttezza. In altre parole, ogni contesto tecnico ha la sua terminologia, e l'apprendimento di quest'ultima è un requisito fondamentale per raggiungere la piena confidenza con l'argomento trattato.

Nota bene: da sempre il mondo embedded è stato piuttosto variegato in tema di licenze di rilascio, libertà di utilizzo, accesso al codice e possibilità di modifica: da soluzioni interamente libere sia dal punto hardware che software (ad esempio i prodotti della famiglia Arduino), a soluzioni miste basate su software libero ma con l'aggiunta di software proprietario (ad esempio i telefoni su base Android) per arrivare a soluzioni interamente proprietarie e chiuse (fenomeno comune nei settori *Industrial* e *Automotive*). Gli autori di questo testo sono dichiaratamente sostenitori del Software Libero[3], e ciò ha sensibilmente condizionato le scelte delle piattaforme approfondite in questo testo, con una accesa preferenza per quelle libere, relegando alla sola citazione, e solo laddove sia ritenuto necessario ai fini di una maggiore comprensione, quelle proprietarie.

[3] `https://www.gnu.org/philosophy/free-sw.html`

1.1 I Sistemi Embedded

Con la terminologia **Sistema Embedded** (tradotto testualmente *"sistema incorporato"*) si tende ad indicare l'insieme composto da hardware e software (occasionalmente definito firmware) dedicato a specifici scopi (*"specific purpose"*) i cui elementi siano tutti quanti integrati ed incorporati. Tale definizione può tuttavia risultare alquanto generica e fuorviante in quanto il numero dei dispositivi a rientrare sotto questa nomenclatura è elevato e composto da elementi assai diversificati per architettura hardware e software. Per comprendere meglio quale sia l'universo dei sistemi embedded è possibile fornire alcuni esempi dei settori di utilizzo e sviluppo di tali sistemi:

- avionica, aeronautica, guida inerziale, sistemi di volo
- POS e bancomat
- stampanti e fotocopiatrici
- elettrodomestici e domotica
- router, switch, firewall
- decoder, media player, telefoni, SmartTV
- apparecchiature medicali
- Internet of Things[4]

L'implementazione di questi dispositivi può richiedere differenti caratteristiche tecniche in base alle finalità per cui vengono progettati. La potenza computazionale di un sistema embedded è un fattore discriminante per identificare le varie categorie sulla base dell'architettura hardware necessaria. È possibile infatti fornire una prima suddivisione sulla base del tipo di elaboratore utilizzato in tre macro-gruppi:

- **PLC** (*Programmable Logic Controller*)
- **Microcontrollori**
- **SoC** (*System-on-Chip*)

[4]Tale dicitura, pur molto di moda, non offre una definizione precisa sulla natura dei dispositivi, ma piuttosto, rappresenta una macro-categoria dove vengono inclusi dispositivi di dimensioni, scopi e potenze computazionali differenti, purché dotati di elementi di connettività.

1.1.1 PLC

I PLC (o Programmable Logic Controller) sono dei dispositivi relativamente semplici, generalmente pensati per lavorare su sistemi di automazione e non destinati alla produzione su vasta scala; in molti casi vengono programmati individualmente per svolgere una sola funzione specifica. Sono generalmente dotati di una buona connettività input/output per consentire la possibilità di interazione con sensori di vario genere (es. temperatura, pressione, peso e posizionamento) ed attuatori (es. cilindri idraulici, relè, motori elettrici o uscite analogiche). Un altro importante fattore dal punto di vista della progettazione di un PLC può essere legato alle condizioni di lavoro a cui deve essere sottoposto, quali ad esempio temperatura, umidità, polvere, stress meccanico e vibrazioni.

In base allo standard IEC 61131-3[36], i PLC sono programmabili con 4 linguaggi di programmazione, 2 testuali (ST e IL) e due linguaggi *grafici* (LD e FBD):

- **FBD** (Function Block Diagram)
- **LD** (Ladder Diagram)
- **ST** (Structured Text) simile al Pascal
- **IL** (Instruction List) simile all'assembly

Per quanto i vari prodotti sul mercato condividano i concetti base di programmazione dei PLC, tali dispositivi non sono interscambiabili o compatibili tra di loro. Talvolta tale incompatibilità è presente anche tra dispositivi dello stesso produttore. I PLC ancora oggi rappresentano un ramo interessante dello sviluppo nel mondo embedded, ma molto specifico e settoriale, e per questo non rientrano negli scopi di questo testo.

1.1.2 Microcontrollori

Analogamente allo sviluppo delle CPU (Central Processing Unit) *general purpose*, per intendersi quelle destinate al mercato dei computer portatili, desktop o workstation, già a partire degli anni 70 si sono distinti i primi **Microcontrollori** o **MCU** (MicroController Unit). L'idea era quella di fornire un sistema hardware completo all'interno del singolo chip, tanto che in un primo momento questi dispositivi erano definiti come "Compu-

ter on Chip". Storicamente, il primo elaboratore della categoria è stato l'Intel 8048 del 1975 con RAM e ROM all'interno del chip di elaborazione. Sin dai primi sviluppi, su alcuni modelli era possibile caricare del software adattato allo scopo designato su una apposita EPROM (vedere sez. 5.1.2). Tra gli anni '70 e gli anni '90 lo sviluppo dei microcontrollori è stato costante, aumentando quelle che erano le dotazioni disponibili. In particolare l'avvento delle EEPROM e successivamente delle Flash Memory (vedere sez. 5.1.2), hanno consentito un notevole miglioramento nei tempi di sviluppo e nelle possibilità di programmazione. Raramente su questi dispositivi, se non su quelli dell'ultima generazione, è possibile sfruttare dei sistemi operativi simili a quelli utilizzati in ambito domestico, sia per le ridotte performance computazionali, sia perché solitamente sono dotati di scarso spazio di memorizzazione. Per questo motivo molti produttori forniscono ambienti di sviluppo (sia ad alto sia a basso livello[5]) specifici per le singole piattaforme, in grado di fornire gli strumenti fondamentali per la creazione di firmware ad-hoc. Su molti microcontrollori moderni sono disponibili anche avanzati strumenti di debug e di scrittura della memoria interna (es JTAG, cfr. sezione 5.2.10). Ad oggi si parla di MCU caratterizzati da capacità di calcolo relativamente limitate (circa 200 MHz nei modelli di punta, ma con una buona casistica sotto i 100 MHz) se paragonate a quelle di CPU *general purpose* (In alcuni casi sino a 4 GHz) o dei SoC. Tale carenza dal punto di vista delle prestazioni è ripagata da ottimi valori di consumo energetico (assai inferiori ad 1 W contro i 30-110 W di una CPU *general purpose*) e da eccezionali fattori di efficienza termica. Bisogna inoltre considerare che il prezzo per singola unità può essere assai inferiore se paragonato ad altre categorie di elaboratori, arrivando in alcuni casi a costare alcuni ordini di grandezza meno di una comune CPU da workstation. Tali caratteristiche hanno determinato l'enorme successo dei MCU, che rappresentano buona parte del mercato dei sistemi embedded. Pur se non adatti ad applicazioni ad alta richiesta computazionale, so-

[5]Nel testo si farà spesso riferimento a linguaggi di "alto" e "basso" livello, indicando con tale dicitura la vicinanza (basso) o la lontananza (alto) di un linguaggio di programmazione dal linguaggio macchina.

no eccellenti in quei sistemi embedded che svolgono operazioni semplici quali le misurazioni, la raccolta di dati ambientali, sistemi di controllo, regolatori ma anche in ambito industriale ed *automotive*[6]. Inoltre, se utilizzati in prodotti soggetti ad economia di scala, il basso costo delle MCU può far considerare queste ultime come alternative più interessanti rispetto ai PLC, nonostante i maggiori costi di sviluppo software. La dotazione di serie di un moderno MCU può essere molto ricca:

- Unità di elaborazione
- Memoria dati (RAM o EPROM)
- Oscillatore[7](esterno o interno)
- Memoria programma (ROM, EPROM, FLASH)
- GPIO (General Purpose Input Output)
- Porte di comunicazione base (USART, I2S, SPI, I2C, USB)
- Porte analogiche (DAC, ADC, PWM)

Nei modelli più avanzati sono disponibili anche interfacce più complesse come WiFi, ZigBee, Ethernet, Touch Screen e LCD.

Nei prossimi capitoli verranno presentati alcuni dei dispositivi qui menzionati.

1.1.3 SoC

I **SoC** (acronimo di *"System on Chip"*) rappresentano l'ultimo stadio evolutivo dei sistemi embedded. Il termine, assai generico, è utilizzato per rappresentare un'ampia gamma di prodotti dalle maggiori potenze computazionali rispetto ai PLC e alle MCU e da dotazioni assai più ricche. La definizione di *System on Chip* deriva dal fatto che in molti casi si tratta di singole unità che al loro interno contengono tutte o quasi le componenti del sistema. A seguire alcuni esempi di componentistica reperibile all'interno un SoC;

- CPU

[6]Con il termine *automotive* si fa riferimento a tutta la tecnologia basata su elaboratori e sensori utilizzata all'interno delle automobili di nuova generazione, siano essi sistemi di intrattenimento che dispositivi di sicurezza. In altri contesti, questo termine può assumere significati differenti.

[7]Circuito elettronico che genera forme d'onda di frequenza, utilizzato come sorgente di clock.

- GPU (Graphic Processing Unit)
- ISP[8]
- Decoder/Encoder video
- Dispositivi di connettività
  (WiFi, Ethernet, Bluetooth, PAN...)
- Display Controller
- Audio Controller
- SPI, I2C, I2S, seriali, GPIO
- USB
- ADC/DAC
- Sensori
  (accelerometri, giroscopi, magnetometri, termometri...)
- GPS
- Modem
- Memoria RAM (sporadicamente)
- Memoria NAND/Flash (molto raro)

Pur esistendo SoC assai ricchi dal punto di vista della dotazione, è alquanto improbabile che tutti gli elementi presenti nella lista siano contemporaneamente presenti al loro interno. In generale, in base al costo e allo scopo di elezione, è possibile trovare un sottoinsieme di questa componentistica.

I SoC sono utilizzati praticamente ovunque, sia in ambito *automotive*, militare, domotico[9], Set Top Box (lettori multimediali, televisori, sistemi di "infotainment" domestici e da viaggio), navigatori satellitari, computer portatili, e negli ultimi anni hanno trovato vastissima applicazione nel settore della telefonia mobile. In termini pratici, si può affermare che siano presenti praticamente in ogni aspetto della vita digitale dell'uomo moderno. Data l'enormità del campo applicativo in cui i SoC sono utilizzati, risulta quantomeno complesso riuscire a delinearne una specifica generale.

A partire dagli anni 80 in poi, ARM e MIPS (vedere sez. 4.2 e 4.3) sono state le due grandi architetture di riferimento nel settore dei SoC, anche se esistono altre architetture che operano

[8] Acronimo di *Image Signal Processor*, elabora le informazioni provenienti dai sensori immagini, convertendole e filtrandole

[9] Con il termine *Domotica* (o *Home Automation*, si intende la categoria di sistemi utilizzati nella automazione domestica e sistemi di controllo della casa.

nel medesimo settore (vedere cap. 4). Come vedremo in seguito, riferirsi genericamente ad un processore ARM o MIPS è per lo più un errore del linguaggio comune derivato da abitudini legate al mondo Desktop/Workstation; a differenza di quanto sia possibile vedere nel campo delle CPU general purpose, il modello di business di questi prodotti è assai differente (si veda la sezione 4.8).

In tempi più recenti la distinzione tra MCU e SoC si è fatta più sottile, tanto che in alcuni contesti il termine è utilizzato in accezione quasi equivalente per dispositivi che dispongono di una architettura da microcontrollore, ma di una dotazione da SoC.

1.2 Struttura del testo

Questo testo è organizzato secondo la seguente struttura:

- Capitolo 2, "Concetti generali": i mattoni di base legati al mondo dell'informatica, della programmazione e dei sistemi operativi;
- Capitolo 3, "Richiami di elettronica": fornisce le conoscenze minime per capire l'interfacciamento elettrico tra un sistema embedded e il mondo fisico;
- Capitolo 4, "Architetture Embedded": panoramica delle piattaforme embedded più diffuse;
- Capitolo 5, "Memorie, I/O e comunicazione": panoramica sulle tecnologie di memorie e comunicazione dei sistemi embedded.
- Capitolo 6, "Il sistema operativo": approfondimento sui sistemi embedded di fascia "alta", con particolare riferimento a GNU/Linux;
- Capitolo 7, "Configurazione GNU/Linux": preparazione di un sistema operativo (basato su GNU/Linux) da installare su una piattaforma embedded;
- Capitolo 8, "FreeRTOS": approfondimento sul diffusissimo "sistema operativo" per piattaforme embedded real-time;
- Capitolo 9, "Arduino e Wiring": approfondimento su una piattaforma embedded di fascia "bassa", senza sistema

operativo, è stata scelta la più diffusa attualmente: Arduino;
- Capitolo 10, “Rete e protocolli”: panoramica sui protocolli di rete e introduzione ad alcuni protocolli di comunicazione di uso diffuso nei sistemi embedded;
- Appendice A, “Ambiente di Test”: preparazione di un ambiente di sviluppo/testing basato sull’interazione fra board e workstation;
- Appendice B, “Esempi pratici”: alcuni esempi pratici di programmazione ed uso di comuni sistemi embedded.

Capitolo 2

Concetti generali

In questo capitolo richiamiamo brevemente alcuni concetti necessari ad una piena padronanza del mondo embedded, legati al campo dei sistemi operativi, alla programmazione, all'informatica in generale.

2.1 Sistemi "monoprogrammati" e "multiprogrammati"

La distinzione tra sistemi cosiddetti "monoprogrammati" e "multiprogrammati" ha senso ormai solo in due casi: nella trattazione della storia dell'Informatica e nel contesto embedded. Intuitivamente, e forse anche ovviamente, i due termini si riferiscono ad ambienti operativi che supportano l'esecuzione "contemporanea"[1] di un solo programma o di più programmi.

I sistemi monoprogrammati rappresentano la realizzazione "naturale" di un computer. Storicamente, infatti, le prime istanze di calcolatori erano in grado di eseguire un solo programma (e una sola istruzione) alla volta. I più avanzati potevano eventualmente essere riprogrammati, cosa che normalmen-

[1] La contemporaneità (più o meno reale), il *time-sharing*, ecc. sono concetti che possono essere approfonditi dal lettore interessato anche ai temi tipici dei corsi di Sistemi Operativi [65]. Al nostro lettore basti il significato intuitivo.

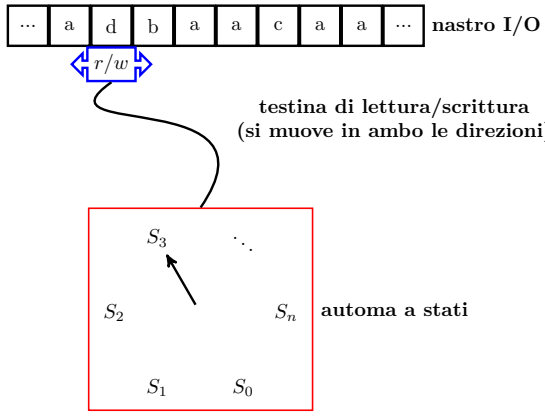


Figura 2.1: Macchina di Turing

te prevedeva lo "spegnimento"[2], la nuova programmazione[3] e la ripresa/riavvio del programma.

Una architettura semplice che si può immaginare per un computer è una più o meno elaborata Macchina di Turing [26] (in breve "MdT") di cui un esempio è rappresentato in figura 2.1. Una MdT è composta da una "macchina a stati"[4] dotata di una o più testine/cursori che leggono e scrivono simboli su uno o più nastri. Le "decisioni" vengono prese in funzione dei simboli sui nastri, infatti una MdT (immaginiamo la versione più semplice, con un solo nastro) esegue in continuazione il seguente *loop*:

- legge il simbolo sotto al cursore
- interpreta il simbolo secondo la macchina a stati interna

[2]Non necessariamente elettrico, si trattava di sospensione della normale esecuzione.

[3]Ad es. mediante sostituzione dei nastri o delle schede perforate o mediante riscrittura (anche a mano, mediante interruttori!) della memoria.

[4]Una "macchina a stati" è un meccanismo capace di "decisioni" nel senso che può reagire ad un input (un simbolo in ingresso) azionando un comportamento funzione sia del simbolo ricevuto sia dello stato interno in cui si trova. L'esempio più banale di una macchina a stati è un pulsante per illuminazione "bistabile": ogni volta che viene premuto agisce diversamente, se la luce era accesa la spegne e viceversa.

- sposta il cursore
- (opzionalmente, prima o dopo lo spostamento) scrive un simbolo sul nastro

Le MdT sono artifici matematici per ragionare formalmente[5] però sono anche utili come metafora semplice per comprendere il funzionamento di un computer odierno.

Se ad esempio volessimo tracciare un parallelo fra MdT e computer potremmo dire che la macchina a stati è la CPU, un nastro è memoria[6], l'alfabeto dei simboli sul nastro è rappresentato da numeri binari (di dimensione arbitraria, es. 8/16/32 bit) e i movimenti sul nastro sono di lunghezza arbitraria compatibilmente con la lunghezza finita del nastro/dimensione della memoria.

Una miglioria importante al concetto di MdT è la possibilità di modificare la macchina a stati interna utilizzando un nastro apposito, cioè la possibilità di "dire" alla MdT di **riprogrammarsi**[7] leggendo la definizione della macchina a stati da uno dei nastri (o da una zona "speciale" di un nastro): ecco che otteniamo un computer moderno in cui i "programmi" (le definizioni delle macchine a stati) si possono attivare dinamicamente a scelta dell'utente.

Questa metafora ci viene utile quando dobbiamo parlare di "architetture": abbiamo la cosiddetta architettura di Von Neumann [22] (un personal computer odierno), una MdT in cui il nastro è unico e contiene sia i dati di lavoro che i programmi (in zone diverse e ben etichettate ovviamente) e poi troviamo la Harvard (ad esempio la *board* Arduino) in cui i nastri del programma e dei dati sono separati.

In una MdT, per cambiare comportamento (programma), bisogna sospendere il *loop*, riscrivere la parte di nastro contenente il programma e riavviarla. È anche immaginabile il caso in cui

[5] In informatica teorica infatti fioccano teoremi sulla complessità dei linguaggi riconosciuti dalle MdT e sulla equivalenza fra MdT di vario tipo.

[6] RAM se il nastro è sia leggibile che scrivibile, ROM se solo leggibile.

[7] Può essere fatto in due modi: "importando" l'intera macchina a stati dentro il cuore della MdT o avendo una meta-macchina a stati che ad ogni passo legge da un altro nastro la specifica dell'azione da intraprendere, ma la sostanza non cambia, si ottiene comunque una MdT il cui comportamento non è *hardcoded*.

il programma possa modificare se stesso[8] scrivendo simboli sul nastro dedicato ai programmi, ma il principio di funzionamento non cambia.

La definizione di "**monoprogrammato**" deriva proprio da questa limitazione; un sistema monoprogrammato è quello su cui è possibile eseguire un solo programma per volta. Ovviamente il programma può essere complesso *ad libitum* e prevedere molti "subcomportamenti" (funzioni/*subroutine*) che vengono attivati su condizione[9], ma le istruzioni (letture di un simbolo dal nastro di lavoro) vengono eseguite una ad una, senza nessun tipo di parallelismo. Ergo se è stata attivata una certa subroutine il resto del programma è disattivato finché tale subroutine non "esce", cioè restituisce il controllo alla *routine* cosiddetta "chiamante".

Un sistema "**multiprogrammato**" invece può essere banalmente dotato di più MdT (CPU o *core*) che funzionano in parallelo, anche sullo stesso nastro se si studiano meccanismi di mutua esclusione[10]. In alternativa può essere dotato di una sola MdT, complicandone la macchina a stati, potendo gestire l'esecuzione di più nastri-programma a rotazione (un simbolo del primo nastro, un simbolo del secondo, uno del terzo, e così via) rapida in modo da realizzare un parallelismo apparente (ad un essere umano).

2.2 Conversione AD e DA

Le conversioni AD (Analogico-Digitale) e DA (Digitale-Analogico) sono processi di trasformazione dei segnali che permettono il trattamento dei segnali analogici (mondo reale, fisico) da parte di sistemi digitali (computer).

La conversione **AD**, o "discretizzazione", trasforma un segnale che varia nel tempo con continuità in una serie discreta (appunto) di misurazioni (es. valori di tensione/corrente) che **in qualche modo** rappresentano il segnale originario.

[8] L'esempio canonico è il linguaggio LISP [63].

[9] Ad esempio simboli particolari incontrati sul nastro della MdT.

[10] O, più in generale, di protocolli di gestione della concorrenza.

Il procedimento è semplice: il segnale analogico in ingresso viene "campionato", cioè misurato a intervalli regolari. Il risultato dell'operazione di campionamento è rappresentato dalla sequenza di misure effettuate.

Per fare un esempio: se ipotizziamo in ingresso un segnale che assume il valore di 5V (Volt) per il 50% del tempo (diciamo 1/2 secondo) e di 0V per il restante e lo campioniamo ad un decimo del periodo (quindi ogni 1/10 di secondo) di quel segnale, otteniamo la sequenza di misure: 5, 5, 5, 5, 5, 0, 0, 0, 0, 0, 5, 5, 5, 5, 5, 0, 0, 0, 0, 0, 5, 5, 5, 5, 5, 0, 0, 0, 0, 0, 5, 5, 5, 5, 5, 0, 0, 0, 0, 0, 5, 5, 5, 5, 5, 0, 0, 0, 0, 0, ... Cioè otteniamo una rappresentazione "numerica" dell'andamento del segnale originale.

La "frequenza di campionamento" è la velocità con cui si fanno le misure, ed indica il numero di letture nell'unità di tempo. Il suo inverso, il "tempo di campionamento" specifica l'intervallo fra una misura e l'altra.

Ovviamente, soprattutto nel caso di segnali molto irregolari e variabili nel tempo, tanto più **frequentemente** si campiona tanto meglio si riesce a cogliere la forma del segnale in ingresso, complicando però la circuiteria e generando sequenze di valori via via crescenti[11].

L'espressione "in qualche modo" poco sopra indica l'impossibilità di rappresentare perfettamente il segnale analogico in ingresso, si pensi al caso classico del cinema: 24 fotogrammi in un secondo devono rappresentare situazioni che variano continuamente nel tempo, tralasciando tutto quello che avviene tra un fotogramma e l'altro. Ad esempio: un proiettile sparato da una pistola viaggia a circa 400 m/s[12], per percorrere la distanza di 3 metri tra l'assassino e la vittima nella stessa stanza impiega 0.0075 s mentre un fotogramma "dura" un ventiquattresimo di secondo (0.041666667 s, circa cinque volte tanto). Un evento che duri molto meno del "tempo di campionamento" può sfuggire (si vede il lampo, ma non il proiettile). Eventi che durino nell'intorno del tempo di campionamento generano informazioni

[11] Ad es. il campionamento standard dei CD audio è a 44.1 kHz a 16 bit per canale, ergo vengono generate 88200 misure al secondo, in byte fanno 176400 byte/s.

[12] Ordine di grandezza, dipende moltissimo dal tipo di cartuccia, polvere da sparo e dal peso del proiettile.

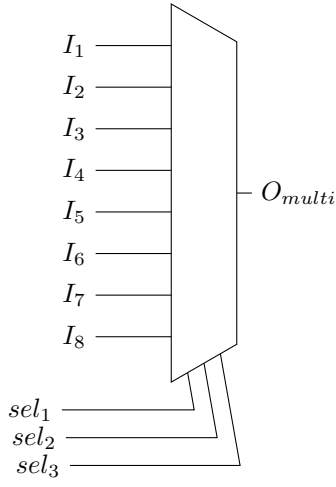


Figura 2.2: Schema di multiplexer

"curiose" (effetto "sfarfallio" o movimento retrogrado delle razze delle ruote delle auto, sempre nel cinema) o più in generale distorte (cfr. Teorema del campionamento in [17]).

La conversione **DA** è il processo inverso rispetto alla AD, si tratta in questo caso di generare un segnale continuo a partire da una sequenza di valori (misure). Le circuiterie per implementarla sono "semplicemente"[13] dei generatori programmabili di tensione.

2.3 Multiplexing

Nel mondo delle piattaforme embedded capita con relativa frequenza di dover gestire informazioni provenienti da sensori (o inviate ad attuatori) in numero maggiore rispetto alla disponibilità di ingressi/uscite (GPIO) della *board* in uso. In que-

[13]Più o meno sofisticati, il problema principale da risolvere è come far variare la tensione di uscita nell'intervallo tra un campione in ingresso e l'altro, di solito lo si fa interpolando con funzioni lineari/sinusoidali.

sti casi possiamo applicare proficuamente qualche tecnica di *multiplexing*.

Il concetto di *multiplexing* è generico, deriva dal contesto della scienza della trasmissione di informazioni, definisce quali sono le metodologie per condividere un canale per trasportare più flussi informativi **contemporaneamente** (pagando in termini di complicazione tecnica, banda passante, rumore, ecc.).

Rimandando ad un testo specifico (ad es. [17]) per gli aspetti formali, descriviamo le due principali modalità di implementare il *multiplexing*.

Divisione di frequenza: i segnali da trasmettere contemporaneamente vengono inviati sul canale usando frequenze diverse, ognuno dei riceventi deve sintonizzarsi sulla frequenza che gli compete come nel caso delle trasmissioni radio (es. banda FM 88-108 MHz). Se i segnali sono abbastanza "lontani" (in frequenza) uno dall'altro i ricevitori riescono ad ascoltare solo quello selezionato[14].

Divisione di tempo: i segnali da trasmettere "contemporaneamente" vengono inviati sul canale in alternanza veloce (i.e., prima uno, poi il secondo, poi il terzo, e tutti gli altri a seguire in rapida successione). Il ricevente, il cosiddetto *de-multiplexer* redistribuisce l'informazione facendo da "centralino", smistando rapidamente le comunicazioni, simulando un quasi "tempo reale". Un esempio **non veloce** è la diffusione di avvisi mediante altoparlanti nei luoghi pubblici: l'annunciatore legge dei messaggi uno dopo l'altro[15] sullo stesso canale, i riceventi sono indirizzati per nome.

Nel contesto dell'elettronica embedded i *multiplexer* (a divisione di tempo o a "selezione") si usano ad esempio per "moltiplicare" un ingresso digitale/analogico per leggere informazioni da più sensori senza dover dedicare un *pin* separato per ogni sensore.

Con riferimento alla figura 2.2, esaminiamo il funzionamento di un *multiplexer* e vediamo come lo si può controllare:

[14] Unica eccezione è Radio Maria.

[15] "Il bambino Luigino è pregato di recarsi alla reparto giocattoli", "Il signor Rossi è pregato di portarsi all'imbarco", ecc.

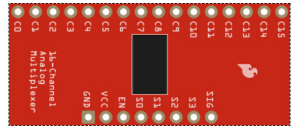


Figura 2.3: Multiplexer (fonte: Fritzing)

- agli ingressi I_n vengono collegati i vari sensori di cui leggere le misure
- sull'output O_{multi} troveremo di volta in volta uno dei segnali in arrivo dagli I_n
- In particolare troveremo il segnale "selezionato" mediante la configurazione di $sel_{1/2/3}$ che viene effettuata in binario[16].

Con questo "multiplexer a 8 vie" stiamo gestendo (alternativamente, si intende) 8 ingressi con 4 piedini (1 di ingresso vero e proprio + 3 di selezione), cioè un risparmio del 50%. Aumentando il livello di multiplexing il rapporto migliora perché ad esempio con 5 piedini (1 ingresso, 4 selezione, cfr. figura 2.3) si controllano 16 ingressi (risparmio 70%) e così via.

Il "prezzo" che si paga per questo risparmio di piedini risiede nella maggiore complicazione dal punto di vista software: infatti per leggere/scrivere un I/O *multiplexed* bisogna prima comandare il multiplexer e poi effettuare l'azione.

2.4 Controlli automatici

Il ruolo più frequente[17] di un sistema embedded è quello del controllo (di funzionamento e di "salute") di un sistema fisico.

Lo "stato"[18] di un sistema viene solitamente specificato in maniera formale[19] in fase di progettazione. In produzione lo sta-

[16] Applicando segnali HIGH/LOW ai *sel* si seleziona uno dei $2^3 = 8$ ingressi, ad esempio 000 seleziona I_1, 001 selezione I_2 e così via.

[17] Anche se recentemente si vedono sempre più spesso applicazioni di *entertainment*, ad esempio.

[18] Inteso come pluralità di condizioni a fronte di situazioni ambientali.

[19] In primis mediante equazioni differenziali, ma anche automi a stati, macchine di Turing, reti di Petri, code, ecc.

to effettivo del sistema viene osservato per verificare il corretto funzionamento rispetto alle specifiche.

L'automazione è una scienza [49, 9] a tutti gli effetti, a base fortemente matematica ed in questa sede si intende darne una sintetica visione qualitativa.

Un sistema può essere descritto tramite variabili di ingresso (e.g., una tensione di alimentazione) e di uscita (e.g., il numero di giri di un motore).

Le variabili di **ingresso** si dividono in:

- **di controllo:** parametri che si possono modificare direttamente (per proseguire l'esempio, il variarne della tensione di alimentazione) per influenzare lo stato del sistema;
- **di ambiente/disturbo:** parametri che non si possono modificare direttamente/semplicemente (e.g., umidità e temperatura dell'ambiente in cui è posizionato il motore di cui si vuol controllare il regime), ma che possono influenzare il comportamento globale del sistema.

Le variabili di **uscita** (o di stato) si dividono in:

- **di prestazione:** l'*output* "effettivo"[20] che si intende ottenere dal sistema;
- **intermedie:** variabili tipicamente più facilmente misurabili rispetto a quelle "di prestazione" ma ad esse direttamente collegate (e.g., la misura della temperatura sulla parete esterna di un forno utilizzata come evidenza della temperatura interna).

Il modello tipicamente utilizzato per rappresentare il funzionamento del sistema definisce una c.d. "funzione di trasferimento" f, nella sua forma generica:

$$\frac{dU(t)}{dt} = f(U(t), C(t), A(t)) \tag{2.1}$$

Dove U è il vettore delle variabili di uscita, C il vettore delle variabili di controllo e A il vettore delle variabili ambientali.

La formula 2.1 definisce la variazione di stato nel **tempo** come dipendente dai valori delle variabili di ingresso, dai disturbi

[20] Si potrebbe dire "vendibile".

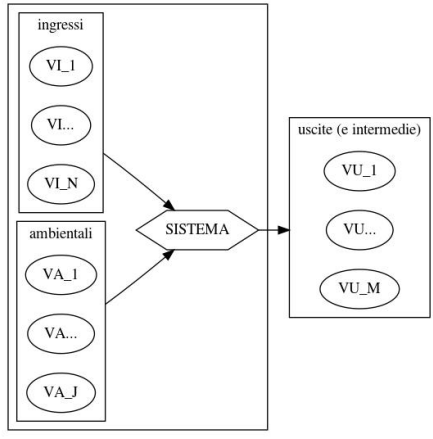


Figura 2.4: Open loop

e dai **valori delle variabili di uscita** (precedenti[21]). Quest'ultima dipendenza non è sempre applicata/applicabile, cioè non è detto che il vettore delle variabili di uscita faccia parte dell'input del modello, che in questo caso si riduce alla:

$$\frac{dU(t)}{dt} = f(C(t), A(t)) \tag{2.2}$$

Se le variabili di uscita non vengono prese in considerazione si parla di modello a "**anello/*loop* aperto**" o "senza retroazione/*feedback*" (figura 2.4, formula 2.2), nell'altro caso invece si parla di "**anello/*loop* chiuso**" o "con retroazione/*feedback*" (figura 2.5, formula 2.1).

La differenza tra i due modelli è che in quello senza retroazione il comportamento atteso[22] del sistema è *hardcoded*, non c'è nessuna "lettura" (*feedback*) dell'effettiva efficacia dell'azione, mentre nel modello a retroazione è possibile codificare azioni correttive in funzione dell'output **effettivamente** ottenuto.

[21] Infatti, nel discreto, la funzione diviene $U(t+1) = f(U(t), C(t), A(t))$ cioè il valore della variabili di uscita all'istante $t+1$ dipende dai valori all'istante t.

[22] Salvo guasti ovviamente.

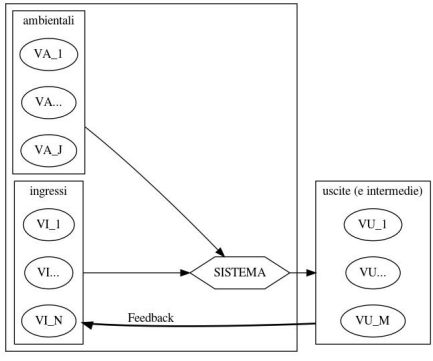


Figura 2.5: Closed loop

Per riprendere l'esempio del motore elettrico, un modello senza retroazione potrebbe definire la velocità di rotazione del motore come direttamente proporzionale[23] alla tensione applicata, ad es. affermando che:

$$Giri = 500 * TensioneIngressoInVolt \tag{2.3}$$

Se questo fosse un modello corretto potremmo calcolare il regime di rotazione nel caso di una alimentazione a 5V (i.e., 2500 giri), indipendentemente dal carico applicato al motore in questione. L'esperienza ci insegna che in realtà il regime di rotazione dipende anche dal carico del motore, ergo il nostro modello è sbagliato. Infatti il modello *open loop* può essere utilizzato solo nei casi in cui è possibile scrivere una funzione corretta, come ad esempio nel caso di una lampadina o di un LED in cui la luminosità non è praticamente influenzata da fattori esterni e dipende solo dalla tensione di alimentazione[24].

Ci sono alcune soluzioni per migliorare il modello esposto in formula 2.3, ad esempio:

1. inserire nella formula anche la misura del carico applicato al motore;

[23] N.B. non è comunque vero nei motori reali.

[24] Se non ci sono limitazioni alla disponibilità di corrente e nel range di tensioni di funzionamento corretto del *device*.

2. utilizzare la retroazione misurando il numero di giri effettivo del motore e applicarlo nel modello.

Se la prima soluzione appare semplice matematicamente (il carico diventa un fattore moltiplicativo da inserire nella formula) non lo è invece per realizzabilità dato che non è sempre banale misurare il carico applicato ad un motore: nel caso di un argano si può misurare il peso applicato con un dinamometro[25], ma nel caso di un veicolo in moto orizzontale ciò risulta più complicato.

La seconda soluzione sembra più interessante, si potrebbe esplicitare in forma di pseudocodice iterativo (con un *delay*[26] per andare a regime in maniera graduale) definendo una funzione che attua il regime desiderato:

```
/** Porta il motore al regime passato come parametro,
ritorna 'true' se ha successo, 'false' se ha
raggiunto valore tensione massimo senza ottenere
risultato (i.e., overload)
N.B. 'giri()', funzione che misura il regime
di rotazione attuale del motore */

boolean vaiAregime(int giriDesiderati){
 while(giri() <= giriDesiderati){
  if(!aumentaTensione(.5)) return false
  delay(500) // millisec
 }
 return true
}

/** Aumenta la tensione applicata al motore,
ritorna 'false' se viene raggiunta la
tensione massima applicabile
N.B.
1) 'MAXTENSIONE', costante che
modella la massima tensione sopportabile dal motore
2) 'tensione', variabile globale che rappresenta
la tensione applicata al motore */

boolean aumentaTensione(float incremento){
```

[25] A motore fermo, ovviamente!

[26] Da tarare con attenzione per non introdurre oscillazioni nel sistema, probabilmente, in un'implementazione sofisticata, il *delay* verrebbe calcolato dinamicamente misurando nel tempo le prestazioni del motore. Si veda la sezione 2.4.1 per una trattazione più generale.

```
 if(tensione+incremento > MAXTENSIONE) return false
 tensione+=incremento
 return true
}
```

In questo modo la tensione di alimentazione non è preimpostata/precalcolata, ma si "adatta" al regime effettivamente raggiunto, senza oltrepassare il limite massimo di alimentazione. Incidentalmente il rapporto fra regime e tensione è un buon indice del carico applicato al motore.

2.4.1 Tecniche di controllo in *closed loop*

Il *closed loop* ha il notevole vantaggio dell'adattabilità dinamica: potendo far riferimento all'efficacia dell'azione, le variabili di uscita, può realizzare la c.d. "correzione del tiro".

Naturalmente, se nell'*open loop* il costo di progettazione risiede nel perfezionamento della formula 2.2 che deve essere quanto più rappresentativa della situazione reale, nel modello *closed loop* si deve studiare al meglio il meccanismo di *feedback*, cioè il tipo di reattività della funzione di trasferimento. Una funzione troppo reattiva (in termini di velocità o entità della reazione) può causare oscillazioni, una troppo poco reattiva rischia invece di non raggiungere mai l'obbiettivo, cioè lo stato desiderato.

Per fortuna esistono alcuni *pattern* ben studiati e applicabili in vari contesti, vediamo il più classico.

PID (Proporzionale-Integrativo-Derivativo)

Il "controllo PID" modella la funzione di trasferimento come una **somma pesata di tre componenti** dipendenti dalla "funzione di errore" $e(t)$: differenza fra lo stato desiderato e quello misurato. Nell'esempio del motore elettrico già citato, che applicheremo fino alla fine di questa sezione, per poter calcolare la variazione di tensione da applicare al motore si potrebbe definire la seguente funzione d'errore:

$$e(t) = RegimeDesiderato - GiriMotore \tag{2.4}$$

Le tre componenti del controllo PID rappresentano le seguenti caratteristiche:

- **la distanza assoluta** dallo stato desiderato;
- **la storia** dell'evoluzione dello stato del sistema;
- **la velocità** di variazione dello stato.

La forma generale della funzione è la seguente:

$$\underbrace{P \cdot e(t)}_{\text{Proporzionale}} + \underbrace{I \cdot \int_0^t e(t')dt'}_{\text{Integrativo}} + \underbrace{D \cdot \frac{de(t)}{dt}}_{\text{Derivativo}} \tag{2.5}$$

Dove P, I e D sono i coefficienti "peso" che stabiliscono l'apporto delle tre componenti (distanza, storia, reattività). I coefficienti vanno inizialmente (in fase di progettazione) pensati/calcolati per avere una configurazione di *startup* con valori "sensati" e in seguito possono essere variati in fase di produzione per regolare il comportamento corretto del sistema (*tuning*).

Nella formula 2.5 la componente **proporzionale** utilizza l'errore "istantaneo"/attuale per valutare quanto il sistema sia lontano dall'obbiettivo, la componente **integrativa** calcola il cumulo degli errori da quando il sistema è stato attivato mentre la componente **derivativa** valuta la velocità di allontanamento/avvicinamento all'obbiettivo.

Possiamo provare ad applicare il modello al nostro esempio, ipotizzando una breve sequenza discreta di misurazioni[27] del regime del motore come:

```
Istante Regime  Errore (vs. desiderato)
t-6:    1000    400
t-5:    1100    300
t-4:    1150    250
t-3:    1170    230
t-2:    1200    200
t-1:    1250    150
t:      1300    100
```

Volendo raggiungere l'obbiettivo di un regime di 1400 giri e applicando la 2.5 le tre componenti diventano:

[27] Una finestra temporale discreta.

- **P:** $(1400 - 1300) = 100$
- **I:** $(400 + 300 + 250 + 230 + 200 + 150 + 100) = 1630$
- **D:** $(1400 - 1300) = 100$[28]

Quindi la formula finale per calcolare la correzione diventa:

$P \cdot 100 + I \cdot 1630 + D \cdot 100$

Con P, I e D a 0.5[29] il valore finale risulta: $50 + 815 + 50 = 915$ che potrebbe essere direttamente utilizzato come variazione in *millivolt*[30] di alimentazione del motore.

Vediamo ora cosa succede se all'istante $t + 1$ misuriamo un regime di:

```
Istante Regime  Errore
t+1:    1450    -50
```

I valori diventano:

- **P:** $(1400 - 1450) = -50$
- **I:** $(400 + 300 + 250 + 230 + 200 + 150 + 100 - 50) = 1580$
- **D:** $(1400 - 1450) = -50$

E la finale:

$0.5 \cdot -50 + 0.5 \cdot 1580 + 0.5 \cdot -50 = 740$

Cioè abbiamo sbagliato qualche coefficiente perché abbiamo calcolato ancora una variazione positiva della tensione nonostante si sia già superato il regime voluto. Probabilmente avremmo dovuto far pesare maggiormente P rispetto a I. Ad esempio tornando all'inizio e attribuendo $P = D = .9$, $I = .05$ si ottiene, all'istante t:

$0.9 \cdot 100 + 0.05 \cdot 1630 + 0.9 \cdot 100 = \sim 260$

Mentre al passo successivo $t + 1$:

$0.9 \cdot -50 + 0.05 \cdot 1580 + 0.9 \cdot -50 = -11$

Ottenendo finalmente una variazione negativa, ricordando infatti che siamo sopra al regime voluto ergo dobbiamo intuitivamente diminuire la tensione di alimentazione.

[28] Nel discreto la derivata equivale alla differenza fra due misure successive se la distanza temporale fra le misure è costante.

[29] Valori scelti a caso.

[30] Stiamo pensando a motori molto piccoli come quelli usati nel modellismo.

Anche solo da questo piccolo e semplice esempio si può intuire come sia importante applicare il modello di controllo corretto[31] e affinarne la parametrizzazione.

I rischi sono molteplici, principalmente:

- **mancato raggiungimento dell'obbiettivo** a causa di una funzione di trasferimento troppo "debole" (la variazione c'è ma è minima e/o non sufficiente), ad esempio con un P bassissimo, o addirittura "sbagliata", qualora non tenesse conto del segno dell'errore;
- **oscillazione**, ad esempio quando la correzione è molto "forte" e tende a sovracorreggere il sistema. Nel nostro caso, se avessimo una funzione con P molto elevato, potremmo cadere in una situazione in cui la variazione di tensione porta a superare il regime desiderato dovendo poi correggere sempre all'istante successivo, oscillando intorno all'ottimo[2].

2.5 Modo di pensare

Il titolo è altisonante, vero, ma il contesto embedded, rispetto ai sistemi cui oggi i programmatori "normali" sono abituati, è molto particolare e spesso obbliga a pensare diversamente per quanto riguarda lo sviluppo software (ma non solo).

Le "risorse" (CPU, RAM, dischi, potenza elettrica, ecc.) di un sistema embedded sono sovente estremamente limitate, ergo lo sviluppatore deve entrare in una modalità di progettazione/-programmazione molto attenta a non "sforare".

L'aspetto che ogni sviluppatore ha maggiormente a cuore è l'ambiente di *runtime* e quindi il contesto della progettazione e programmazione del software.

2.5.1 Memoria

Lo sviluppo di un programma, sia esso scritto in un linguaggio imperativo tradizionale o in un più moderno a oggetti (o altri

[31] Esistono anche molti altri modelli (oltre al PID) quali: "robusto", "adattativo", "ottimo", "sliding mode", "deadbeat".

paradigmi anche esoterici[32]), comporta comunque il trattamento di dati che prima o poi dovranno transitare dalla **memoria** che nel contesto "embedded" è di gran lunga inferiore a quella a cui si potrebbe essere abituati. In molti casi, Arduino su tutti (si veda sezione 4.7.2), parliamo di pochi kilobyte di RAM. Ergo nella progettazione di un programma lo sviluppatore dovrà ragionare molto bene e a priori sul numero e sul tipo delle variabili dichiarate nel corso del programma stesso. Ecco il motivo per cui prima ancora di cominciare a scrivere codice è bene studiare i tipi di dato (esempio in 9.4.1) disponibili nel linguaggio di programmazione scelto (o, a volte, imposto) per la piattaforma in uso. Utilizzare un `int` - tipicamente 2 byte - al posto di un `long` - tipicamente 4 byte - ove possibile (cioè se non si devono rappresentare numeri "grandi") fa risparmiare, ovviamente, 2 byte. Altra attenzione importante da mettere in atto è che molte "librerie"[33] per funzionare utilizzano memoria e, a meno di non andare a leggerne il codice sorgente, non è dato sapere quali e quante variabili dichiarano, purtroppo è raro che venga esplicitato il consumo di memoria nella documentazione.

Rimanendo sul tema memoria, oltre che allocando variabili esistono altri modi di consumarla: **invocando funzioni**. Ogni invocazione di funzione "costa" memoria: per chiamare una funzione l'ambiente di runtime deve salvare il contesto di provenienza[34] nello "*stack*" (area di memoria, contrapposta di solito allo "*heap*", dove vengono memorizzati dati di passaggio, appunto). A ciascuna invocazione corrisponde una crescita del-

[32] Ad esempio, per la piattaforma ESP8266 (citata successivamente) sono disponibili: FORTH, uLisp (versione embedded di Common Lisp), MicroPython, LUA, ...

[33] Insiemi di funzioni predefinite disponibili in molti ambienti per scopi usuali come accedere alla rete, leggere una SD card, ecc.

[34] Al minimo va memorizzato il PC (Program Counter, indirizzo di memoria della prossima istruzione da eseguire, è un registro della CPU) per sapere dove si trovava l'esecuzione prima di attivare la funzione, quindi 2, 4 o più byte a seconda dell'architettura del sistema. Nella peggiore delle ipotesi va salvato il PC, altri registri della CPU e i parametri passati alla funzione: tanti byte quanti sono i tipi degli argomenti della funzione stessa. Ad esempio, invocare una funzione `somma(int a, long b, double c)` ipotizzando 2 byte per `int` e 4 byte cadauno per `long` e `double` costa 10 byte solo di parametri passati.

lo *stack* che verrà poi parzialmente svuotato al ritorno dalla funzione.

Naturalmente questa situazione non deve spingere al ritorno del `GOTO`[35]. Semplicemente sarà sufficiente fare attenzione alla struttura di insieme del programma in modo da evitare per quanto possibile invocazioni multiple che rischierebbero uno "sfondamento" dello *stack* (il famoso errore "StackOverflow" che dà origine al nome di un notissimo sito[36]).

Infine, per concludere l'argomento "invocazione funzioni", diventa ora palese il motivo per cui si dovrebbe evitare l'uso della **ricorsione** in un contesto a bassa disponibilità di memoria. Anche riuscendo a scrivere correttamente una funzione ricorsiva complessa (cioè avendo una condizione di uscita sempre verificata) il costo di esecuzione è enorme in termini di memoria. Lo *stack* cresce monotonicamente fino al raggiungimento della condizione di termine per poi svuotarsi rapidamente. Se il procedimento ricorsivo genera un numero di iterazioni elevato lo *stack* viene "sfondato" con elevata probabilità.

E non va dimenticata la "**memoria di programma**" cioè lo spazio impegnato dal codice compilato: il "codice binario" che viene caricato sulla *board* occupa spazio (che può andare a scapito della "memoria di lavoro" in funzione della architettura interna della memoria, se condivisa o meno) in RAM e tale spazio, sovente limitato, deve essere gestito in maniera "centellinata". Mai come nel contesto embedded bisogna interpretare attentamente le informazioni emesse del compilatore sul binario generato: dimensione del file "oggetto" (*object file*), totale dei byte occupati dalle variabili statiche, ecc. Un fattore importante che influisce sulla dimensione finale del compilato è la "bravura" del compilatore nell'ottimizzare il codice e nella sua capacità di "linkare"[37] dinamicamente solo le parti di codice (ad esempio le

[35] Si veda la famosa lettera di Dijkstra [18] e si divaghi poi, se il lettore vorrà anche divertirsi un po', sulla pletora di articoli "Considered harmful" culminata in un meta-articolo di Meyer (`http://meyerweb.com/eric/comment/chech.html`

[36] `http://stackoverflow.com/`

[37] La traduzione letterale sarebbe "collegare", ma non si usa nella terminologia corrente italiana. Il *linking* è il processo per cui il compilatore realizza il "collage" dei pezzi di codice compilato dai vari file sorgente

librerie incluse nel sorgente principale) effettivamente utilizzate.

Rispetto ai sistemi desktop/server, molti sistemi embedded espongono architetture estremamente semplici, e nei sistemi più compatti è raro avere a disposizione una MMU (Memory Management Unit). Senza MMU non si ha nessuna protezione della memoria e non è possibile gestire memoria virtuale o indirizzamenti "esoterici", rilocazioni, ecc. Ergo, a meno di circonvoluzioni software (che comunque rallenterebbero fin troppo l'esecuzione e quindi raramente vengono messe in atto) per inserire controlli di coerenza nell'accesso alla memoria, tutti gli accessi alla memoria **non** sono controllati se non dal programmatore stesso, ad esempio:

- accesso ad un array: sforare l'indice vuol dire andare a leggere/scrivere in zone di memoria non previste con effetti potenzialmente catastrofici (si "sporca" la memoria dedicata ad altre variabili);
- sforamento stack/heap: riempire lo stack oltre il suo limite (andando quindi a "coprire" parte dello heap) non scatena nessun "avviso", anche in questo caso gli effetti sono potenzialmente catastrofici (idem).

2.5.2 CPU

I **processori**, salvo nei sistemi di fascia alta, sono molto semplici e non forniscono istruzioni dedicate alla gestione del multitasking/multithreading. Ciò che viene supportata normalmente è la funzionalità di trattamento degli *interrupt* (cfr. la sezione 9.4.11 e il Capitolo 4 di [70]) per cui è ad esempio possibile creare un programma che non sia obbligato a fare *polling* sulle porte di I/O. È possibile semplicemente "agganciare" una funzione all'arrivo di un segnale su una determinata porta, tale funzione verrà eseguita, **interrompendo** (da cui il nome *interrupt*) temporaneamente l'esecuzione normale del programma, alla ricezione del segnale. Non avendo disponibilità di funzioni per il multithreading vero e proprio (cioè con capacità di interrompere un flusso, salvarne lo stato, attivare un altro flusso, interromperlo e riprendere il precedente dallo stato in cui era stato lasciato)

creando l'eseguibile finale.

si ricorre spesso al cosiddetto "**multitasking cooperativo**". Il multitasking cooperativo è principalmente un modo di organizzare il proprio codice per permettere l'esecuzione di più flussi (in simulazione di contemporaneità) usando un singolo *thread* di esecuzione. Il meccanismo ruota attorno ad un *main* che esegue un *loop* il cui unico scopo è quello di invocare altre funzioni ("*task*") a istanti predefiniti nel tempo (es. ai secondi 5, 10, ecc. di ogni minuto chiama la "f1", ai 6, 11, ecc. la "f2" e via così). Ogni funzione-task deve essere breve (perché altrimenti mantiene il controllo per troppo tempo e fa sforare l'invocazione temporizzata delle altre) e deve, ove opportuno, saper gestire uno "stato" memorizzato in alcune variabili per poter lavorare "iterativamente"[38]. Ovviamente questo approccio complica la struttura del codice e non è esente da errori.

2.6 Real Time

Uno dei concetti chiave nella suddivisione tra i vari sistemi embedded è il cosiddetto **"real time"**; sintetizzando si può dire che per sistema *real-time* si intende quel sistema composto da hardware e software in grado di fornire una risposta o eseguire una operazione secondo tempistiche certe (indicate con il termine inglese di *"Time Constraints"*) o "deterministiche". Si noti bene che tale concetto non ha molto a che vedere con l'effettiva potenza computazionale del sistema; sistemi Desktop e Workstation, dotati di enormi potenze computazionali, non sono minimamente assimilabili a sistemi real-time, in quanto, pur se estremamente veloci, raramente possono garantire una risposta in un tempo determinato. Un sistema real-time deve quindi essere progettato di proposito, partendo proprio dalle sue spe-

[38] Si immagini di dover elaborare i dati di un array e che l'elaborazione di un dato sia "costosa" dal punto di vista del tempo di elaborazione. La funzione che deve gestire l'elaborazione dovrà trattare un sottoinsieme degli elementi dell'array (invece che tutta la struttura) ad ogni invocazione, salvando in una variabile "globale" il punto a cui era giunta (es. l'indice dell'array) per poi riprendere da lì alla successiva invocazione da parte del *main loop*.

cifiche in fatto di tempi di risposta (ai quali ci si riferisce con il termine inglese di *deadline*).

Al di là della definizione di sistema real-time, esiste una classificazione in tre differenti tipologie, basata su quali siano le effettive necessità temporali del sistema.

Hard real-time: il mancato rispetto di una deadline corrisponde al totale fallimento del sistema;

Soft real-time: il sistema non fallisce in caso di mancate deadline, ma "degrada"[39] all'aumentare delle tempistiche non rispettate;

Firm (*strict*) real-time: il sistema può tollerare un numero molto limitato di mancate deadline, ma fallisce in caso superino un certo limite.

Per rendere più chiare le differenze tra un *Hard* e un *Soft* real-time è sufficiente pensare a due tipiche applicazioni pratiche: il cosiddetto ABS ed un sistema di comunicazione vocale.

L'ABS (acronimo di Anti-lock Braking System) è un sistema di antibloccaggio dell'impianto frenante di una automobile che interviene in caso di frenate violente; come è facile intuire, se il tempo di reazione di questo sistema non è pressoché istantaneo, perde la sua funzionalità. In un sistema di comunicazione vocale invece, per quanto sia importante mantenere quanto più possibile il sincronismo tra gli elementi in comunicazione, è considerata accettabile una certa quantità di ritardo, o, in alcuni casi, la perdita di qualche informazione (i.e., parola). Ovviamente, quando il ritardo è eccessivo, la qualità del servizio diventa troppo bassa per essere in qualche modo utile.

Dal punto di vista hardware, a differenza di quanto avviene nei sistemi embedded general-purpose, i sistemi embedded real-time utilizzano architetture hardware compatte ed essenziali e sono progettati con il principale obiettivo della massima affidabilità (talvolta sacrificandone le prestazioni). Questi tipi di sistemi sono solitamente utilizzati in ambienti *mission critical*[40], e ciò richiede che tutto il design subisca avanzati processi

[39] La degradazione può essere sia sulle prestazioni del sistema, sia nella qualità del servizio offerto.

[40] Tale termine può indicare un ambiente in cui, se una determinata operazione non viene svolta nelle modalità prestabilite può produrre gravi

di verifica, validazione, certificazione e documentazione[41].

Analogamente a quanto avviene con l'hardware, anche il software per i sistemi real-time deve essere progettato specificamente per lo scopo, essenziale, affidabile ed altamente certificato. Per questo motivo si tende ad utilizzare una diversa categoria di sistemi operativi noti come RTOS (*Real-Time Operating System*). Il codice di questi RTOS è generalmente molto più compatto, con un numero ridotto di funzioni (se comparato con quelli Desktop/Workstation), e disegnato per essere ottimizzato su singoli compiti specifici.

2.7 Licenze Software

Premessa: gli autori di questo testo sono dichiaratamente attivisti a supporto del Software Libero in ogni contesto ove i suoi principi possano essere applicati. Le idee alla base del Software Libero non si limitano esclusivamente al software, ma si estendono per filosofia a tutta la produzione creativa e culturale. Ciò ha marcatamente influenzato le scelte sugli argomenti da trattare, cercando comunque di attenersi alla realtà e alla obiettività tecnica.

Un concetto che molte volte sfugge ai non addetti ai lavori è che il software in quanto tale non viene venduto come un qualsiasi altro oggetto, bensì viene concesso *in licenza*. Una licenza[42] è formalmente un contratto tra chi ha scritto il software e il suo utilizzatore finale. Essa descrive, spesso con minuzia di particolari, i termini e le modalità d'uso, le possibilità di modifica e di redistribuzione del software stesso. Sottovalutare l'aspetto della licenza, sia per l'utilizzatore che per lo sviluppatore può essere un errore pagato a carissimo prezzo, sia in termini tecnici che economici.

conseguenze.

[41] In questi casi vengono applicati processi formali di validazione del sistema, ad esempio basati su Reti di Petri [67].

[42] Per molti è solamente quel testo molto lungo che appare durante l'installazione di un nuovo software e che quasi nessuno legge (e ciò è male), di cui ci si sbarazza cliccando sul classico tasto "*I Agree*" (Accetto i termini di Licenza).

2.7.1 Licenze Proprietarie

Molto del software utilizzato a livello domestico ed aziendale, purtroppo, rientra nella categoria del software "proprietario"[43], rilasciato comunemente con licenze restrittive chiamate genericamente **EULA**[44]. Come principio generale, l'accettazione o meno di un EULA discrimina l'accesso e l'uso del software: senza accettazione, pur avendo pagato il prodotto, non si ha alcun diritto di uso. Per quanto questo tipo di licenza sia molto comune, in realtà ogni software proprietario ha una sua propria variante, scritta dell'azienda che rilascia il software, rendendo di fatto impossibile presentarne ogni singolo aspetto (basti pensare che anche versioni successive del medesimo software possono avere delle regole o limitazioni molto differenti, regolate quindi da EULA completamente diverse). Considerato ciò si possono comunque descrivere dei vincoli generali che non si applicano a tutti i software proprietari in assoluto, ma ad una loro nutrita maggioranza.

- **Divieto di far girare il programma su più dispositivi.** Molti prodotti, in particolare quelli relativi a sistemi operativi e applicazioni, impongono di comprare una licenza per ogni dispositivo hardware su cui il software in questione deve essere eseguito. In altri casi le aziende optano per una sola licenza di sfruttamento con il pagamento di una *royalty*[45] per ogni dispositivo utilizzato.
- **Divieto di fare *reverse engineering***[46]. Tale divieto non è esercitabile in tutte le legislazioni (trova la sua massima applicazione negli USA). Per i sostenitori del soft-

[43] Italianizzazione di "*proprietary software*" che rende bene l'idea dello scopo della licenza che avvantaggia, appunto, solo il proprietario dei diritti di copyright (l'autore del software) e relega l'utente a mero pagatore.

[44] End User License Agreement

[45] Termine inglese atto ad indicare il diritto del titolare di una proprietà intellettuale o brevetto ad ottenere compensazione economica per il loro uso.

[46] Termine inglese (talvolta tradotto con il termine "ingegneria inversa") che indica l'attività svolta per analizzare funzionalmente e strutturalmente un prodotto finito (sia hardware che software) cercando di ricavarne il maggior numero possibile di informazioni. Nel caso del software si mira ad arrivare sino al codice sorgente (quindi agli algoritmi) partendo da un binario.

ware libero, tale limitazione viene paragonata all'acquisto di una automobile con il cofano sigillato.
- **Divieto di rivendere il programma.** Tale limite però non è valido ovunque, e, per quanto scritto in licenza in alcune legislazioni non ha valore.
- **Divieto di studiare le prestazioni** del programma e di fare confronti con altri programmi e di pubblicare i risultati.
- **Divieto di modifica** del programma.
- **Molte licenze sono "a scadenza"**, dopo un certo periodo il programma cessa di funzionare.
- **Divieto di esportazione** del programma.
- **Divieto di fare copie di sicurezza.** In alcune legislazioni, ad esempio l'Italia, questo divieto non è valido.

Uno degli elementi che risalta in questa lista, oltre ai tanti limiti imposti, è ovviamente che il quadro normativo è assai variegato tra le varie aree geografiche, con differenze che talvolta generano confusioni risolte solo tramite lunghe, complesse e costose cause legali.

2.7.2 Software Libero ed Open Source

La quasi totalità del software presentato nei successivi capitoli è rilasciata e condivisa con licenze definite *FOSS* (*Free & Open Source Software*). In tal senso vale la pena approfondire cosa si intenda con il concetto di *Free Software* e con quello di *Open Source*: per quanto vengano utilizzati talvolta come sinonimi e nonostante molti software e licenze ricadano in entrambe le definizioni, le due definizioni possono avere connotazioni differenti.

Software Libero

Le **licenze libere** [62] per il software sono quelle che rispettano e proteggono le seguenti libertà fondamentali:

L0: **Libertà di eseguire** il programma come si desidera, per qualsiasi scopo.

L1: **Libertà di studiare**[47] come funziona il programma e di modificarlo in modo da adattarlo alle proprie necessità. L'accesso al codice sorgente ne è un prerequisito.

L2: **Libertà di ridistribuire** copie in modo da aiutare il prossimo.

L3: **Libertà di migliorare** il programma e distribuirne pubblicamente i miglioramenti da voi apportati (e le vostre versioni modificate in genere), in modo tale che tutta la comunità ne tragga beneficio. L'accesso al codice sorgente ne è un prerequisito.

Esiste un discreto numero di licenze che rientrano nei parametri delle quattro libertà sopra elencate e sono quindi riconosciute come licenze libere. E' possibile consultarne una lista esaustiva sul sito del progetto **GNU**[48]. Senza nulla togliere alle altre licenze, la **GNU GPL** (*GNU*[49] *General Public License*) nelle sue molteplici varianti rappresenta l'esempio più celebre di questa filosofia, veicolando molto bene lo spirito del Software Libero.

Open Source

Come già anticipato, il concetto di *Open Source* differisce per definizione da quello di Software Libero, fornendo dei criteri simili ma non identici (e, per alcuni, più restrittivi). Tale definizione è espressa nel seguente "decalogo" tratto dal sito della *Open Source Initiative* [31]:

- **Ridistribuzione libera**. La licenza non può impedire ad alcuna parte in causa la vendita o la cessione del software. Chiunque deve poter fare tutte le copie che vuole, venderle o cederle, e non deve pagare nessuno per poter fare ciò.
- **Codice sorgente**. Il programma deve includere il codice sorgente. Codice deliberatamente offuscato non è ammesso. Questo in quanto il codice sorgente è necessario per modificare o riparare un programma.

[47] Alla "libertà di studio", cioè l'accesso alla conoscenza, non sempre viene attribuita la giusta attenzione, si potrebbe osare elevarla al rango di più importante di tutte.

[48] `https://www.gnu.org/licenses/licenses.it.html`

[49] Acrostico "ricorsivo" che sta per *GNU is Not Unix*

- **Opere derivate**. La licenza deve permettere modifiche e opere derivate e deve consentire la loro distribuzione sotto i medesimi termini della licenza del software originale, in quanto il software serve a poco se non si può modificare per fare la manutenzione ad esempio per la correzione di errori o il porting su altri sistemi operativi.
- **Integrità del codice sorgente dell'autore**. La licenza può proibire che il codice sorgente venga distribuito in forma modificata solo se la licenza permette la distribuzione di pezze ("patch file") con il codice sorgente allo scopo di migliorare il programma al momento della costruzione.
- **Nessuna discriminazione contro persone o gruppi**. La licenza deve essere applicabile per tutti, senza alcuna discriminazione per quanto nobile possa essere l'obiettivo della discriminazione. Ad esempio non si può negare la licenza d'uso neanche a forze di polizia di regimi dittatoriali.
- **Nessuna discriminazione di settori**. Analogamente alla condizione precedente, questa impedisce che si possa negare la licenza d'uso in determinati settori, per quanto questi possano essere deplorevoli. Non si può dunque impedire l'uso di tale software per produrre armi chimiche o altri strumenti di distruzione di massa.
- **Distribuzione della licenza**. I diritti relativi al programma devono applicarsi a tutti coloro ai quali il programma sia ridistribuito, senza necessità di esecuzione di una licenza aggiuntiva.
- **La licenza non dev'essere specifica a un prodotto**. I diritti relativi a un programma non devono dipendere dall'essere il programma parte di una particolare distribuzione di software.
- **La licenza non deve contaminare altro software**. La licenza non deve porre restrizioni ad altro software che sia distribuito insieme a quello licenziato.
- **La licenza deve essere tecnologicamente neutra**. Nessuna clausola della licenza deve essere proclamata su alcuna singola tecnologia o stile di interfaccia.

Anche in questo caso, il numero di licenze compatibili con

la definizione di *Open Source* è assai ampio e molte di esse sono in comune con la lista di quelle compatibili con la definizione di Software Libero.

Semplificando a grandi linee la situazione, si può affermare che il *Software Libero* si concentra su valori etici e filosofici, mentre l'*Open Source* è legato a criteri pratici di sviluppo e rilascio.

2.7.3 Considerazioni sulle licenze

Per essere completamente obiettivi vale la pena chiarire alcuni concetti:

- Libero **non** vuol dire gratis: la libertà di uso, studio, modifica e redistribuzione non implica la gratuità. Un software libero può essere venduto. Esiste inoltre un considerevole numero di aziende che rilascia con licenza libera il codice, ma che si fa pagare il supporto professionale, la certificazione, l'assistenza e la formazione. E' anche vero che nel mondo del Software Libero è molto difficile ottenere una "rendita di posizione" (prezzi alti per mancanza di concorrenza), a tutto vantaggio degli utenti.
- Libero non implica il "Pubblico Dominio": è comune credenza che si possa prendere del codice rilasciato con licenza libera o Open Source e farne tutto quello che si vuole ignorandone la licenza. Ciò è assai sbagliato e pericoloso, soprattutto in ambito aziendale: attingendo ed utilizzando software FOSS, si è (quasi sempre) tenuti alle medesime clausole di rilascio espresse nella licenza originale. Più semplicemente, in generale, non si può prendere un software libero e trasformarlo a piacimento in software proprietario. Tale errata credenza ha portato non poche volte a contenziosi legali.
- Proprietario non vuol sempre dire assenza di accesso al codice: alcune aziende rilasciano il codice esclusivamente ai propri partner attraverso l'applicazione di vincoli NDA[50].

[50] *Non Disclosure Agreement*: una o più parti si impegnano a non rivelare informazioni definite confidenziali, pena pesanti risarcimenti pecuniari.

- Proprietario non vuol dire migliore: il prezzo del software non è indice in alcun modo della sua qualità.
- Proprietario può voler dire *lock-in*[51]: sviluppando un prodotto basandosi su software di cui non si può accedere al codice o che, pur avendo accesso al codice, non è possibile modificarlo e redistribuirlo, implica il più delle volte il dover continuare a pagare la medesima azienda, indipendentemente dalla qualità o della convenienza della medesima, per tutta la vita del prodotto in sviluppo.
- Libero non vuol dire manutenuto per sempre: se è vero che con il software FOSS il rischio di *lock-in* è assai limitato, esiste ed è concreta la possibilità che l'azienda o la *community* che porta avanti lo sviluppo, arrivata ad un certo punto possa cessare tale attività[52]. In questo caso, l'azienda utilizzatrice di software rilasciati con licenza FOSS per i propri prodotti può e deve considerare l'eventualità di farsi carico anche dello sviluppo del software FOSS in utilizzo.

Come si evince da queste considerazioni, è una discussione tutt'altro che leggera, e che per certi versi va avanti da oltre trent'anni. Gli aspetti sono molteplici e vanno considerati aspetti filosofici, economici, legali, tecnici e pratici. Per approfondimenti in italiano si consiglia la lettura del recente [57].

In ultima analisi si può dire che l'approccio del Software Libero è quello di considerare l'utente come un possibile collaboratore e sviluppatore (un partner!), mentre l'approccio proprietario inquadra il destinatario del software puramente come utilizzatore finale a cui vendere varie possibilità di azione.

Pur non potendo fornire un giudizio definitivo, che si rimanda alla considerazione del lettore, gli autori di questo testo

[51] Per esteso *Vendor Lock-In*: in economia indica un rapporto di dipendenza fra un cliente ed un fornitore di beni e servizi, usato generalmente con accezione negativa, in particolare riferendosi al costo considerevole per uscire dalla situazione di *lock-in*

[52] Esattamente come una azienda che produce software proprietario può fallire o dismettere un prodotto. Salvo che nel caso del software libero il prodotto rimane accessibile (basta averne almeno una copia, con i sorgenti ovviamente) indefinitamente.

sono fortemente convinti che, soprattutto per lo sviluppatore alle prime armi, il mondo del FOSS offra una maggiore possibilità di apprendimento e sviluppo con una richiesta economica assai inferiore (spesso zero se si esclude il costo del hardware).

Capitolo 3

Richiami di elettronica

Il presente capitolo è dedicato al riassunto di alcuni basilari concetti di elettricità, elettronica e fisica, il minimo indispensabile per poter collegare una piattaforma embedded (es. un Arduino o un RaspberryPI con i suoi GPIO - General Purpose I/O) al mondo fisico mediante qualche sensore e/o attuatore senza distruggere nulla né farsi del male dato che gli informatici puri non toccano mai un saldatore né si trovano mai ad affrontare alte tensioni a mani nude. Il taglio è volutamente abbastanza leggero e spesso metaforico, lo scopo è dare un'idea di come trattare il mondo "esterno" rispetto all'universo dei soli bit.

Per approfondimenti consigliamo un testo atipico nel formato di impaginazione, ma completo e ben spiegato, il famoso Mims[48]. Volendo approfondire ulteriormente senza per forza arrivare ad un testo universitario sull'elettronica si può affrontare tranquillamente il Neri[51], il testo per l'esame da radioamatore, in cui si trovano anche alcuni esercizi sulle leggi di Ohm.

3.1 Richiami sui principi

3.1.1 Elettricità

Elettricità (ed elettrone) è un termine che deriva dal greco *electron* (ambra, materiale noto per la facilità con cui permette di produrre carica elettrostatica per sfregamento): raggruppa l'insieme dei fenomeni fisici che coinvolgono il trattamento e lo spostamento di cariche elettriche e degli effetti ad essi collegati (calore, magnetismo, ecc.).

Ai fini di questo testo ci interesserà principalmente la gestione dei **segnali** elettrici, cioè di quelle grandezze di tensione e corrente che interessano i conduttori (o i semiconduttori) e a cui possiamo associare un'informazione. Non verrà trattata l'elettrostatica se non limitatamente alla descrizione di un condensatore.

Il termine "corrente" richiama il concetto di movimento, infatti si ha passaggio di corrente quando degli elettroni (le particelle subatomiche caricate negativamente che "orbitano" intorno ai nuclei degli atomi) liberi si muovono lungo un conduttore trainati da una differenza di potenziale elettrico. Tali elettroni possono essere usati per effettuare un lavoro, generare calore, scatenare reazioni chimiche, trasportare informazione, ecc.

Il **potenziale elettrico** è una forza di attrazione (creabile mediante pile/batterie, dinamo, condensatori, ecc.) che cerca di spostare degli elettroni da un punto all'altro (poli) di un circuito. Maggiore il potenziale, maggiore la forza ("elettromotrice") e quindi maggiore la quantità di elettroni che si possono spostare nell'unità di tempo. La velocità con cui si spostano gli elettroni passando da un atomo all'altro di un conduttore non è elevata (dell'ordine di qualche mm/s), ma l'effetto della catena di spostamenti (un elettrone spostatosi lascia un "buco" che trascina un altro elettrone) si muove a velocità comparabili con quella della luce.

Il potenziale prende il nome tecnico di "tensione" e si misura in Volt (simbolo `V`) mentre la quantità di corrente che scorre nell'unità di tempo si chiama "intensità" e si misura in Ampere (simbolo `A`). L'Ampere corrisponde alla quantità di carica (Coulomb, un "tot" di elettroni, per la precisione $\sim 6.24 \times 10^{18}$)

trasportata in un secondo. Quindi, in un conduttore, a tensione più elevata corrisponde maggiore intensità di corrente.

3.1.2 Legge di Ohm

In particolare fu Georg Ohm a studiare la relazione fra tensione e (intensità di) corrente, la c.d. "Legge di Ohm"[19] infatti lega queste due grandezze alla resistenza, così:

$$I = \frac{V}{R} \tag{3.1}$$

Cioè l'intensità di corrente in un conduttore è direttamente proporzionale alla tensione e inversamente proporzionale alla resistenza del conduttore (che trasporta gli elettroni). L'unità di misura della resistenza, che prende il nome dallo scienziato, è per l'appunto l'Ohm (simbolo Ω).

La resistenza "naturale" (chiamata *resistività*, simbolo ρ) di un conduttore è dovuta al tipo di materiale usato.

La resistenza complessiva di un circuito può essere modificata (maggiorandola) mediante l'inserzione di un componente elettronico chiamato "resistore" (cfr. Sezione 3.3.4). Ciò può servire ad esempio a limitare la corrente, a parità di tensione, ove non servano valori elevati, anche perché il calore dissipato da un circuito è proporzionale al quadrato della corrente che vi scorre, secondo la formula:

$$P(otenza) = R \times I^2 \tag{3.2}$$

Naturalmente l'equazione 3.1 può essere esplicitata in tutte le sue forme, cioè: $R = V/I$ e $V = R \times I$ a seconda di quale sia l'incognita e quali i dati noti.

E combinando la formula 3.1 con la 3.2 si può esprimere la potenza in funzione della tensione e resistenza invece che della corrente e resistenza o in funzione di tensione e corrente:

$$P(otenza) = R \times I^2 = R \times \left(\frac{V}{R}\right)^2 = \frac{V^2}{R} = V \times I \tag{3.3}$$

3.1.3 Leggi di Kirchhoff

Gustav Robert Kirchhoff nel 1845 formulò una generalizzazione delle leggi di Ohm definendo due leggi di conservazione:

- della corrente
- della tensione

La **legge di Kirchhoff delle correnti** (LKC o LKI o KCL) formalizza il fatto che in un "nodo" (punto di incontro di più percorsi all'interno di un circuito, giunzione) le correnti entranti e le correnti uscenti si bilanciano, in parole povere "non si creano né spariscono elettroni". Matematicamente:

$$\sum_{segmenti\ k\ entranti/uscenti\ nel\ nodo} I_k = 0 \tag{3.4}$$

La **legge di Kirchhoff delle tensioni** (LKT o LKV o KVL) formalizza il fatto che in un circuito la somma algebrica delle tensioni è 0, in parole povere "non si creano né spariscono differenze di potenziale". Matematicamente:

$$\sum_{sui\ tratti\ j\ che\ compongono\ il\ circuito} V_j = 0 \tag{3.5}$$

3.1.4 Continua e alternata

La corrente esiste in due forme: continua e alternata. La differenza consta nel comportamento degli elettroni che fluiscono nei conduttori, nel caso della corrente **continua** essi si muovono sempre nella stessa direzione, nella **alternata** cambiano verso, vanno cioè avanti e indietro secondo un periodo che prende il nome di "frequenza" che si misura in Hertz (ad esempio i 50 Hz della corrente a 220 V di rete italiana). In molti schemi si trovano simboli specifici che indicano i valori di tensione esplicitando se "continua" (e.g., `12 Vcc`, `12 V=`) o "alternata" (e.g., `220 Vac`, `220 V~`).

Le differenze fra i due tipi di corrente sono notevoli.

Ad esempio nella **produzione**:

- la corrente alternata viene prodotta meccanicamente (o elettronicamente negli *inverter*[1] moderni) muovendo/facendo ruotare un insieme di conduttori all'interno di un campo magnetico creando quindi per *induzione elettromagnetica*, appunto, una corrente nei conduttori stessi;
- la corrente continua invece viene prodotta chimicamente (pile/batterie, cfr. sezione 3.3.6), per accumulo elettrostatico e anche meccanicamente tramite dinamo (per la precisione in quest'ultimo caso si tratta di corrente c.d. "pulsante").

L'aspetto dell'**immagazzinamento** è fondamentale e dipende essenzialmente dalle tecniche di produzione citate. Infatti la corrente continua è immagazzinabile, negli "accumulatori" (chimici: batterie a base di materiali disparati come piombo, litio, nickel, ecc.) o nei "condensatori" (elettrostatici, cfr. sezione 3.3.5), mentre quella alternata non è direttamente accumulabile a meno di non trasformare l'energia elettrica associata in altri tipi di energia (e.g., *potenziale* nei bacini idroelettrici).

Altro fattore importante è la **trasformabilità** (e quindi indirettamente la trasmissibilità): la corrente continua non è trasformabile facilmente[2] mentre per quella alternata è sufficiente, appunto, un *trasformatore* (figura 3.1). Il trasformatore è un componente meccanicamente semplice: non ha parti in movimento e non contiene componenti attivi (semiconduttori), deve solo essere eventualmente raffreddato (tipicamente a olio) se deve gestire flussi elevati di corrente. Nella sua forma base è composto da due bobine (avvolgimenti di filo conduttore) vicine (normalmente sono avvolte sullo stesso nucleo) in modo che il campo elettromagnetico prodotto da una di esse (c.d. "avvolgimento primario") possa influenzare l'altra (c.d. "avvolgimento secondario") inducendo una corrente la cui tensione dipende da quella della prima bobina e dal rapporto fra i numeri di *spire*

[1] Apparato che trasforma una corrente in ingresso continua in una alternata in uscita, tipicamente a tensione diversa da quella di ingresso, e.g., gli inverter da auto/barca/camper che accettano 12/24 Vcc in ingresso e forniscono 220 Vac in uscita.

[2] Esistono convertitori continua-continua a semiconduttore, ma le dimensioni e il calore dissipato da tali apparecchi salgono molto rapidamente al crescere di tensione e soprattutto corrente gestite.

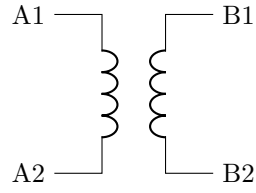


Figura 3.1: Simbolo del trasformatore

delle due bobine. Il trasformatore non funziona con una corrente continua (deve essere alternata o pulsante).

La trasformabilità influisce pesantemente sulla **trasmissibilità** a grandi distanze della corrente elettrica. Infatti, applicando le varie forme della Legge di Ohm (vedi formula 3.1), si può intuire che per portare "potenza" (capacità di produrre lavoro) a distanza si deve trasportare tensione (forza) e corrente (intensità) dato che la potenza è il prodotto $V \times I$. Purtroppo il calore dissipato da un conduttore è proporzionale al quadrato della intensità di corrente ($R \times I^2$) ergo è più conveniente alzare la tensione per elevare l'energia trasportata. È per questo che le linee (elettrodotti) che distribuiscono corrente elettrica sul territorio utilizzano tensioni molto elevate (ordine di grandezza delle centinaia di migliaia di Volt), la tensione viene poi ridotta nelle stazioni di trasformazione man mano che si "avvicina" all'utente finale [3].

3.1.5 Segnali

L'elettricità ha però un importante funzione: quella di trasportare/trasmettere **informazione** a grande distanza e velocemente.

Prima dell'avvento dell'elettricità l'unico modo per trasmettere informazione a distanza era quello di trasportare **atomi** in

[3]Interessante, significativa e densa di valore storico la diatriba [39] avvenuta alla fine del '800 fra i fautori della corrente continua (in primis Thomas Alva Edison[38] con la sua società) e di quella alternata (George Westinghouse - fondatore della omonima società - e Nikola Tesla[53]) che alla fine vide vincere (per fortuna!) la seconda.

giro per il mondo: ad es. un libro di carta/pergamena recava con sè informazione sotto forma dei caratteri stampati con inchiostro chimico che un lettore umano doveva interpretare per carpirne il contenuto e farlo proprio. La velocità di trasmissione era limitata dalla velocità dei mezzi di trasporto: cavalli, velieri e persone a piedi. Nella migliore delle ipotesi stiamo pensando a qualche decina di km/h. Anche quando verso la fine del 1700 si cominciano a vedere i primi treni a vapore la situazione non cambia di molto, si guadagna giusto qualche km/h. Bisogna trovare un *carrier*, un trasportatore, più veloce. E, no, i piccioni non vanno benissimo: trasportano troppo poco carico e non sempre arrivano a destinazione[71].

Una prima idea interessante di fine XVIII secolo è il telegrafo **ottico** Chappe: un sistema di segnalamento basato su "pale" (gli apparati sembravano dei mulini a vento con due bracci) che trasmettevano simboli secondo un codice convenzionale variandone la configurazione, i messaggi venivano letti a grande distanza mediante cannocchiali. Il meccanismo è analogo alle segnalazioni che si vedono fare sulle portaerei dagli "omini con le bandiere" agli aerei che stanno decollando o agli elicotteri che stanno atterrando: ogni posizione della braccia corrisponde ad una semantica nota da entrambi i partecipanti alla conversazione. In questo caso la velocità di trasmissione del singolo simbolo (una configurazione delle pale) era pressoché "istantanea" (velocità della luce), ma il problema era rappresentato dalla impraticabilità del meccanismo di configurazione: spostare le pale era un'operazione relativamente lunga (qualche minuto) per cui il numero di simboli trasmessi per unità di tempo rimaneva molto bassa. E col tempo cattivo la comunicazione era interrotta.

La vera svolta si ha nel 1837 (collochiamo questa data: sono passati solo dieci anni dalla pubblicazione della legge di Ohm) quando il quarantaseienne Samuel Morse, dopo alcuni anni di esperimenti, realizza (e brevetterà, qualche anno dopo) il primo **telegrafo elettrico**. La novità di questa invenzione è l'uso dell'elettricità che permette:

- un'ottima velocità di trasmissione del segnale, certo meno veloce della luce usata dal telegrafo Chappe, ma dato che utilizza dei cavi sospesi o interrati la mancanza di visibilità

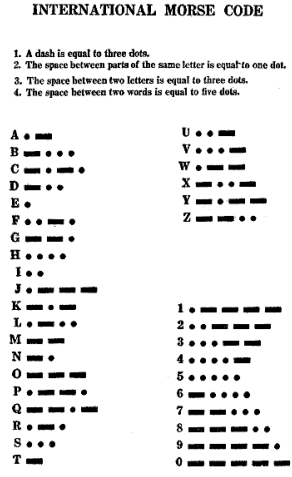


Figura 3.2: Codice Morse (fonte: wikipedia)

"ottica" non influenza la comunicazione;

- il cavo segue i rilievi del terreno, ergo può portare il segnale anche al di là di una montagna o, come accade oggi, al di là di un oceano (curvatura terrestre);
- il meccanismo di invio del segnale è semplicissimo, è solo un pulsante elettrico che chiude un circuito (il lato ricevente è un elettromagnete che scatta quando riceve l'impulso), non ci sono "pale" da muovere, si può "digitare" (proprio con un singolo dito!) il proprio messaggio (quasi) alla velocità del pensiero[4];
- la distanza massima alla quale si può portare l'informazione dipende, guarda caso, dalla legge di Ohm, infatti ogni cavo ha una sua resistenza propria[5] e usando materiale a bassa resistività come il rame è possibile fare cavi molto

[4]Oggi il Codice Morse non si utilizza più (ufficialmente) nelle comunicazioni ma quando era ancora in uso estensivo i "*recordmen* del tasto" riuscivano a trasmettere centinaia di caratteri al minuto!

[5]Funzione della *resistività* del materiale, ad esempio il rame ha $\rho = 1.68 \times 10^{-8}(\Omega * m)$ così per calcolare la resistenza di un cavo si può moltiplicare ρ per la lunghezza e dividere per la sezione.

lunghi su cui mandare segnali elettrici con voltaggi relativamente bassi, tali da poter alimentare gli apparecchi telegrafici con delle semplici batterie piombo+acido;
- costruire una linea telegrafica, nella sua forma più semplice, comporta solo la stesura di un cavo, anche a terra se isolato, srotolando una bobina mentre si cammina... i primi esempi di "cablaggi" per telecomunicazioni!

L'informazione trasportata dal telegrafo elettrico era codificata mediante un segnale elettrico che faceva scattare un elettromagnete. Il ricevente doveva "ascoltare" i ticchettii dell'elettromagnete e tradurre la sequenza di impulsi nelle lettere corrispondenti (figura 3.2). Citiamo questo meccanismo perché è interessante notare come questo codice rappresenti il primo esempio d'uso estensivo della cosiddetta PWM (Pulse Width Modulation, si veda più avanti, sezione 3.4), anche se in senso molto lato. I due simboli, "punto" e "linea", vengono codificati per durata dell'impulso, il punto è un breve *click* e rilascio subitaneo, mentre la linea è un impulso più lungo, *click* + pausa + rilascio. Chiaramente non è possibile usare la sola presenza/assenza (acceso/spento, 1 e 0) di segnale per trasmettere informazione perché l'assenza di segnale è la condizione di riposo del sistema (silenzio) e in questo caso l'assenza dell'impulso non sarebbe interpretabile (è "silenzio" o è uno "0"?). Ancora, intuitivamente, il Codice Morse definisce solo **due** (e non di più) simboli primari (punto e linea) che differiscono per durata dell'impulso perché un umano non sarebbe in grado di distinguere facilmente e senza errori durate diverse che potrebbero corrispondere a più simboli. Un circuito elettrico o un computer invece non avrebbero grossi problemi a fare misurazioni di precisione, sebbene a costo di complicazioni circuitali e di bassa protezione contro il "rumore". Il telegrafo elettrico è la prima applicazione pratica dell'uso della corrente elettrica per la trasmissione di segnali a lunga distanza e ad "alta" velocità. È la prima volta che vengono trasmessi "**bit**" (sotto forma di elettroni lungo un conduttore) invece che atomi.

Questa parentesi storica ci serve anche come scusa per introdurre il concetto di "forma d'onda", un modo grafico di rappresentare lo stato della tensione o della corrente elettrica in un

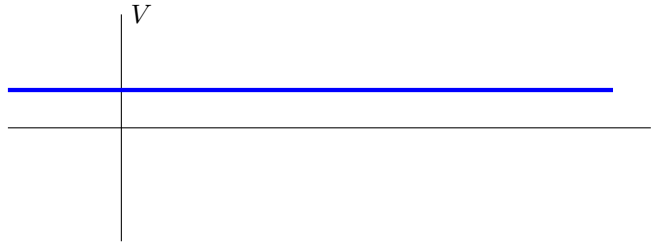


Figura 3.3: Tensione continua

punto/ramo di un circuito nel tempo.

3.2 Forme d'onda

In elettronica si parla spesso di "forme d'onda" per rappresentare segnali. Una forma d'onda è la rappresentazione cartesiana dell'andamento della grandezza elettrica (tensione o corrente) nel tempo. Una forma d'onda ci dice in modo estremamente visuale come varia nel tempo la [tensione in un certo punto] o la [corrente in un certo ramo] di un circuito. Normalmente i segnali vengono definiti matematicamente, scrivendone la funzione che ne modella l'andamento, ma attraverso un grafico di forma d'onda si coglie al volo (a colpo d'occhio) il tipo di segnale che si sta trattando e/o il comportamento di un determinato circuito che crea o influenza i segnali in gioco.

Per cominciare con un esempio banale vediamo (in figura 3.3) l'aspetto di una corrente continua come potrebbe uscire da un alimentatore cosiddetto "stabilizzato" (la cui tensione viene mantenuta costante, appunto): il valore della V non varia nel tempo.

Invece la figura 3.4 esprime un'**ipotetica**[6] situazione di "scarica" (ad esempio pile, condensatori) in cui la tensione cala nel tempo fino a raggiungere lo 0.

[6]Idealizzata! In realtà è ben lungi dall'essere lineare.

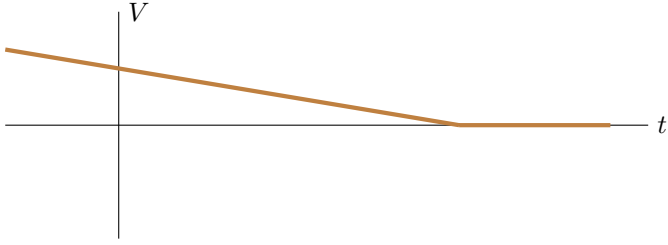


Figura 3.4: Tensione di scarica (ideale!)

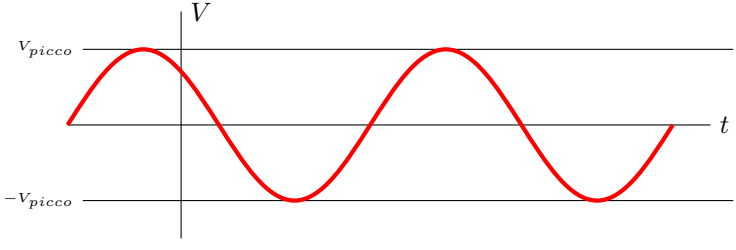


Figura 3.5: Tensione alternata

Figura 3.6: Codice Morse, la lettera “A”

Per finire, la figura 3.5 esprime una tensione alternata, come potrebbe essere quella della rete domestica, in cui la tensione varia continuamente tra positivo e negativo.

Nel caso di forme d’onda “periodiche” (sinusoidali, a impulsi, eccetera) regolari si esprime anche una “frequenza”, misurata in Hz (Hertz, cicli al secondo) e multipli, cioè l’informazione che ci dice il periodo tipico di variazione del segnale ciclico. Ad esempio, un segnale sinusoidale a 50 Hz (la rete elettrica domestica) è quello il cui ciclo completo (da picco a picco) si svolge 50 volte al secondo.

È anche importante ricordare che un segnale sinusoidale (ma più in generale qualunque segnale variabile nel tempo) trasporta una “potenza” elettrica inferiore a quello ipotetico relativo all’ampiezza massima. Infatti, intuitivamente: il segnale varia tra un minimo e un massimo potenziale e la potenza è funzione del potenziale (cfr. formula 3.2), ergo la potenza cosiddetta “efficace” è una “integrazione” della potenza istantanea nel tempo. Senza entrare nei dettagli matematici (integrali in dt), per un segnale perfettamente sinusoidale, la “tensione efficace” si calcola con la formula 3.6:

$$V_{eff} = \frac{V_{picco}}{\sqrt{2}} \tag{3.6}$$

L’abilità di leggere una forma d’onda ci permette di capire come funzionano i componenti elettrici/elettronici e anche di interpretare il funzionamento dei protocolli di comunicazione (al livello elettrico ovviamente). Ad esempio, la forma d’onda generata da un interruttore (o dal pulsante di un telegrafo elettrico!) potrebbe essere quella mostrata in figura 3.6.

3.3 Componenti di base

Questa sezione è dedicata ad una carrellata sui componenti "di base", quelli legati più all'elettricità/elettrotecnica che all'elettronica[7], componentistica cosiddetta "passiva" (resistenze, condensatori, ecc.) e componentistica elettromeccanica (relè, ecc. perché nella terminologia corrente i componenti "attivi" sono i semiconduttori) senza sconfinare troppo nei semiconduttori che saranno brevemente descritti nella sezione 3.5.

Negli schemi[8] elettrici/elettronici ogni componente ha un suo simbolo. Esistono diversi standard per la simbologia dei componenti, ma per fortuna le differenze non sono notevoli e conoscendo un qualsiasi standard è possibile comprendere comunque uno schema con minimo sforzo[11].

3.3.1 Interruttori

Simbolo:

Un interruttore, come dice la parola stessa, interrompe il flusso di corrente a comando. E' un componente meccanico tipicamente azionato a mano. Si può trovare negli stati *chiuso* o *aperto*. Essendo gli stati riferiti al circuito che il componente interrompe, essi indicano rispettivamente passaggio (*circuito chiuso*) e non-passaggio (*circuito aperto*) di corrente. Quelli dotati di una molla di ritorno prendono il nome di pulsanti e vengono costruiti in due versioni: "normalmente aperto" (la corrente passa quando si preme il pulsante) e "normalmente chiuso" (viceversa).

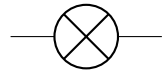
[7] La differenza fra elettrotecnica ed elettronica è che la prima studia "grossolanamente" il comportamento della corrente nei conduttori, mentre la seconda studia il comportamento di dettaglio degli elettroni nel vuoto o nella materia. Ai fini di questo testo si può utilizzare come confine tra una e l'altra il momento in cui si introducono i semiconduttori.

[8] Gli schemi elettrici di questo capitolo sono generati tramite il pacchetto `circuitikz` disponibile in LaTeX.

3.3.2 Lampadine

Difficile dire qualcosa di originale sulla lampadina... Inventata da Thomas Alva Edison[9], era composta da un filamento protetto da un bulbo di vetro in cui veniva fatto il vuoto. Al passaggio della corrente il filamento diventava incandescente producendo luce. Oggi le nuove normative ecologiche prevedono la progressiva sostituzione delle "lampade a incandescenza" (che in effetti hanno una bassa efficienza generando molto calore disperso oltre alla luce) con alternative "fredde" come i LED. In elettronica si usa(va)no come "spie" di stato dei circuiti più che per illuminare.

Simbolo:



3.3.3 Relè

Un relè è un pulsante comandato da un elettromagnete[10]. L'apertura/chiusura del circuito è quindi comandata non a mano bensì attraverso un segnale elettrico. Viene tipicamente usato per comandare un flusso elettrico "importante" (in corrente e/o tensione) usando un altro flusso elettrico più "maneggevole" (a bassa tensione/corrente). Inoltre garantisce l'isolamento elettrico tra il circuito di controllo (la bobina) e i contatti di uscita.

Piccola nota di uso pratico: i contatti elettrici di un relè sono di solito trattati (ad es. dorati) per allungarne la vita utile. Infatti ogni volta che un relè si apre o chiude scocca una pur minima scintilla[11], tale "scintillazione" rovina il contatto

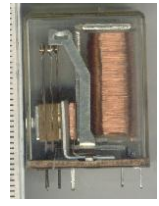
[9] In realtà non sono tutti d'accordo sul fatto che la lampadina sia di Edison, e molti la attribuiscono a Heinrich Göbel

[10] Un magnete **non** permanente, costituito da un nucleo di metallo ferromagnetico avvolto da una bobina di filo elettrico: facendo scorrere corrente nella bobina il nucleo diventa temporaneamente (finché c'è corrente) una calamita.

[11] L'entità dipende da tensione e corrente in gioco.

consumandolo e/o depositandovi residui che a lungo andare lo rendono non più conduttore.

Simbolo di un relè interruttore, di un relè deviatore e aspetto fisico (fonte: wikipedia):



3.3.4 Resistenze

Una resistenza (formalmente "resistore") resiste al passaggio di corrente, appunto. Se attraversata da un flusso di corrente provoca una caduta di tensione e una perdita di potenza che viene dissipata in calore. Nelle applicazioni domestiche viene utilizzata ad esempio proprio al puro scopo di generare calore: molti elettrodomestici (lavatrici, lavastoviglie, asciugacapelli, stufette, ecc.) devono scaldare fluidi (aria, acqua) per essere efficaci, hanno bisogno di calore, le resistenze vengono in questi casi dimensionate con la sola idea di dissipare più calore possibile per cederlo al fluido di lavoro. Un asciugacapelli o una stufetta non sono altro che resistenze poste davanti ad una ventola che asporta il calore generato per dirigerlo verso i capelli o l'ambiente da riscaldare.

Simbolo:

Nei circuiti elettronici invece, tipicamente le resistenze vengono utilizzate:

- per la loro caratteristica caduta di tensione, infatti ai capi di una resistenza in cui passa corrente si verifica una caduta di tensione calcolabile mediante la legge di Ohm $V = R \times I$; un caso generale di utilizzo della caduta di

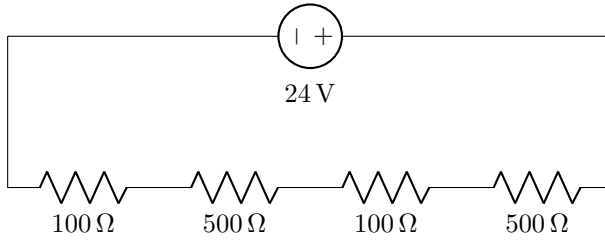


Figura 3.7: Esempio di partitore

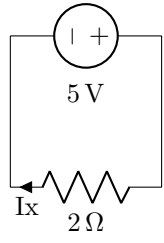


Figura 3.8: Esempio di circuito di cui calcolare corrente

tensione su una resistenza è il "**partitore di tensione**" (figura 3.7) in cui ad esempio una tensione di alimentazione può essere "divisa" (partizionata, appunto) in varie "sotto-tensioni" (in figura qui sotto un esempio, le tensioni ai capi delle resistenze da 100Ω sono di 2V mentre ai capi di quelle de 500Ω sono di 10V, infatti 2+10+2+10=24);

- per limitare la corrente massima applicata ad un circuito/componente data una certa tensione di alimentazione, usando sempre la legge di Ohm $I = V/R$.

Ad esempio, per calcolare/dimensionare un valore massimo di corrente applicato ad un circuito si deve ricorrere alla legge di Ohm, ad esempio, dato un circuito semplice come quello mostrato in figura 3.8. Applicando la formula 3.1 è possibile calcolare la corrente massima che passa in "Ix", infatti $5V = 2\Omega \times I?A$ che diventa $I? = 2.5A$ (ampere). La potenza dissipata (formula

Figura 3.9: Resistenze per elettronica di bassa potenza (fonte: wikipedia)

3.2) è $P = R \times I?^2$ cioè $P = 2\Omega \times 2.5^2 A = 2 \times 6.25 = 12.5$ (Watt).

Le caratteristiche di resistenza dei componenti usati in elettronica, essendo tipicamente di piccole dimensioni (quelli grandi hanno spazio per lettere e numeri), utilizzano un codice colore da cui è possibile capire il valore in Ω (e multipli) e la percentuale di tolleranza sul valore stesso. Il codice colore è formato da anelli colorati dipinti sul componente stesso (figura 3.10). I valori di resistenza si calcolano applicando la tabella in figura 3.11

3.3.5 Condensatori

Il condensatore (*capacitor* in inglese) è un componente che permette l'accumulo temporaneo di elettroni. Idealmente è composto da due piastre metalliche (conduttive) ampie e ravvicinate (da cui il simbolo), spesso separate da un "dielettrico" (un materiale più isolante della semplice aria, ne aumenta la capacità). È possibile "caricarlo" applicando una differenza di potenziale ai suoi capi, spostando quindi elettroni da una piastra all'altra. Questi elettroni rimangono asimmetricamente distribuiti anche quando si toglie alimentazione. Un buon condensatore senza perdite riesce a mantenere la propria carica anche per un tempo molto lungo, è per questo che in caso di operazioni su

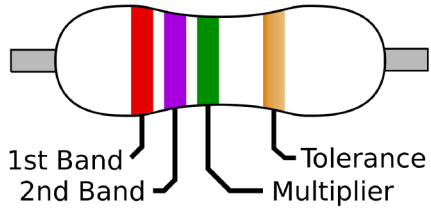


Figura 3.10: Bande colorate su resistenza (fonte: wikipedia)

Colore		Anello 1 Cifra 1	Anello 2 Cifra 2	Anello 3 Moltiplicatore	Anello 4 Tolleranza
nessuno	xxx	-	-	-	
argento		-	-	0.01	
oro		-	-	0.1	
nero		-		1	-
marrone		1	1	10	
rosso		2	2	100	
arancio		3	3	1000	-
giallo		4	4	10000	-
verde		5	5	100000	
blu		6	6	1000000	
viola		7	7	10000000	
grigio		8	8	100000000	
bianco		9	9	1000000000	-
Selection:		1	2	10	±1%

Figura 3.11: Tabella colori resistenze

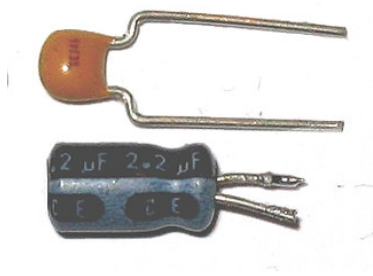
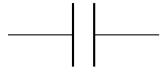


Figura 3.12: Cond. ceramico (sopra) ed elettrolitico (fonte: wikipedia)

circuiti ad alta tensione si consiglia di attendere molto tempo dopo aver tolto l'alimentazione per evitare il rischio di scosse.

Simbolo:



Gli elettroni accumulati su una delle due piastre creano una differenza di potenziale che si può utilizzare applicando un carico ai capi del condensatore.

Un condensatore si può applicare a valle di un segnale sinusoidale (figura 3.5) o, meglio, di un segnale "pulsante" (sinusoidale con tutti i lobi dallo stesso lato, ad esempio tutti i negativi ribaltati sul positivo) per "appiattirlo" (*spike removing* in inglese), per renderlo cioè meno variabile nel tempo. Infatti il condensatore si carica quando il segnale sale di tensione e si scarica (sommando la sua tensione a quella del segnale in ingresso) durante la discesa, funziona quindi da "accumulatore inerziale".

La vera utilità "elettronica" del condensatore è nei circuiti RC (Resistenza-Condensatore), opportunamente collegati (cfr. sezione 3.3.8) permettono la "gestione" di segnali variabili nel tempo. Intuitivamente: un condensatore si scarica su un carico resistivo con una velocità che decresce nel tempo perché la

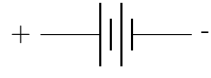
tensione, man mano che gli elettroni fluiscono, scende (sempre legge di Ohm).

Nota bene: normalmente i condensatori non hanno polarità, cioè è il loro comportamento è indifferente all'inversione fisica del componente (come le resistenze) in un circuito, salvo quelli cosiddetti "elettrolitici" che per la loro particolarità costruttiva (il dielettrico è tale solo in un solo verso) vanno montati seguendo l'indicazione "+" e "-" sull'etichettatura, pena il rischio di esplosione.

3.3.6 Pile, accumulatori, generatori, ...

Quasi ogni tipo di circuito necessita di un generatore di energia elettrica, dove non si vede è perché non è esplicito (come in una radio a galena[50]).

Simbolo pila:



Nell'elettronica embedded si utilizzano di solito tensioni relativamente basse (ad es. 5, 12, 24 V) che vengono fornite tramite:

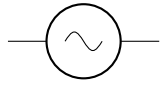
- batterie ricaricabili (al piombo+acido sigillate, ioni di litio, nichel-metallo idruro, ecc.);
- pile (zinco-carbone, litio, alcaline, ecc.);
- alimentatori "stabilizzati" (convertono la tensione di rete, 220 V≂, in una più bassa continua, ad es. 5 V=).

Nella progettazione dei circuiti bisogna dimensionare correttamente la sezione di alimentazione altrimenti, ovviamente, il circuito non funzionerà o, peggio, funzionerà erraticamente.

Le caratteristiche principali di una fonte di alimentazione sono la tensione (V, continua/alternata) e la corrente massima fornita (sottomultipli di A, ad es. mA, milli-Ampere $= 1/1000A$). Nel caso delle pile/batterie è bene conoscere anche la capacità di carica (mA/h milli-Ampere ora, ad es. $500mA/h$ significa che l'erogatore può fornire **teoricamente** $500mA$ per un'ora, $250mA$ per due ore e così via), la corrente massima di scarica

(di solito espressa sulle batterie per automobile) e in certi casi anche la densità di energia (misurata in $W \times h/kg$ - peso - e $W \times h/L$ - volume) se si vogliono pianificare anche ingombri e pesi, si pensi ad esempio ai droni.

Simbolo generatore a corrente alternata:

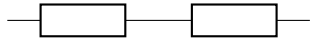


3.3.7 Serie e parallelo

"In serie" e "in parallelo" sono modi di collegare due o più componenti tra loro in modo che le proprietà dell'insieme siano una combinazione delle proprietà dei singoli componenti. Citeremo solo i casi più semplici, cioè:

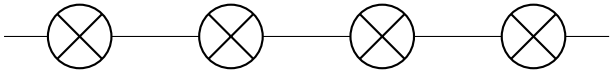
- coppia di resistenze;
- coppia di condensatori;
- coppia resistenza-condensatore.

Il collegamento in **serie** è quello in cui i componenti sono "accodati" uno con l'altro:



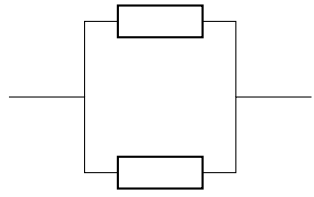
In questo caso il flusso della corrente passa prima in un componente e poi nell'altro. La stessa **corrente** attraversa ogni "bipolo" della serie (cfr. sezione 3.1.3).

Esempio classico è quello delle lampadine di Natale in serie:



In cui ogni lampadina può (e capiremo tra poco il perché) essere prevista per una tensione molto inferiore rispetto all'alimentazione del circuito intero.

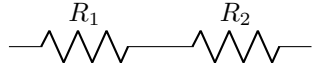
Mentre il collegamento in **parallelo** è quello in cui ogni componente è "affiancato" all'altro:



E in questo caso il flusso della corrente si partiziona (non necessariamente al 50%) nei due rami. La **tensione** è comune a tutti i "bipoli" in parallelo (cfr. sempre sezione 3.1.3).

R in serie

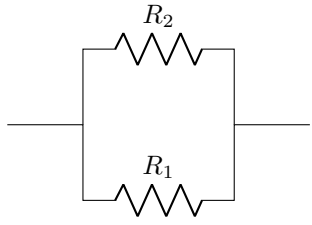
Due (o più) resistenze in serie "sommano" i loro effetti, cioè la resistenza totale del componente "aggregato" è semplicemente la somma delle singole resistenze: $R_{tot} = R_1 + R_2 + ... + R_n$.



Il caso delle lampadine di Natale: ogni lampadina viene "data per una certa tensione di alimentazione" nel senso che in realtà ha una sua resistenza interna e una corrente massima che la può attraversare (altrimenti si brucia). Dalla legge di Ohm ne deriva la tensione massima possibile. Se ne colleghiamo tante in serie la resistenza si sommerà per cui potremo alimentare il tutto con una tensione maggiore senza far salire la corrente, quindi senza bruciare le lampadine. Ad esempio, avessimo lampadine da $12V$ (da auto), potremmo tranquillamente collegarne 20 in serie e alimentare il tutto a $220V$ senza timore (se non quello di stare attenti alle scosse!).

R in parallelo

Nel caso del collegamento in parallelo di resistenze il flusso di corrente si distribuisce proporzionalmente al valore delle resistenze stesse secondo la formula: $1/R_{tot} = 1/R_1 + 1/R_2 + ... + 1/R_n$.



Il caso particolare della coppia di resistenze in parallelo si riduce alla formula:

$$\frac{1}{R_{tot}} = \frac{1}{R_1} + \frac{1}{R_2} = \frac{R_1 + R_2}{R_1 \times R_2} \tag{3.7}$$

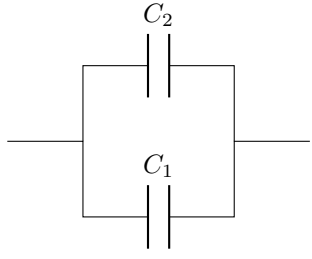
che quindi diventa:

$$R_{tot} = \frac{R_1 \times R_2}{R_1 + R_2} \tag{3.8}$$

C in parallelo

Il mondo fisico è spesso elegante: i condensatori si comportano "inversamente" alle resistenze quando si tratta di collegarli in serie/parallelo.

Due (o più) condensatori in **parallelo "sommano"** i loro effetti, cioè la capacità totale del componente "aggregato" è semplicemente la somma delle singole capacità: $C_{tot} = C_1 + C_2 + ... + C_n$.



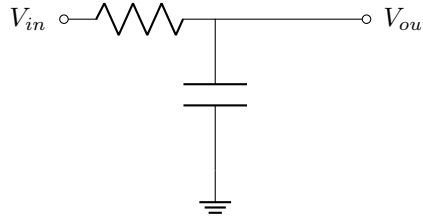
C in serie

La capacità totale di due o più condensatori in serie si calcola **come il parallelo delle resistenze**: $1/C_{tot} = 1/C_1 + 1/C_2 + ... + 1/C_n$.

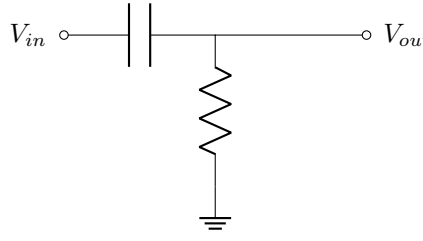
3.3.8 Circuiti RC

I cosiddetti "Circuiti RC" (Resistenza-Condensatore), nella loro forma più semplice sono composti da una sola resistenza e da un solo condensatore accoppiati in serie/parallelo, in due combinazioni:

1. "passa-basso" o "integratore"



2. "passa-alto" o "differenziatore"



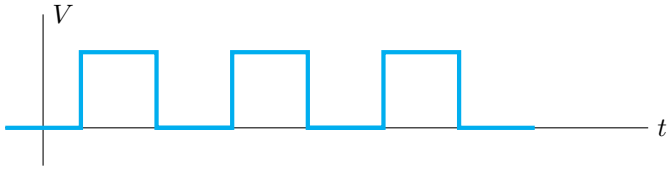


Figura 3.13: Esempio di onda quadra

Se all'ingresso vengono iniettati segnali sinusoidali, all'uscita si avranno segnali con la stessa forma d'onda (anche se non in fase) e con diversi gradi di attenuazione in funzione della frequenza, da cui i nomi di "passa alto" (fa passare, cioè non attenua, i segnali ad alta frequenza) o "passa basso" (viceversa).

Quando invece in ingresso (V_{in}) viene applicata una tensione variabile nel tempo di forma quadra (figura 3.13), cioè una tensione che varia solo tra zero e un livello fisso di tensione, in uscita dai due circuiti sopra descritti si otterranno forme d'onda molto diverse dall'originale. In questi casi i nomi dei circuiti RC presentati diventano "integratore" e "differenziatore".

Nel caso dell'integratore il segnale di uscita assumerà la forma del "dente di sega" (vedere figura 3.14), mentre nel caso del differenziatore la forma dell'onda di uscita sarà un "treno di impulsi" (figura 3.15).

L'integratore, intuitivamente, funziona nel seguente modo:

- il condensatore richiede del tempo per caricarsi, in particolare richiede un tempo $\tau = R \times C$ (chiamata **costante di tempo del circuito RC**) per arrivare al 63.2% della sua capacità, un altro τ per il successivo 63.2% del rimanente 36.8% e così via per avvicinamenti successivi fino al "limite" del 100%;
- ecco il motivo della salita "dolce" (figura 3.14) della tensione in uscita rispetto a quella in ingresso;
- se il ciclo dell'onda quadra in ingresso è molto breve il condensatore non fa mai in tempo a raggiungere la carica massima;
- cioè l'ampiezza del segnale in uscita non arriva mai alla stessa ampiezza del segnale in ingresso;

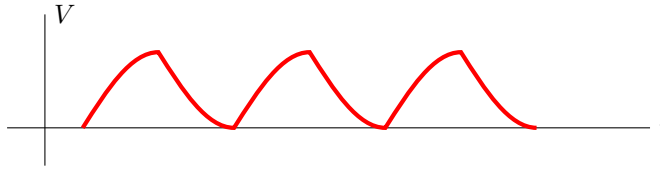


Figura 3.14: Esempio di onda a “dente di sega”

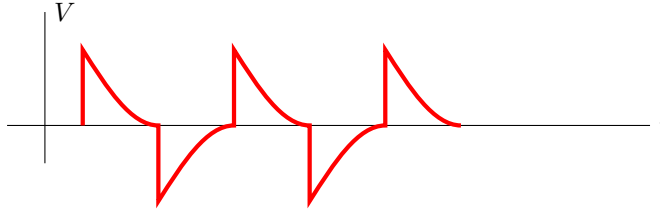


Figura 3.15: Esempio “treno di impulsi”

- questa situazione peggiora al salire della frequenza.

Il differenziatore, intuitivamente, funziona nel seguente modo:

- durante la fase alta della quadra il condensatore si carica;
- ai capi della resistenza si forma un potenziale negativo che man mano che il condensatore si carica tende a 0 (perché a condensatore carico non c’è più passaggio si corrente);
- durante la fase 0 dell’onda quadra in ingresso;
- il condensatore si scarica sulla resistenza (invertendo il verso di scorrimento degli elettroni);
- generando una tensione positiva decrescente nel tempo ai capi di R;
- ecco il motivo della discesa relativamente dolce (figura 3.15) della tensione in uscita;
- se il ciclo dell’onda quadra in ingresso è molto lungo gli “impulsi” sono molto sottili e distanti tra loro;
- questa situazione peggiora al decrescere della frequenza.

3.4 PWM (Pulse Width Modulation)

Dopo aver introdotto alcuni componenti di base con il loro effetto sulla corrente riprendiamo il concetto di forma d'onda (sezione 3.2) e dedichiamo un piccolo approfondimento ad un tipo di segnale usato molto spesso nel campo *embedded*: la Pulse Width Modulation, o modulazione della larghezza dell'impulso.

Un segnale PWM semplice ha una forma d'onda "quadra" (figura 3.13), tipicamente con un periodo fisso, ma non simmetrica, cioè la durata della semi-onda alta ("1" logico, 5 V TTL) è raramente uguale alla durata della semi-onda bassa ("0" logico, 0V TTL). La modulazione (variazione) del segnale viene realizzata allungando o accorciando la durata (tra 0 e il periodo) dell'impulso alto. La durata dell'impulso viene misurata in percentuale, si definisce *duty cycle* e specifica la durata rispetto al periodo, ad esempio 50% definisce una PWM quadra simmetrica, 10% una quadra "accesa" per il 1/10 del periodo, e così via.

La PWM è un tipo di segnale che viene usato sia per trasmettere informazione che per pilotare efficientemente (in modo modulato) carichi "attivi", ad esempio motori elettrici previsti per la corrente continua.

Nel caso della trasmissione dei dati il principio è lo stesso della telegrafia Morse (sezione 3.1.5): la durata dell'impulso porta semantica, mediante circuiteria di precisione è possibile misurare tale durata e associarvi informazione[12].

Nel caso del pilotaggio dei carichi la PWM sfrutta la caratteristica inerzia (tendenza a mantenere lo stato) di alcuni carichi (motori, lampadine a filamento) per ottenere un effetto di "regolazione continua". Si pensi ad un motore elettrico, una volta alimentato (avviato) e a regime, quando si stacca l'alimentazione esso continua a girare per inerzia, sempre più lentamente fino a fermarsi, impiegando un tempo variabile funzione del carico fisico che sta muovendo. Se proviamo ad accendere e

[12]Esempio: invece che contare gli impulsi e quindi dover attendere n periodi per raccogliere n bit si potrebbe codificare una PWM a 9 step/simboli (10%,20%,30%,40%,50%,60%,70%,80%,90% *duty cycle*) trasmettendo così una cifra decimale per impulso. Ovviamente la robustezza ai disturbi peggiora e la circuiteria è più complessa.

spegnere velocemente e continuamente l'interruttore del motore (dobbiamo essere molto regolari nella procedura) osserveremo che il motore continua a girare, ma ad una velocità costantemente più bassa. Stiamo in effetti realizzando una *"poor man's PWM"* (PWM dei poveri) a mano. Nella realtà dei circuiti *driver* dei motori (o nei *dimmer* delle lampade a incandescenza) troviamo un generatore di PWM di cui si può cambiare (di solito mediante un "potenziometro": una resistenza variabile) il *duty cycle*.

Ricordando il concetto di V_{eff} (la formula era 3.6, ma per la PWM è sufficiente moltiplicare la tensione massima per la durata del *duty cycle* in percentuale) si può affermare che variando la durata dell'impulso varia la tensione efficace (e quindi anche la potenza efficace erogata) disponibile a valle. Un circuito PWM, rispetto ad un circuito che varii la tensione in maniera lineare, è costruttivamente più semplice e più efficace (disperde meno potenza sotto forma di calore).

L'effetto della PWM su un carico è riassunto graficamente e **qualitativamente**[13] (!!!) mediante le forme d'onda delle figure 3.16, 3.17, 3.18 e 3.19, in blu spesso la forma d'onda PWM generata dal driver, in rosso sottile la forma d'onda in presenza di un eventuale condensatore, in verde (linea orizzontale) la V_{eff} equivalente.

3.5 Semiconduttori

I semiconduttori sono componenti fondamentali per tutto il mondo elettronico: senza di essi saremmo ancora alle calcolatrici meccaniche (figura 3.20), a quelle elettromeccaniche o, al meglio, ai computer a valvole termoioniche.

I componenti "semiconduttori" prendono il nome dai cristalli che li contengono, tipicamente a base silicio, opportunamente "drogati" (combinati con Boro, Fosforo e altri elementi) si comportano, appunto, come semi-conduttori: conducono più o

[13] Le forme d'onda reali dipendono dal tipo di carico (che può non essere puramente resistivo). Molti motori in corrente continua sono collegati con un condensatore in parallelo per addolcire la forma d'onda in ingresso, per ridurre lo stress meccanico e i disturbi elettromagnetici.

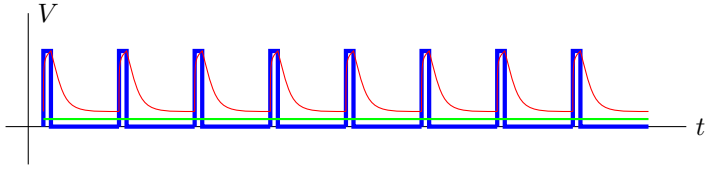


Figura 3.16: PWM al 10%

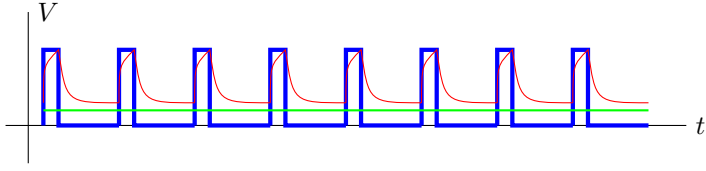


Figura 3.17: PWM al 20%

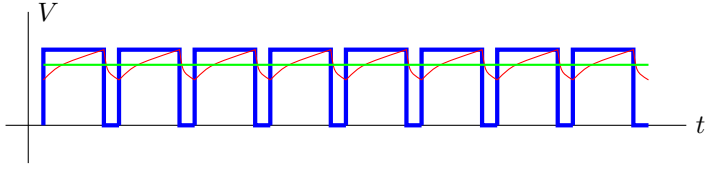


Figura 3.18: PWM al 80%

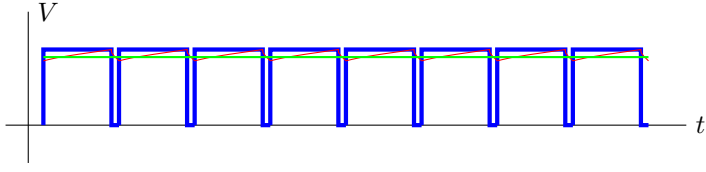


Figura 3.19: PWM al 90%

Figura 3.20: Calcolatrice meccanica tascabile CURTA

meno corrente in funzione di varie condizioni al contorno, ad es. temperatura, luce, altre correnti "vicine", eccetera.

Il silicio viene utilizzato sotto forma di cristalli. A seconda del tipo di materiale con cui viene drogato, in questi cristalli il numero di elettroni è superiore o inferiore al "naturale" (il numero esatto per bilanciare i protoni nei nuclei). I cristalli con elettroni in più vengono denominati "N" (***N**egative*) dato che gli elettroni hanno carica negativa, mentre quelli con elettroni in meno sono denominati "P" (***P**ositive*) poiché in questo caso vince la carica positiva dei protoni non bilanciata da un ugual numero di elettroni. Accoppiando opportunamente due **o più** cristalli "N" e cristalli "P" vengono create le cosiddette "giunzioni" che vanno a formare i componenti semiconduttori, ad esempio un transistor che è composto da due giunzioni a loro volta giunte: PNP o NPN.

3.5.1 Diodi e LED

I **diodi** sono i semiconduttori più semplici, sono composti da una singola giunzione PN, la loro caratteristica è quella di far fluire la corrente in una sola direzione.

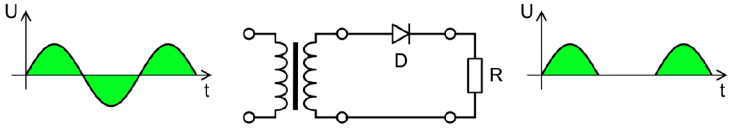
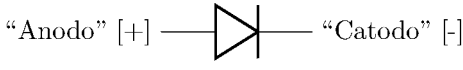


Figura 3.21: Rettifica di una tensione alternata (fonte: Wdwd)

Simbolo diodo:



Applicando una tensione positiva all'anodo[14] la corrente scorre, viceversa il diodo si comporta come un interruttore aperto... almeno fino al raggiungimento di una tensione troppo elevata (di *breakdown*, rottura) che "si fa strada" comunque.

Questa caratteristica è utilissima nei cosiddetti "rettificatori" (figura 3.21) circuiti che sfruttando l'unidirezionalità del diodo (o di più diodi combinati, eventualmente con altri componenti come i condensatori già citati) prendono in ingresso una tensione alternata e rilasciano all'uscita una tensione pulsante o quasi continua. Il "quasi" è d'obbligo perché è sempre molto difficile eliminare tutte le tracce della forma d'onda originale, il cosiddetto *ripple* residuo.

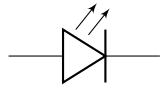
Esistono molti tipi speciali di diodo (Zener, Tunnel, ecc.), ma citeremo qui solo il più noto alla grande massa, il LED (*Light E-mitting D-iode*). Esso è forse l'unico semiconduttore che non viene usato per la sua caratteristica principale (condurre corrente in una sola direzione), ma bensì per la sua caratteristica secondaria: **emettere luce**. Tutti i diodi in realtà emettono luce (che non si vede perché è poco intensa e perché il corpo del componente non è trasparente), ma i LED vengono drogati appositamente per esaltare questo effetto collaterale. Si usano i LED come "spie" di funzionamento degli apparecchi. I LED

[14] Il termine "anodo" indica il polo negativo se riferito ad una pila, il polo positivo se riferito ad un componente. Per ricordarselo: l'anodo è il polo da dove fluiscono gli elettroni in uscita.

Figura 3.22: Pannello frontale di uno dei primi home computer, l'Altair 8800, pieno di LED (fonte: Todd Dailey)

infatti, hanno due enormi vantaggi rispetto alle lampadine a filamento: consumano pochissimo (almeno un ordine di grandezza) e durano moltissimo (svariati ordini di grandezza).

Simbolo LED:

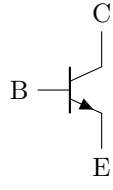


Oggigiorno la tecnologia dei LED è molto avanzata e da qualche anno è possibile usarli anche per l'illuminotecnica dato che sono stati prodotti LED di potenza equivalenti (ma più ecologici e duraturi) alle lampadine a incandescenza. Inoltre la facilità con cui si può emettere luce colorata ha permesso la realizzazione di lampade "multicolore" attraverso i cosiddetti LED RGB (***R**ed, **G**reen, **B**lue*), composti da tre LED separatamente alimentabili, variando la corrente (o, meglio, usando la PWM, vedi sezione 3.4) sulle tre componenti colore si creano tonalità a piacere.

3.5.2 Transistor

I **transistor**[15] "classici"[16] sono semiconduttori a due giunzioni, quindi esistono di tipo PNP e NPN, ai fini di questo testo ha poco senso distinguerli dato che ci interessa capire i principi di funzionamento. Non entreremo nel dettaglio del "calcolo della polarizzazione"[51], i.e., calcolare le resistenze con cui complementare un particolare modello di transistor per farlo "lavorare" correttamente.

Simbolo, versione NPN:



Le lettere marcano la **B**ase, l'**E**mettitore e il **C**ollettore.

Il transistor si comporta da "rubinetto" della corrente: la corrente che fluisce tra Emettitore (di elettroni) e Collettore (raccoglie ciò che arriva dall'Emettitore) **è funzione della corrente** che scorre tra Emettitore e Base. È importante sapere che la corrente di Base (quella che fluisce tra E e B) è inferiore a quella che scorre tra E e C. Cioè si realizza una situazione in cui **una piccola corrente influenza una grande corrente**. Il rapporto fra le due correnti è il fattore di amplificazione del transistor, simbolo β o hFE[17]. Se la corrente di "pilotaggio" è[18] un segnale **debole** variabile nel tempo, in uscita sul Collettore si otterrà una corrente di forma analoga a quella di ingresso (a

[15] Inventati nel 1947 da John Bardeen, Walter Brattain e William Shockley (vinsero poi il Nobel nel 1956).

[16] Qui tratteremo solo i cosiddetti transistor bipolari, eviteremo di dettagliare il ventaglio dei semiconduttori della famiglia dei transistor: FET, MOSFET, UJT, ...

[17] La sigla deriva dalla contrazione di "**Hybrid** parameter **forward** current gain, common **emitter**"

[18] Stiamo super semplificando, in realtà il segnale da amplificare va sommato ad un segnale di "base" continuo che mantiene il transistor in "area di lavoro".

meno di distorsioni se l'amplificazione non è lineare), ma con una intensità molto maggiore.

In elettronica **digitale** i transistor si usano in maniera "booleana" (on/off), cioè similmente a dei relè a stato solido, col vantaggio di non avere parti meccaniche in movimento, di lavorare con tensioni e correnti più basse e con dimensioni di gran lunga minori (ordine di grandezza 10 *nm* dentro i chip integrati, vedi nel seguito).

Ultimo dettaglio molto importante e interessante: i transistor tradizionali lavorano "in corrente", cioè utilizzano un flusso di corrente (pur piccolo, ma non nullo) per pilotare la corrente di uscita, ma esiste anche una famiglia di transistor che utilizza una tecnologia che lavora "in tensione", i cosiddetti FET (**F**ield **E**ffect **T**ransistor). I FET sfruttano un campo elettrico invece che un flusso di corrente per "strozzare" (i lettori ci passino il termine) il passaggio degli elettroni tra *source* e *drain* (gli analoghi di emettitore e collettore). Il vantaggio primario del FET è l'efficienza termica: non utilizzando corrente di pilotaggio non disperde calore "inutile" contrariamente al transistor tradizionale che assorbe corrente anche a riposo. Il meccanismo del FET è alla base dell'elettronica digitale moderna, cosiddetta "CMOS", che, a partire dagli anni '70, ha permesso una notevole densità di componenti (calore e vicinanza non vanno d'accordo).

3.5.3 Circuiti integrati

Il termine stesso "circuito integrato" ne esplicita l'architettura: si tratta infatti di combinare alcuni[19]) componenti a formare un qualche tipo di circuito... **integrati** (appunto!) direttamente su un unico "pezzettino"[20] di silicio.

Il processo di costruzione di un circuito integrato è essenzialmente chimico: vengono depositati vari strati di silicio e altri elementi secondo una "mappa" che implementa il circui-

[19] Da poche unità fino a milioni... Una nota legge **empirica**, la legge di Moore, afferma che il numero di transistor per unità di superficie raddoppia ogni due anni, recentemente è stata messa in discussione[46, 72], si prevede che i limiti delle tecnologie attuali (la dimensione dell'atomo stesso) verranno raggiunti intorno al 2025.

[20] *chip* in inglese, da cui il termine colloquiale dei circuiti integrati

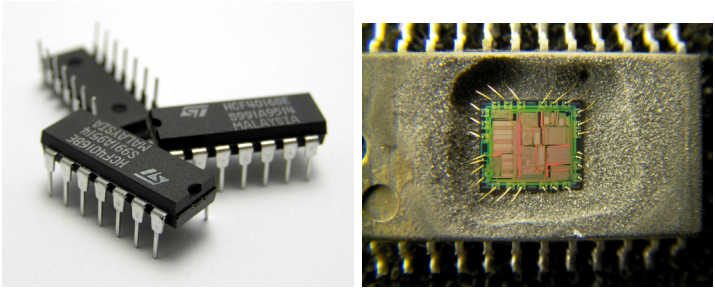


Figura 3.23: Esempio di circuiti integrati, in *packaging* e dettaglio costruttivo (fonte: wikipedia)

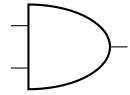
to voluto. I componenti risultano "in piedi", nella terza dimensione: le parti dei componenti sono infatti "impilate" sui vari strati. Il chip finito viene annegato in materiale resinoplastico, i collegamenti elettrici tra il chip stesso e i "piedini" di montaggio/saldatura sono sottili "capelli" d'oro (figura 3.23).

I circuiti integrati si dividono essenzialmente in quelli "analogici" (o "lineari") e in quelli "digitali" (o "logici"). I primi lavorano con segnali a variazione continua: amplificano, generano segnali, regolano, ecc. I secondi invece sono quelli maggiormente usati in informatica, gestiscono segnali "booleani": segnali che variano solo tra due livelli di tensione[21] equivalenti ai bit 0 e 1.

Gli "integrati"[22] digitali sono composizioni di mattoni di base chiamati "porte" (*gate* in inglese) logiche: meccanismi elettrici che calcolano le funzioni logiche classiche come `AND`, `OR`, `XOR`, ecc. Ad esempio si esamini una porta logica `AND` digitale, il cui simbolo è:

[21] Storicamente 0 e 5 Volt, i cosiddetti livelli TTL (**T**ransistor-**T**ransistor) **L**ogic). Lo standard TTL definisce dei range di tensione per identificare lo 0 e l'1 anche in presenza di disturbi e attenuazioni: un segnale tra 0 e 0.8 V viene considerato "0", tra 2 e 5 V viene letto come un "1", mentre nella finestra 0.8-2V il segnale viene ritenuto incerto e non se ne garantisce una corretta interpretazione.

[22] Colloquialmente abbreviato. Oppure "CI" come acronimo.



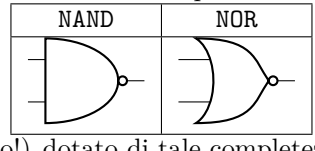
Gli ingressi sono quelli di sinistra, l'uscita è quella di destra. In logica la funzione `AND` è *true* quando entrambi gli ingressi sono *true*. In elettronica digitale l'implementazione si traduce in un circuito che genera un segnale TTL "alto" quando a entrambi i suoi ingressi viene applicato un segnale TTL "alto". In parole povere l'uscita va a 5 V quando a entrambi i suoi ingressi si applicano 5 V. Nell'elettronica digitale si usano molto spesso porte logiche a **tre stati** (*three state*):

1. a massa (circuito chiuso), livello basso, LOW;
2. a livello (es. 5 V del TTL), livello alto, HIGH;
3. circuito aperto, o **alta impedenza**.

Il terzo stato, l'alta impedenza, è quello in cui la porta si comporta come se non esistesse nel circuito, cioè non lo influenza, non impone uno stato sulle porte eventualmente connesse in parallelo. Ciò è molto importante quando si hanno più porte collegate ad un *bus* comune e solo una porta per volta deve trasmettere segnali.

Disegnando mappe di silicio da depositare sui chip è possibile realizzare qualunque combinazione di porte logiche... ma, come spesso accade, le scienze matematiche sono venute in aiuto ai progettisti di circuiti integrati. Infatti:

- applicando i teoremi di De Morgan[27];
- sapendo che esiste il concetto di *completezza funzionale*[59][23];
- sapendo che esistono ben due operatori:



  ognuno (da solo!) dotato di tale completezza.

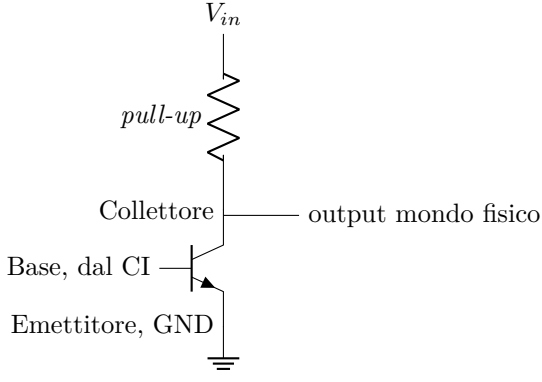
Ne consegue che è possibile realizzare qualunque tipo di circuito integrato digitale usando **unicamente** varie combinazioni

[23] Caratteristica di un sottoinsieme di operatori booleani, il sottoinsieme è funzionalmente completo se, opportunamente combinati i suoi membri, permette l'espressione di tutte le possibili "tabelle di verità".

di un solo tipo di porta (o tutte `NAND` o tutte `NOR`)! Il già citato Mims[48] dettaglia molto bene le varie combinazioni circuitali per implementare le funzioni tipiche per l'elettronica digitale (memorie, conversioni binario/decimale, registri, contatori, ecc.). Nota storica: nel 1971 un italiano, Federico Faggin[25], entrato da poco in Intel, realizzò il primo microprocessore integrato commerciale, il famosissimo (tra gli informatici) 4004.

Open collector e open drain

Per poter far funzionare un CI ad una tensione interna qualsivoglia (magari anche molto bassa per ridurre la potenza dissipata) e mantenere verso l'esterno dei livelli di tensione standard (ad es. i 5 V del TTL) si utilizzano dei semplici circuiti di interfaccia, interni e inclusi nei circuiti integrati, chiamati *open collector* e *open drain*. Applicando, a valle della porta logica, un transistor:



Si può applicare una resistenza cosiddetta di *pull-up* tra Collettore e V_{in} così da ottenere un livello logico "alto" quando il transistor non conduce (vince la tensione portata dalla resistenza) e basso quando il transistor conduce (vince il collegamento a massa).

FPGA

Gli integrati FPGA (Field Programmable Gate Array) sono dei particolari tipi di CI programmabili dall'"utente finale" (inteso come sviluppatore embedded in questo caso). In parole povere, sono CI dotati di un gran numero di porte logiche le cui connessioni sono programmabili[24], potendo definire come connettere le migliaia di porte logiche di una FPGA è possibile specificare tramite un linguaggio di programmazione (tipicamente VHDL[55]) come dovrà comportarsi il CI una volta inserito nel circuito.

Il vantaggio enorme di questi componenti (oltre alla riprogrammabilità) è la velocità di esecuzione delle loro funzioni, tanto che vengono utilizzati nella cattura di dati da sensori ad altissima frequenza[25]. Per contro, lo svantaggio (a causa del quale le FPGA non vengono usate ad esempio per creare delle CPU per i computer normali) principale è la loro bassa efficienza energetica e spaziale (nel senso di area di silicio), anche se recentemente tale *gap* è andato riducendosi.

3.6 Strumenti di misura

In un laboratorio di elettronica si trovano parecchi strumenti di misura, sia **passivi** (ad esempio un voltmetro per misurare una tensione), sia **attivi** (strumenti che producono ad esempio un segnale da *iniettare* in un circuito per vedere come reagisce). Qui ci limiteremo a descrivere quei pochissimi e compatti strumenti indispensabili ad un professionista, cioè il minimo per capire se un sensore/attuatore funziona, se un GPIO produce qualche tensione in uscita, misurare una resistenza senza dover trovare la tabella colori, ecc.

Anche in questo ambito rimarremo minimalisti, una buona panoramica, oltre al già citato Mims[48] si trova al capito-

[24] Anche più volte per i modelli recenti, le prime FPGA uscite sul mercato erano invece da "bruciare" stile PROM (Programmable Read Only Memory) o EPROM (Erasable PROM), cancellabili mediante luce ultravioletta.

[25] Negli acceleratori di particelle ad esempio, dove gli eventi avvengono in tempi brevissimi e le frequenze di campionamento sono pertanto molto alte.

lo “Misure elettriche” di una qualunque edizione del “Manuale dell’Ingegnere”[15], dove si troveranno, ben spiegati, i principi di funzionamento, le teorie sulle misure (errori sistematici, ecc.) e altri esempi applicativi.

Gli strumenti di misura si possono dividere salomonicamente in “analogici” (con lancetta che si muove con continuità su un quadrante graduato) e “digitali” (visualizzano le misure sotto forma di cifre su un display a segmenti o simili). I primi sono soggetti a errore di lettura umano in funzione della posizione dell’occhio rispetto al quadrante.

Uno strumento analogico è tipicamente composto da un ago montato solidalmente su una bobina imperniata su un “cardine”, una molla di recupero e un nucleo magnetico. Il passaggio di corrente nella bobina crea un campo magnetico temporaneo che interagisce con quello del nucleo magnetico spostando così la bobina e l’ago sul quadrante. A riposo la molla tiene l’ago sullo zero.

Considerazione quasi filosofica: ogni strumento di misura applicato ad un sistema sotto osservazione **ne modifica lo stato**. Alcuni strumenti riescono a influenzare poco il sistema, altri devono farlo (ad esempio per misurare la corrente erogabile da una batteria bisogna, anche se per pochi istanti, provare a estrarle una corrente molto elevata utilizzando un carico da 0 Ω). Uno dei fattori di qualità degli strumenti di misura elettronica è proprio la capacità di influenzare il meno possibile il sistema misurato.

3.6.1 “Tester di circuito”

Titolo pomposo per un attrezzo di “fortuna”: una batteria (opzionale) e una lampadina attaccata ad una coppia di fili. Uno strumento semplice per alcuni usi invero molto **grezzi**.

La sola lampadina, collegata a due fili opportunamente dotati di “coccodrilli” (morsetti a molla) può servire a “misurare” (molto a occhio o in maniera “booleana”: c’è/non c’è) la tensione presente in un punto di un circuito. L’esempio più classico è

nell'ambito degli impianti elettrici delle automobili[26]: collegando uno dei coccodrilli a "massa" (tipicamente al telaio dell'auto) si può saggiare, toccando in vari punti dell'impianto elettrico, se arriva tensione (i.e., si accende la lampadina) e quindi se un certo attuatore (motorino avviamento, motore tergicristalli, ecc.) riceve alimentazione o meno. Se riceve alimentazione può essere rotto, se non la riceve il difetto è a monte. Ovviamente la lampadina deve essere dimensionata per un range di tensione vicino all'impianto che si vuol saggiare: non ha senso usare una lampadina a 220V su un impianto elettrico d'auto (12V), non si accenderà mai.

La versione del tester-lampadina in ambito embedded è un LED (da solo se la tensione del circuito lo consente, con una resistenza in serie altrimenti) che si può connettere ad un piedino GPIO di una board per vedere se il programma che abbiamo scritto "emette qualcosa". Su Arduino ad esempio ne trovate uno già saldato sulla board, connesso ad uno dei piedini di I/O digitale (il 13 sulla Arduino UNO).

La versione "sofisticata" del tester di cui sopra è dotata anche di una pila/batteria in serie con la lampadina. Con questo strumento si può verificare se un circuito è continuo, se non ha interruzioni. Ad esempio per saggiare la continuità di un cavo di alimentazione interrato, invece che dissotterrarlo, basta cortocircuitare i capi ad un estremo e collegare il nostro "tester con lampadina" ai capi dell'estremo rimasto: se la lampadina si accende il cavo è sano.

3.6.2 Voltmetro

Il voltmetro misura la tensione (differenza di potenziale, Volt) tra due punti di un circuito. Si può usare su un circuito alimentato e "intonso" (senza staccare nulla o tagliare collegamenti), cioè **in parallelo**. Serve ad esempio a capire se ci sono sezioni del circuito senza alimentazione o con alimentazione insufficiente: su molti schemi elettrici è indicata la tensione *expected* -

[26]Ci riferiamo alle parti molto "elettriche", non quelle "elettroniche". Cioè escludiamo la comunicazione con le centraline moderne per le quali serve un PC e un collegamento CAN-BUS (cfr. sezione 5.2.6) o perfino Ethernet/WiFi.

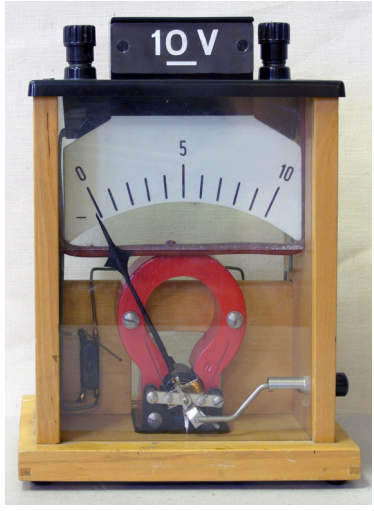


Figura 3.24: Voltmetro analogico (fonte: Hannes Grobe)

attesa - nei punti principali del circuito, addirittura in alcune schede, tipicamente in elettronica analogica, vengono predisposti dei *testpoint* ("piloncini" saldati sulla scheda) a cui applicare voltmetri e/o strumenti di vario genere.

3.6.3 Amperometro, pinza amperometrica

L'amperometro misura la corrente che scorre in un determinato ramo (dove viene applicato lo strumento) del circuito. Salvo

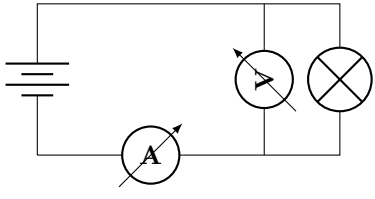


Figura 3.25: Collegamento di un voltmetro (parallelo) e di un amperometro (serie)

casi particolari (pinze amperometriche, che però sono poco sensibili alle basse correnti) l'amperometro va inserito **in serie**, cioè interrompendo il circuito nel punto di misura e inserendovi i terminali dello strumento. Serve a capire quale intensità di corrente (quanti elettroni per unità di tempo) fluisce, sia per stimare un consumo (es. in figura 3.25 per misurare il consumo della lampadina in funzione) sia per valutare la potenza impiegata (mediante le leggi di Ohm: $P = R \times I^2$).

Nota bene: la resistenza interna di un amperometro è bassissima dato che deve influenzare il meno possibile il circuito che sta misurando (e in cui si frappone) ergo attenzione a non collegarlo MAI direttamente (in parallelo) ad una fonte di alimentazione, il risultato sarebbe un classico "corto circuito". Nella migliore delle ipotesi scattano (se presenti!!!) le protezioni elettroniche (magnetotermico, fusibile, ecc.), nella peggiore, in caso di alte correnti in gioco, si fondono i cavi[27].

La "pinza amperometrica" è un particolare tipo di amperometro che non necessita l'interruzione del circuito da misurare: è proprio una pinza con due ganasce che si chiudono su un filo passante, la misura viene fatta per induzione elettromagnetica su correnti alternate e per effetto Hall[28] su correnti continue.

3.6.4 Ohmmetro, prova-diodi

L'ohmmetro, come si può intuire dal nome, misura i valori di resistenza, ad esempio dei componenti "resistenza" (appunto) se non si ha tempo, voglia o occhi buoni per leggere il codice colori. Indirettamente serve anche per saggiare grossolanamente la condizione di una giunzione (diodi e transistor): si può

[27] Uno degli autori ha vissuto di persona la fusione (a causa di un corto circuito... ehm, causato dall'autore stesso) pressoché istantanea di un impianto elettrico (senza alcuna protezione) di una vecchia cantina: nel giro di pochi secondi si sono sciolte e incendiate le guaine plastiche di TUTTI i fili presenti a monte del corto circuito!

[28] L'effetto Hall descrive la comparsa di una tensione in un conduttore percorso da corrente e immerso in un campo magnetico (esterno, prodotto ad esempio da un altro conduttore sotto osservazione), misurando questa "tensione di Hall" è possibile calcolare l'intensità del campo magnetico (nota la corrente).

capire se la giunzione è interrotta (resistenza "infinita" in entrambi i sensi), in corto (resistenza zero in entrambi i sensi) o "sperabilmente"[29] sana (resistenza infinita in un senso e zero nell'altro).

A differenza dei due strumenti precedenti l'ohmmetro è uno strumento **attivo**: è composto da un amperometro e da una pila[30] che fornisce la tensione di riferimento[31]. Al componente da misurare viene applicata la tensione di riferimento, misurando quindi la corrente che scorre nel circuito e applicando le ormai onnipresenti leggi di Ohm si può ricavare il valore di $R_{incognita} = V_{riferimento}/I_{misurata}$.

3.6.5 Multi-tester/multimetro

Voltmetri, amperometri, ohmmetri... esistevano nei vecchi laboratori che ormai si vedono solo nei film. Oggigiorno, salvo rari casi di strumenti di altissima precisione, le funzioni vengono integrate nei cosiddetti "multi-tester/multimetri" (in figura 3.26 una versione analogica *vintage*), strumenti con un singolo quadrante o display, configurabili utilizzando un selettore.

3.6.6 Oscilloscopio

L'oscilloscopio, come dice il suo etimo, serve ad osservare le forme d'onda. Se al suo ingresso si immette un "segnale" (cioè ad esempio lo si collega tra due punti di un circuito come si farebbe con un voltmetro) variabile nel tempo l'oscilloscopio produce sul proprio display (catodico o, più recentemente, LCD) il grafico dell'andamento della tensione nel tempo. In figura 3.27 si vede un apparato in funzione che mostra un'onda a dente di sega e una quadra, molti oscilloscopi moderni hanno infatti più ingressi in modo da poter confrontare più segnali contemporaneamente

[29] Servono successivamente strumenti più sofisticati per capire se la curva di lavoro è integra o meno. Comunque i danni tipici di una giunzione sono "booleani", è raro che un semiconduttore si rovini funzionando poi in maniera parziale.

[30] Nota bene: ricordarsi di cambiarla ogni tanto!

[31] In alternativa si può costruire con un voltmetro, in questo caso viene misurata la caduta di tensione sul componente sotto esame.

Figura 3.26: Tester *vintage* ICE

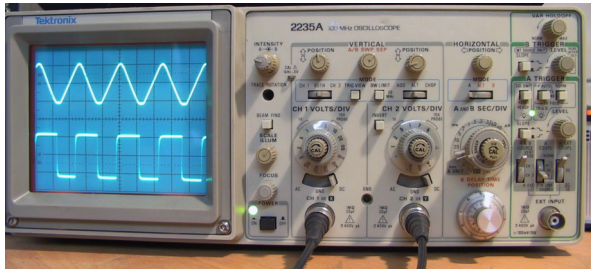


Figura 3.27: Oscilloscopio in funzione

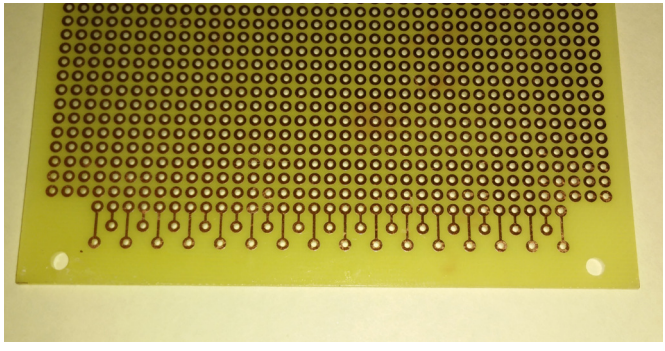


Figura 3.28: Basetta “millefori”

e, soprattutto, sincronicamente (verificando cioè le fasi dei vari segnali).

Ogni oscilloscopio è costruito per supportare una determinata frequenza massima di ingresso, oltre la quale il segnale non viene rappresentato correttamente. Il prezzo degli apparati aumenta all'aumentare della massima frequenza di ingresso gestita.

3.7 Montaggio fisico

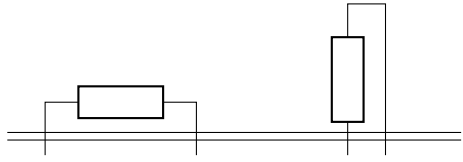
Per realizzare un circuito elettronico bisogna **connettere** più componenti tra loro. Nel corso della storia i modi per realizzare queste connessioni sono cambiati, passati di moda, ritornati in auge, criticati e lodati, sperimentati e migliorati... Oltre alla funzione di connessione è necessario anche un **supporto fisico** che dia stabilità e robustezza meccanica ai circuiti.

Le connessioni sono principalmente di due tipi: per attrito, per saldatura.

I supporti fisici in genere sono rappresentati da “basette” (piani rettangolari di materiale plastico isolante semi-rigido) dotate o meno di “piste” conduttive[32]. Tali basette sono forate in modo da poter infilare i terminali dei componenti e fungere

[32]Nei “circuiti stampati” moderni le piste sono anche multi-piano/strato.

da sede e supporto strutturale (es. figura 3.28). I terminali dei componenti si infilano nei fori predisposti sulla basetta, il componente può essere posizionato sia in orizzontale che in verticale:



I componenti vengono poi di solito saldati alle "piazzole" ramate (attorno ad ogni foro sulla basetta).

Da qualche anno a questa parte sono entrate in commercio le cosiddette *breadboard* (figura 3.29), supporti evoluti rispetto alle basette tradizionali. Le *breadboard* sono comunque delle "millefori", ma con notevoli miglioramenti:

- i componenti si inseriscono a pressione sulla scheda, vengono tenuti in posizione da contatti a molla, ergo non c'è bisogno di saldare;
- i fori sono "cablati" nel senso che alcune righe e alcune colonne sono già elettricamente connesse tra loro[33], ergo si risparmiano collegamenti;
- sono strutturalmente abbastanza robuste e sono dotate di una striscia adesiva sul fondo (per il montaggio senza viti dentro ai contenitori).

Attualmente il processo di sviluppo di una "applicazione" embedded prevede la prototipazione su *breadboard* per poi passare al progetto dei circuiti stampati definitivi "di produzione". Molti strumenti software, anche liberi (come Fritzing `http://fritzing.org`) supportano il disegno integrato dello schema elettrico, della board e del circuito stampato nell'ambito della stessa applicazione. Il "circuito stampato" (o PCB - Printed Circuit Board) è una basetta in materiale isolante (tipo quello della millefori) su cui vengono depositate le piste in rame che

[33] In figura 3.29 - ruotata di 90 gradi, messa quindi in verticale - tutti i fori sulla **colonne** "+" (e "-") sono connessi e servono per alimentare più componenti, tutti i fori su ogni **riga** da 1 a 63 (lato ABCDE e FGHIJ separatamente) sono connessi e servono a posizionare componenti ed effettuare collegamenti con i cavetti flessibili (*patch*, *Dupont*) come mostrato.

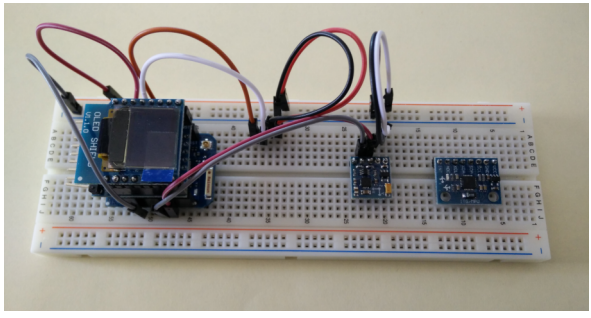


Figura 3.29: Breadboard

realizzano il circuito finale, i componenti verranno poi saldati sulla basetta.

3.8 Sensori

Cos'è un **sensore**?

Parafrasando e riassumendo da Treccani[1] lo si può definire un oggetto basato su un materiale che cambia le sue proprietà elettriche[34] al variare delle condizioni ambientali a cui è sottoposto. Con "proprietà elettriche" si intende ad esempio la resistenza al passaggio di una corrente: una fotoresistenza, come dice la parola stessa, è una resistenza sensibile alla luce, varia la sua resistenza interna al variare della luce che la colpisce; avendo una sorgente di potenziale (i.e., una pila) si può misurare l'intensità luminosa applicando (ohibò!) le leggi di Ohm ($V/I = R$), in questo caso usando quindi un Ohmmetro.

Molti sensori sono semplicissimi costruttivamente, si pensi ad una termocoppia: due metalli diversi accoppiati solidalmente a forma di barra, la temperatura dilata diversamente i due metalli, la barra si piega e può andare a toccare un contatto

[34] O ottiche o meccaniche, ecc., ma limitiamoci al mondo elettronico. Anche perché qualora si disponesse solamente di un sensore puramente meccanico sarebbe sempre possibile accoppiarlo ad un sensore "elettrico" per effettuare una misura indiretta. Ad esempio un manometro a depressione (meccanico) accoppiato ad un potenziometro (movimento $->$ resistenza).

elettrico (sensore booleano, circuito aperto o chiuso, resistenza infinita o zero).

Altri sensori sono più complessi, coinvolgono magari alcuni semiconduttori[35] e potrebbero perfino, come si suol dire, "uscire in numerico"[36]

Alcuni esempi interessanti, specie dal punto di vista didattico, qui di seguito.

- microfono/sensore di rumore a condensatore: se una delle due "piastre" di un condensatore è montata in modo che possa muoversi quando investita da onde sonore si ottiene un condensatore la cui capacità varia in funzione del rumore ambientale;
- microfono/sensore di rumore a carbone: la polvere di carbone impacchettata esprime una resistenza elettrica che è funzione della compressione, quando onde sonore colpiscono il pacchetto si nota una variazione della resistenza che segue la forma delle onde stesse;
- interruttore a mercurio[37]: un bulbo di vetro (tipo piccola lampadina) con due elettrodi inseriti e contenente una goccia di mercurio, orientandolo nello spazio la goccia si muove e va a chiudere il contatto elettrico sugli elettrodi realizzando un "tilt sensor" (sensore di pendenza);
- sensore di distanza a ultrasuoni: una coppia mini-buzzer (emettitore di suoni) e mini-microfono, tipico sensore complesso (dotato cioè di circuiteria non banale di contorno), il buzzer emette ultrasuoni che il microfono cattura se riflessi da qualche ostacolo, il tempo di andata e ritorno[38] viene utilizzato per calcolare la distanza da un ostacolo "riflettente";

[35] Tutti i semiconduttori sono sensibili alla luce, alla temperatura, ai campi magnetici, alle radiazioni, ecc., drogandoli opportunamente è possibile amplificare questi "difetti".

[36] Invece di esporre l'informazione sotto forma di grandezza elettrica emettono direttamente informazione digitale (bitstream) secondo qualche tipo di protocollo, ad es. un treno di impulsi.

[37] Oggi difficilmente reperibili a causa del bando sull'uso del mercurio.

[38] Velocità del suono circa 350 m/s.

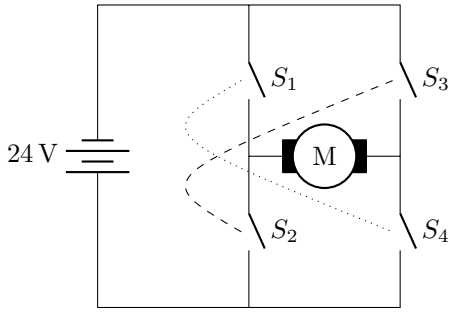


Figura 3.30: Ponte H

3.9 Attuatori

Cos'è un **attuatore**?

Un attuatore è un oggetto che cambia le sue proprietà meccaniche al variare dei segnali elettrici che gli vengono forniti. Il già citato relè (sezione 3.3.3) è un attuatore, anche se particolare, perché varia le sue proprietà elettriche (passando per quelle meccaniche di un elettromagnete!) in funzione della corrente che vi scorre. Un semplice motore a corrente continua è un attuatore: passa da *statico* a *in moto* in funzione della tensione/corrente. Un servomotore o un motore "passo-passo" sono motori particolari che in funzione del segnale in ingresso posizionano il loro asse secondo un angolo (più o meno) preciso. Un *asciugacapelli* trasforma un "segnale" (un po' potente invero!) in vento e calore. E via così.

Un circuito interessante per comandare un motore a corrente continua è il cosiddetto "ponte H" (figura 3.30, una semplice combinazione di "interruttori" (viene tipicamente realizzato mediante relè o semiconduttori[39]). Chiudendo opportunamente (i.e., o S_1, S_4 o S_2, S_3 ovviamente mai S_1, S_2 o S_3, S_4 pena un corto circuito sull'alimentazione) i contatti è facilissimo decidere il senso di rotazione del motore M a corrente continua.

[39]Il più noto e usato è l'integrato L298 (`https://www.st.com/en/motor-drivers/l298.html`) che sovente si vede montato nei vari *shield* per Arduino.

3.10 MEMS

I MEMS (Micro Electro-Mechanical Systems)[14] sono device meccanici molto, ma molto, piccoli, così piccoli da poter essere inglobati in un chip. Sono "meccanici" nel senso che si basano su effetti fisici (es. il movimento di una, pur piccola, massa) che vengono misurati e trasformati in dati numerici. Sono anche "siliconici" (anche se non è detto che il materiale utilizzato sia sempre il silicio) nel senso che il processo produttivo per realizzarli non è meccanico ma utilizza le tecniche derivate dalla costruzione dei chip e dei semiconduttori, vengono cioè usate tecnologie di deposizione dei materiali e fotolitografia.

I sensori (o perfino micro attuatori!) che si possono realizzare sono di varia natura, ad esempio: accelerometri (misurano il movimento di una piccolissima massa), microfoni (piezoelettrici o meccanici), sensori di pressione/forza, proiettori (basati sulla regolazione di tanti micro specchi), interruttori ottici (idem), sensori biomedicali, trasduttori ultrasonici, micro motori elettrostatici, ecc.

Capitolo 4

Architetture Embedded

Lo scopo di questo capitolo è quello di fornire una introduzione generale alle architetture hardware presenti nei sistemi embedded più comuni. Pur non scendendo nei dettagli implementativi ed architetturali di ogni singola tecnologia presentata, si vuole dare al lettore una comprensione generale del sistema e delle sue componenti, sapendole collocare per caratteristiche e funzionalità. Nel corso degli anni della storia dello sviluppo dei sistemi embedded, sono state numerose le architetture proposte e utilizzate nei più svariati ambiti d'applicazione con alterne fortune, verranno prese in esame quelle più significative.

4.1 Instruction Set Architecture

Per definire una architettura è necessario conoscere quello che è il suo modello astratto; tale modello prende il nome di *Instruction Set Architecture* (ISA), e comprende la tipologia di dati che può trattare, i registri, la modellazione di input/output e le istruzioni macchina. Vale la pena di ricordare che da un singolo ISA possono derivare più implementazioni, ovvero più processori possono condividere il medesimo ISA. Questo concetto è

fondamentale per capire come, software scritto e compilato per un determinato ISA (o un suo sottoinsieme) possa funzionare su CPU di produttori e periodi differenti, ma con lo stesso Instruction Set.

In base alla tipologia di ISA è possibile distinguere 4 differenti tipi di architetture:
CISC: *Complex Instruction Set Computing*
RISC: *Reduced Instruction Set Computing*
VLIW: *Very Long Instruction Word*
EPIC: *Explicitly Parallel Instruction Computing*

Le architetture VLIW ed EPIC sono architetture ideate per raggiungere ottimi livelli di parallelizzazione nella esecuzione delle istruzioni. Per quanto siano estremamente interessanti, non rientrano negli scopi di questo testo, nel quale si farà riferimento esclusivamente a CISC e RISC.

Prima di approfondire queste tipologie di architetture, è giusto puntualizzare che nonostante ci siano delle enormi differenze teoriche tra CISC e RISC, nelle varie implementazioni pratiche tali differenze possono risultare meno accentuate, con la possibilità di vedere degli ISA "ibridi". In questi termini è più corretto parlare di "Approccio RISC e "Approccio CISC", specificando quale sia l'idea generale alla base.

4.1.1 CISC

CISC (Complex Instruction Set Computing) è un termine creato a posteriori rispetto allo sviluppo di queste architetture (per distinguerlo dall'opposto RISC). Con tale dicitura si intende quella tipologia di processori caratterizzati da architetture con un notevole numero di *"low-level instructions"* (istruzioni a basso livello) ed in cui le istruzioni sono a lunghezza variabile e possono richiedere molteplici cicli di clock per essere eseguite.

Il loro vantaggio è offrire un linguaggio macchina espressivo con istruzioni non molto distanti da quelle di un linguaggio di alto livello, quindi apparentemente più facili da programmare. Contestualmente però, tale approccio risulta più costoso in termini di "silicio", ovvero come numero di componenti interne necessarie a realizzare l'architettura.

Alcuni esempi di CPU che nel tempo sono stati etichettati come CISC:

- System/360
- VAX
- PDP11
- Z80
- Motorola 68000
- Intel 8080
- MT6502
- x86

L'architettura x86 è un esempio di quanto il confine tra questi due approcci possa diventare sottile in fase realizzativa; se apparentemente si presenta con le caratteristiche tipiche del CISC, con un elevato numero di istruzioni complesse, in realtà dispone di una sezione interna atta alla traduzione delle istruzioni CISC in microistruzioni che vengono poi elaborate in modo simile a quanto avviene nei processori RISC.

4.1.2 RISC

RISC (Reduced Instruction Set Computing) è un design per CPU basato su un *instruction set* semplificato, composto da istruzioni semplici, molto ottimizzate e in grado di essere eseguite con pochi cicli di CPU (idealmente solo uno, anche se nella pratica non è sempre così). Il vantaggio di un simile approccio è di avere un basso costo in "silicio" (da tradursi con una logica interna ridotta) e una grande efficienza, ma a costo di una notevole distanza dai linguaggi di alto livello e con una conseguente elevata difficoltà di programmazione a basso livello. Per colmare tale distanza sono necessari dei compilatori assai più efficienti rispetto al modello CISC.

Alcuni esempi di processori RISC:

- ARM
- ARC
- Blackfin
- Atmel AVR
- MIPS
- PA-RISC

- SuperH
- SPARC

A supporto di quanto detto in merito al sottile confine tra CISC e RISC sul piano pratico, c'è da considerare il fatto che ad oggi molti processori dichiaratamente RISC, hanno un instruction set più ricco di alcuni processori CISC. Considerare il termine "Reduced" riferito al numero di istruzioni potrebbe infatti risultare fuorviante, e sarebbe forse più indicato considerarlo in merito alla complessità delle istruzioni che non al loro numero complessivo.

Osservando la lista delle architetture RISC, è facile notare come la maggior parte di quelle più affermate nel mondo embedded appartenga a questa famiglia, mentre quelle CISC abbiano dominato in ambito Desktop/Server/Mainframe per lungo tempo.

4.1.3 Endianness

Come noto, i calcolatori utilizzano la logica binaria memorizzando ed utilizzando dati sotto forma di bit (singoli digit di valore 0 o 1).

Gruppi di 8 bit prendono il nome di byte. Un numero, ad esempio 48879 (in esadecimale 0xBEEF) viene rappresentato in forme binaria come ***10111110 11101111***.

Gli 8 bit a sinistra, proprio perché in quella posizione, sono i bit più significativi, ovvero quelli che rappresentano la parte del numero con le potenze di 2 più elevate. Azzerando i bit a destra, infatti, il valore rappresentato corrisponderebbe al valore 48640 decimale (0xBE00).

Gli 8 bit a destra, proprio perché posti in quella posizione, invece sono quelli meno significativi, ovvero quelli che rappresentano la parte del numero con le potenze di 2 più basse. Azzerando i bit a sinistra, infatti, il valore rappresentato in decimale diventa 23 (0x00EF).

Una caratteristica distintiva degli elaboratori è la *Endianness*; con tale termine si indica l'ordine con cui il calcolatore legge e memorizza i byte nella memoria.

Little Endian: i byte sono memorizzati a partire dal byte meno significativo al più significativo.

Big Endian: i byte sono memorizzati a partire dal byte più significativo.

Per semplificare, in un sistema *Big Endian*, la rappresentazione esadecimale del valore memorizzato è **BE EF**, mentre in un sistema *Little Endian* i byte (ma non i bit) sono invertiti, cioè **EF BE**.

4.1.4 FPGA

Prima di affrontare le varie architetture hardware per embedded, ha senso menzionare un tipo particolare di tecnologia nota comunemente come **FPGA**, acronimo di *Field-Programmable gate array*. Semplificando molto il concetto: si tratta di un insieme variabile per numero, in base al produttore ed al modello, di *logic gate* (tr. porte logiche) con connessioni programmabili. In molti casi in realtà, è assai più probabile che si usino i cosiddetti **logic blocks**, ovvero l'unione funzionale di più porte logiche (es. i *flip-flop*.

La loro struttura interna, la possibilità di programmazione, ed il grande numero di connessioni tra i vari blocchi logici, rende le FPGA estremamente configurabili (è di fatto possibile impostare le FPGA ad eseguire funzioni differenti), ma proporzionalmente assai più costose rispetto ad una MCU o un SoC: tali fattori rendono questo strumento particolarmente adatto in fase di progettazione dell'hardware, ma assai improbabile il loro impiego all'interno di un prodotto finale. I maggiori produttori di FPGA sono Xilinx e Altera (ora di proprietà Intel).

Per semplificare ancora il concetto, si immagini di poter scrivere un elemento hardware di elaborazione con degli strumenti software, definendone il comportamento con un linguaggio di alto livello; ciò ha velocizzato esponenzialmente la possibilità dei produttori di progettare e testare le ***IP*** senza affrontare gli enormi costi della stampa su silicio.

I linguaggi utilizzati per le FPGA vengono genericamente definiti **HDL** (*Hardware description language*), tra cui si possono citare i seguenti:

Verilog (IEEE1364) Uno dei linguaggi più utilizzati per la progettazione di IP;

VHDL (IEEE1164) Acronimo di *VHSIC Hardware Description Language*, è un linguaggio più strutturato rispetto al verilog, con maggiori possibilità di portabilità;

SystemC Un set di classi C++ per descrivere il comportamento hardware desiderato. Se pur molto pratico e di facile impatto per chi viene dalla programmazione software, trova meno riscontro in ambiti ampiamente professionali.

In base al tipo di produttore della FPGA e del HDL di riferimento esistono numerosi manuali su cui è possibile approfondire questo argomento, di sicuro interesse per chi vuole passare alla progettazione Hardware, ma che è molto fuori dal perimetro di questo testo.

4.2 ARM

I primi processori ARM nascono dalla Acorn Computers (da lì il primo significato dell'acronimo: Acorn Risc Machine) nei primi anni 80. In un epoca in cui si scorgeva già il futuro dominio dei computer definiti *"IBM PC compatibile"*[1], l'azienda di Cambridge aveva rivolto la propria attenzione ad un'implementazione migliorata del già noto MT6502 (considerato troppo modesto per adeguarsi alla nuova era delle interfacce grafiche). L'idea generale era di produrre un'architettura estremamente semplice ed efficiente, da contrapporre alla complessità delle architetture Intel e Motorola (quest'ultimo utilizzato dagli emergenti computer Apple), riducendo significativamente i consumi. Se in un primo momento la Acorn aveva scelto di produrre direttamente i chip, appoggiandosi a VLSI come partner, in un secondo momento decise che sarebbe stato più conveniente licenziare i propri prodotti a ditte terze.

[1] Con "IBM PC compatibile" si identificava tutta quella serie di computer compatibili con i computer IBM PC, XT e AT, anche se in seguito tale espressione si è ridotta *PC Compatibile* riferita a tutti i computer desktop che utilizzavano processori con architettura x86 e sistema operativo Microsoft DOS/Windows

Nel corso degli anni, ARM (divenuta poi Advanced Risc Machine Ltd) ha proseguito lo sviluppo, raggiungendo la leadership in tutti i campi del settore embedded, ed arrivando di recente ad aggredire il mercato home computing e server.

4.2.1 Architettura e *core*

Occorre fare alcune precisazioni relative alla nomenclatura dei *core* e delle architetture:

Architettura: con architettura si indica il macro-insieme di CPU che condividono il medesimo ISA (Instruction Set Architecture), al netto delle estensioni. Ad una singola architettura possono corrispondere dei *core* assai differenti.

Profilo: a partire dalla famiglia dei Cortex, si identificano tre diversi profili sulla base dello scopo per cui è stato progettato il *core*;

- Profilo A (Application): in genere utilizzati per dispositivi multimediali quali tablet, smartphone, set top box.
- Profilo R (Real-time): utilizzati per sviluppare dispositivi in grado di lavorare in tempo reale. Il requisito real-time richiede numerosi interventi sia hardware che software.
- Profilo M (Microcontroller): sono utilizzati per produrre microcontrollori (MCU).

Core: identifica con precisione l'architettura, le estensioni architetturali utilizzate e le caratteristiche interne.

A seguire una tabella di sintesi su alcune delle architetture e dei core di riferimento. La lista completa è assai più lunga ed esaustiva (per non parlare degli esempi), ma non utile ai fini della trattazione.

Architettura	Profilo	Core	Esempio
ARMv1	-	ARM1	-
ARMv2	-	ARM2,ARM250, ARM3	-
ARMv3	-	ARM6, ARM7	-
ARMv4	-	ARM8	StrongARM
ARMv4T	-	ARM7TDMI, ARM9TDMI	XScale
ARMv5	-	ARM7EJ,ARM9E,ARM10E	TI DaVinci
ARMv6	-	ARM11	Raspberry Pi
ARMv6-M	M	Cortex-M0(+), Cortex-M1	Arduino ZeroPro
ARMv7A	A	Cortex-A(5,7,8,9,12,15,17)	BeagleBoneBlack
ARMv7R	R	Cortex-R(4,5,7) -	
ARMv7M	M	Cortex-M(4-7),	STM32(F4xx)
ARMv8A	A	Cortex-A(53,57,72)	Apple iPad

4.3 MIPS

MIPS (acronimo di *Microprocessor without Interlocked Pipeline Stages*) nasce come progetto di sviluppo presso la Stanford Univerity (Santa Clara, California) nei primi anni 80. La particolarità, da cui deriva il nome, è che tutte le istruzioni presenti nella pipeline dovevano essere concluse in un solo ciclo di clock, evitando stalli e ritardi ed eliminando di fatto la necessità di un sistema di controllo (interlock). L'idea alla base delle architetture MIPS (e in generale delle architetture RISC) era semplificare le operazioni complesse (es moltiplicazioni e divisioni) in una serie di operazioni più semplici. Tale approccio, se pur accademicamente interessante, fu poi abbandonato nel momento in cui la neonata MIPS Computer Systems (poi MIPS Technologies dopo l'aquisizione di SGI) si propose sul mercato. A partire dagli anni '90 i processori MIPS furono licenziati a produttori terzi, e furono proposte architetture a 32bit (MIPS32) e a 64bit (MIPS64). A differenza di ARM, MIPS trova maggiore successo nel mercato dei SuperComputer, ovvero per quelle macchine progettate per disporre di una grande potenza di calcolo. Tra i core ideati da MIPS risultano di grande successo il MIPS R3000A (PlayStation) e R4000 (Playstation 2).

Dal 2012 MIPS Technologies è passata sotto il controllo di Imagination Technologies e sono stati annunciati nuovi prodotti appartenenti ai profili M (Microcontroller, paragonabili alla famiglia dei CortexMx), I (Interaptiv, paragonabili alla famiglia dei CortexRx e con la fascia bassa dei CortexAx) e P

(Performance, paragonabili alla famiglia dei CortexAx di fascia alta).

A seguire una tabella di sintesi su alcune delle architetture e dei core di riferimento MIPS dal 1999 al 2015. Anche in questo caso, la lista completa sarebbe assai più lunga e contestualmente non necessaria agli scopi di questa trattazione.

Nome	**Anno**	**Max Freq. in MHz (ad oggi)**
4k	1999	-
24k	2003	1468
34k	2006	1454
74k	2007	1080
1004k	2008	1100
1074k	2010	1500
microAptiv	2012	-
interAptiv	2012	-
P5600	2013	2000
M5100	2014	497
M5150	2014	576
P6600	2015	2000
I6400	2014	1000
M6200	2015	750

Notare bene che la massima frequenza di esercizio è indicata sulla base della implementazione più veloce sulla litografia più performante, in alcuni casi realizzata molti anni dopo l'effettivo rilascio della prima implementazione.

4.4 ARC

I processori ARC (Argonaut Risc Core) sono una famiglia di processori disegnati e progettati dalla ARC International (dal 2010 passata a Synopsys). Caratterizzati da un ISA molto flessibile (è possibile aggiungere delle istruzioni "custom" - con relativa logica - all'ISA), i core ARC sono utilizzati quasi esclusivamente per applicazioni embedded. La loro semplicità dal punto di vista del design consente ai SoC che utilizzano questa architettura di lavorare a frequenze di 2GHz con consumi assai ridotti (circa 100mW su ARC HS, 28nm). A seguire, alcuni

esempi dei core ARC ampiamente utilizzati nell'elaborazione di dati da sensori, DSP ed applicazioni audio.

ARC HS: ideale per applicazioni embedded che richiedono una elaborazione rapida dei segnali, può raggiungere frequenze di 2.2GHz con un consumo approssimativo di 80mw. Il core HS38 dispone anche di MMU .

ARC EM: utilizzati generalmente per DSP e o per l'elaborazione dati da sensori. Questi core hanno un area di stampa su silicio molto piccola, e possono arrivare a consumare 3μW/MHz.

ARC 700: questi core sono pensati per applicazioni embedded più impegnative, quale ad esempio un sistema Linux, dispone di MMU, e può raggiungere 1.2GHz su stampa a 28nm.

ARC 600: pensati per l'elaborazione audio, i core della famiglia 600 sono considerati ottimali per l'uso come DSP.

Soundwave Audio System: ideato specificatamente per sistemi audio, utilizzato in dispositivi quali TV, set-top box, blu-ray, dispositivi audio portatili.

I processori ARC sono estendibili, oltre che con MMU (Memory management Unit) e FPU (Floating Point Unit) per le famiglie che già non ne dispongono, anche con particolari estensioni quali ARC XY Advanced DSP (migliora le performance di accesso ad alcune area di memoria), ARC Real-Time Trace ARC Secure.

4.5 AVR

I chip AVR rientrano nella categoria dei microcontrollori (vedere sez. 1.1.2). Ideato da Alf-Egil Bogen e Vegard Wollan presso la Nordic Semiconductor, si tratta di una evoluzione delle architetture Harvard (cfr. capitolo 2.1) ad 8 bit, nota all'epoca come μRISC. Sviluppati da Atmel a partire dal 1996, inizialmente concepiti per competere con l'Intel 8051 (all'epoca il termine di paragone in ambito MCU) con il quale, almeno all'inizio, con-

divideva quasi interamente il *pinout*[2] ad eccezione del pin di reset. Le MCU AVR sono state le prime ad utilizzare una memoria Flash On-Chip[3] al posto delle ROM/EEPROM esterne utilizzate fino a quel momento.

Come avviene di consueto, le MCU AVR sono suddivise in famiglie:

tinyAVR (ATTiny): 0.5-16 kb memoria programma,6-32 pinout

megaAVR (ATMega): 4-256 kb memoria programma,28-100 pinout, istruzioni estese

XMEGA (ATxmega): 16-384 kb memoria programma, 44-64-100 pinout, istruzione estese, ADC.

FPSILC (AVR FPGA): FPGA 5-40k gate, SRAM per la memoria programma.(vedere sez. 4.1.4)

I chip AVR hanno trovato un grande utilizzo nelle applicazioni embedded a bassa richiesta computazionale, ed hanno avuto ancora più popolarità grazie al progetto Arduino (si veda sez. 4.7.2 e cap. 9), che li ha utilizzati in molte configurazioni differenti.

Nonostante non esista una conferma ufficiale, pare che il nome AVR derivi dalle iniziali degli ideatori (A.lf V.egard RISC).

A discapito del nome invece, l'architettura AVR32 ha ben poco a che vedere con l'omonima AVR. Disegnata dalla stessa casa madre degli attuali AVR (Atmel), nasce nel 2006 con il proposito di entrare in competizione alle architetture ARM per sistemi embedded di fascia media.

4.6 Xtensa

Sviluppato da Tensilica (oggi di proprietà di Cadence), Xtensa DPU (Data Plane Processing Unit) è un microcontrollore ad elevate prestazioni, caratterizzato da un approccio RISC (uno dei fondatori, Chris Rowen, aveva partecipato alla prima "stesura" dell'architettura MIPS) e da bassi consumi. Lo scopo, intuibile anche dal nome, era quello di fornire una architettura

[2]Con questo termine si intende la configurazione elettrica e funzionale dei pin

[3]Situata nello stesso package fisico del MCU

compatta ed al tempo stesso estensibile, facendo della estrema flessibilità il suo elemento di punta. Uno degli elementi distintivi al momento del rilascio delle ISA Xtensa, era che, nonostante fosse una architettura a 32bit, utilizzava istruzioni a 16 o 24 bit (riducendo sostanzialmente la dimensione del codice). Ad oggi, altre architetture (quali MIPS e ARM) offrono la possibilità di utilizzare istruzioni di dimensione ridotta su processori a 32bit. Come altre architecture ad approccio RISC, Xtensa può essere utilizzato sia in modalità Big-Endian che Little-Endian.

Nel corso degli anni, Xtensa ha trovato spazio nel mercato DSP legato all'audio (es. Amd TrueAudio) e nei chip Serial-WiFi, quali gli ESP8266.

4.7 Esempi pratici

4.7.1 MCU, SoC e Board

In generale, sia per quanto riguarda i SoC che per quanto riguarda le MCU, si tende ad integrare più componenti possibili all'interno del chip. Ciò nonostante, il solo chip, per quanto funzionale così come prodotto, sarebbe difficilmente utilizzabile per prototipare e sviluppare il prodotto finale. Oltre alla comune gestione energetica (evidentemente necessaria) molti dispositivi necessitano di spazio di storage o di esportare alcune porte di I/O in modo semplice da interfacciare o, ancora, di sensoristica accessoria.

Per questi motivi, molti produttori di chip di varia grandezza e potenza, oltre a fornire la documentazione e talvolta gli strumenti per lo sviluppo software, mettono a disposizione dei clienti anche delle board che a seconda dei casi prendono il nome di *"evaluation board"* o *"development board"*.

Una **eval board**, è solitamente realizzata su PCB di piccole/medie dimensioni, è ideata per far facilitare lo sviluppo hardware e software di prodotti basati sul chip di riferimento, oltre che per promuovere il chip stesso presso gli sviluppatori o i produttori di dispositivi. In alcuni casi infatti, soprattutto in ambienti marketing, queste board assumono il nome di **demo board** (*"demonstration board"*).

Sinteticamente, una *eval board* dispone generalmente di tutti o quasi i collegamenti fisici per l'hardware presente nel MCU o nel SoC, oltre che ad una serie di dispositivi accessori atti a dimostrarne tutte le potenzialità in ambiti applicativi differenti. Oltre ai collegamenti classici (rete, video, audio, bus), una caratteristica mediamente molto apprezzata dagli sviluppatori è la disponibilità di connessioni per il debug a basso livello (es. JTAG), raramente presenti sul prodotto finito.

In base a queste considerazioni si può facilmente capire come la presenza di una eval board ben progettata possa concorrere pesantemente al successo o meno di una determinata piattaforma. Il alcuni casi, come ad esempio nel caso delle RaspberryPI, il successo della eval board ha superato quello della piattaforma stessa, facendola considerare come un prodotto finito da utilizzare così come è, e non per la finalizzazione di un dispositivo differente.

Validation Board

Un caso particolare sono le cosiddette **"Validation Board"**; si tratta di norma di board dalle grandi dimensioni usate per validare una piattaforma (MCU o SoC). Sono di grandi dimensioni perché sono dotate di numerose porte di debug, batterie di pin e test point. Lo scopo di queste board è in genere di effettuare la "silicon validation", ovvero un processo di test e verifica atto a evidenziare la presenza o meno di errori di progettazione o di lacune strutturali all'interno del chip. Raramente una validation board esce sul mercato, ed in generale sono create di volta in volta durante le fasi di validazione di un nuovo SoC o MCU, ed in numero assai limitato.

4.7.2 Arduino

Nato ad Ivrea intorno al 2005[4], Arduino nasce in Italia come progetto di prototipazione elettronica a basso costo per fini didattici. Dedicato al mercato hobbistico ed amatoriale, ha tra le sue caratteristiche più apprezzate la possibilità (da parte dell'utente finale) di disporre completamente (e liberamente!) di tutti i progetti hardware (viene definito Open Hardware) oltre che la

presenza sul mercato di Kit che consentono di personalizzare la board di sviluppo (o di assemblarla interamente).

La piattaforma Arduino ha avuto uno sviluppo rapidissimo in termini di evoluzione con numerose implementazioni differenziate tra di loro a partire dalla scelta del MCU.

Alcuni modelli:

Serial Arduino ATmega8, porta seriale
Arduino Extreme ATmega8, porta USB
Arduino Mini ATmega168 SMD, board mini
Arduino Nano ATmega168 SMD, board nano, USB
LilyPad Arduino ATmega168 SMD, wearable
NG ATMega8, porta USB
NG Plus ATMega 168, porta USB
BT ATmega168, Bluetooth
Diecimila ATMega168, porta USB
Duemilanove Atmega328, Alimentazione DC
Mega ATmega1280, memoria addizionale.
Mega2560 ATMega2560
Due Atmel SAM3X8E Cortex-M3
Zero Pro Atmel SAMD21 Cortex-M0+

Per comprendere le differenze tra le varie piattaforme si possono comparare l'ATMega8[7] e il SAM3X8E[8] dove si può notare l'estrema variabilità del range hardware disponibile:

	ATMega8	SAM3X8E
arch	AVR	ARM
bit	8	32
flash (kb)	8	512
Max I/O Pin	23	103
Max freq(MHz)	16	84
USB	-	1

Sviluppo software

Tra i motivi del grande successo di Arduino nel campo hobbistico, oltre al fatto di essere molto aperto ed economico, c'è anche l'ambiente di sviluppo. Le piattaforme Arduino infatti dispongono di un ambiente di sviluppo (IDE) derivato da Wiring (ambiente di sviluppo integrato scritto in Java e rilasciato

con licenza libera[4] e basato su Processing (linguaggio di programmazione ad oggetti simile a Java e rilasciato con licenza GPL). Sia il linguaggio di programmazione che l'IDE sono pensati e sviluppati al fine di risultare estremamente semplici dal punto di vista del programmatore, mascherando il più possibile gli aspetti legati alle funzionalità di basso livello, per concentrarsi sugli aspetti creativi e funzionali (di alto livello). Ciò ha fornito, nel bene e nel male, una grande accessibilità allo sviluppo anche da parte di utenti senza grande dimestichezza con la programmazione. I programmi realizzati con questo IDE prendono il nome di **"sketch"**. Il caricamento dello sketch sul dispositivo è piuttosto semplice, e differisce tra i vari modelli solo per le caratteristiche di collegamento del dispositivo (USB, Seriale, Bluetooth, WiFi) al computer di sviluppo.

Il capitolo 9 presenta un approfondimento dedicato allo sviluppo su Arduino.

4.7.3 Discovery STM32

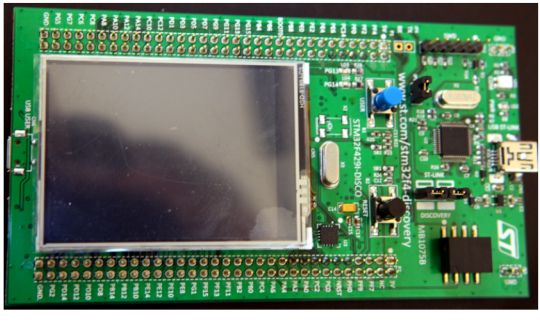


Figura 4.1: Una discovery board STM32 F429 dotata di LCD

STM32[52] è una famiglia di microcontrollori creati da ST-Microelectronics, basati sui core ARM, profilo M. L'estrema versatilità, l'ampia gamma di prodotti e la semplicità di sviluppo su questa piattaforma, hanno attirato l'interesse di nume-

[4] Per una definizione di "licenza libera" cfr. sezione 6.1.1

rosi sviluppatori, oltre che una discreta adozione all'interno di progetti di sistemi embedded.

Anche in questo caso non è possibile fare riferimento ad STM32 come ad un unico chip; nel corso degli anni STMicroelectronics ha fornito svariati modelli di questo MCU, differenziati notevolmente da hardware e dotazione di base.

Famiglia	Core	SRAM	Flash	Velocità	Novità
F0	Cortex-M0+	4-20 kb	16-128 kb	48 MHz	-
L0	Cortex-M0+	8 kb	32-64 kb	32 MHz	USB
F1,F2	Cortex-M3	4-128 kb	16-1024 kb	72-120 MHz	F2[5]
F3,F4	Cortex-M4F	16-192 kb	64-2048 kb	72-180 MHz	F4(LCD)
F7	Cortex-M7	-	-	-	-

Si veda ad esempio la board 32F429IDiscovery[47] (figura 4.1), basata su STM32 F4 e le sue caratteristiche pricipali:

- Core: Cortex-M4F 84/168/180 MHz
- Static RAM: 512/1024/2048 kb
- USB 2.0 OTG
- CAN 2.0B
- SPI/I2C/I2S, USART, UART, SDIO, ADC, DAC, GPIO, WDT
- LCD-TFT controller

STM32 Nucleo

Vogliamo anche citare la board "Nucleo" che è un prodotto di ST relativamente recente per la prototipazione di progetti basati su STM32. Come si può vedere in figura 4.2 il formato è analogo a quello di Arduino (tanto che STM32 fornisce una piedinatura compatibile per utilizzare gli *shield* pensati per Arduino), il caricamento del software avviene via USB utilizzando un ambiente di sviluppo C/C++ in locale (ad es. esistono *plugin* per Eclipse), ma viene anche supportato l'ambiente web puro "Mbed"[6] (cfr. figura 4.3), un interessante progetto che permette lo sviluppo software per molte[7] piattaforme embedded senza dover installare nulla sul proprio pc, si accede via web (bisogna

[6] `https://www.mbed.com/en/`

[7] 122 al 2 agosto 2017

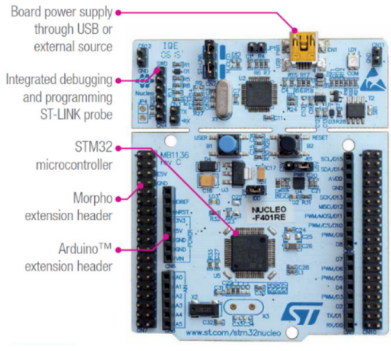


Figura 4.2: Board Nucleo (fonte: ST)

creare un account) e i binari compilati vengono scaricati in locale per poi essere caricati (USB, seriale, ecc. in funzione di ciò che si dispone) sulla board. La Nucleo viene vista dal computer come un disco esterno USB, basta "droppare" il file binario sul disco perché la board lo carichi nella propria flash.

Software

Nonostante ci siano stati già dei risultati incoraggianti dal punto di vista del porting di sistemi GNU/Linux su questo tipo di piattaforma, le prestazioni ridotte (se comparate con un SoC) non rendono questa alternativa ancora valida. In alternativa, esistono un gran numero di ambienti di sviluppo (Keil, IAR, Atollic, TrueStudio o anche direttamente una toolchain basata su `gcc`) che consentono di sviluppare "*from scratch*"[8] una propria applicazione personalizzata, appoggiandosi o meno a pseudo sistemi operativi real-time (es. FreeRTOS, cfr. sez. 8).

Date le ridotte dimensioni della SRAM e della Flash, si prediligono i sistemi minimali ai sistemi operativi completi, massimizzando l'efficienza computazionale del dispositivo, ed eseguendo solamente il codice strettamente necessario.

[8] In questo contesto si riferisce alla possibilità di partire dal solo codice sorgente con a disposizione il BSP (Board Support Package).

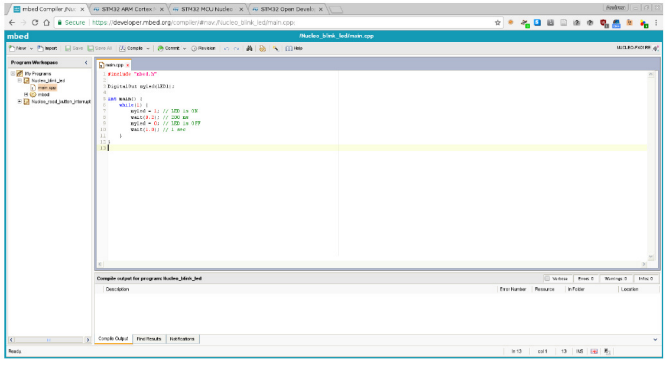


Figura 4.3: Schermata dell'IDE web di Mbed

Ambito applicativo

La famiglia di MCU STM32, così come alcuni dei prodotti dei principali concorrenti (serie MSP di Texas Instruments, Kinetis di Freescale) trovano ampio utilizzo in settori come il comparto automotive, il settore biomedicale e l'ambito militare. A differenza di quanto avviene in altri settori, la scelta di usare una MCU al posto di un SoC non è legata strettamente a semplici fattori di costo (che pur influenzano in fase di progettazione) e talvolta nemmeno legata a fattori di consumo energetico.

L'estrema semplicità architetturale, l'elevato grado di personalizzazione e la possibilità di poter lavorare senza particolari problemi con sistemi *OS-less*[9], consentono alle aziende di poter sviluppare hardware e software in grado di superare i rigidi protocolli di certificazione a cui sono sottoposti i prodotti delle sopracitate categorie (automotive, biomedicale, militare). In molti casi infatti si tratta di sistemi che non devono svolgere gravosi compiti computazionali, ma che devono essere certificati e testati in modo tale da fornire la massima affidabilità: nel settore biomedicale, in particolare per apparecchi che devono operare per il monitoraggio dei parametri vitali dei pazienti, ogni singola riga di codice deve seguire delle regole specifiche di programmazione, interazione e documentazione[37]. Otte-

[9]Privi di un sistema operativo completo.

nere un risultato analogo con software "Linux oriented" rappresenterebbe uno sforzo mastodontico, antieconomico e poco produttivo.

4.7.4 ESP8266

- MCU Tensilica Xtensa LX106 a 80MHz
- 160 kb di RAM (96 kb dati, 64 kb istruzioni)
- 802.11 WiFi b/g/n
- I2S, SPI, I2C, 10 bit ADC, e 16 GPIO
- UART[10]

L'ottimo rapporto prezzo/prestazioni (ad oggi è possibile trovare kit ESP8266 al di sotto dei 5 dollari) ha reso molto popolare l'ESP8266, sia in ambito hobbistico che in quello professionale. Molti anche gli ambienti di sviluppo disponibili per questa board.

4.7.5 Ci40

La Ci40 rappresenta un board nata specificatamente per il mercato IoT, con particolare attenzione alla connettività e al risparmio energetico. Si può notare facilmente come non sia presente al suo interno alcun tipo di elemento legato all'uso di intefaccie grafiche o HMI (Human-Machine Interface). Caratteristiche:

- Processore: cXT200 Mips Interaptiv
- 256MB RAM
- RJ45 Ethernet port
- USB port
- 2 MB (16Mbit) NOR Flash
- 512MB (4Gbit) NAND Memory
- MicroSD card slot
- 2 MikroBUS Adapter
- Debug USB

Caratteristiche del cXT200

- MIPS interAptiv dual core 550 MHz

[10]UART è l'acronimo di Universal Asynchronous Receiver/Transmitter, un device seriale asincrono con velocità e formato dati modificabili.

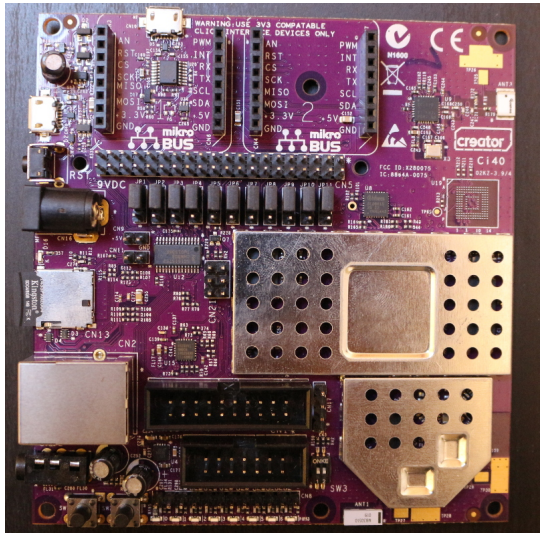


Figura 4.4: Ci40

- 2 x 32/32 kb L1 cache
- 512 kb L2 cache
- Floating point unit
- 802.11 AC 2x2 Wi-Fi
- 802.15.4 6LoWPAN
- Bluetooth 4.1 e BTLE (Low Energy)
- I2C, I2S, SPI, GPIO, Serial

4.7.6 BeagleBone Black

Ricordando quanto già evidenziato in precedenza, una eval board deve essere un buon compromesso tra flessibilità di utilizzo, semplicità di programmazione e debug, mantenendo la possibilità di mostrare tutte le capacità del SoC. Tutto ciò che si trova al di fuori del SoC è aggiunto per soli motivi pratici e dimostrativi.

Caratteristiche:

- Processore: TI AM335x Sitara 1GHz ARMő Cortex-A8
- 512MB DDR3 RAM

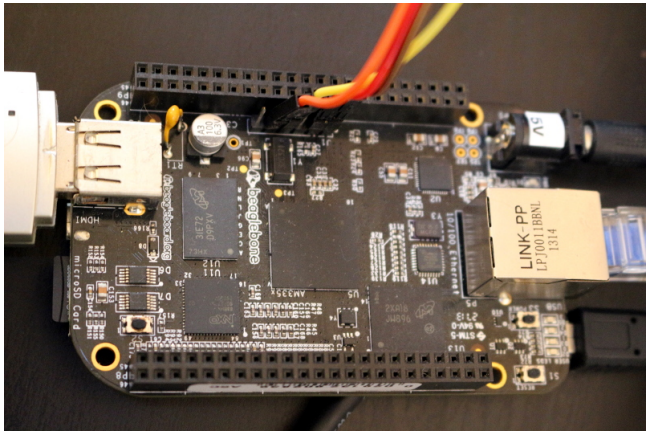


Figura 4.5: Una BeagleBone Black collegata

- 4GB 8-bit eMMC on-board flash storage
- USB client
- USB host
- Ethernet
- HDMI
- 2x 46 pin headers

TI AM335x Sitara

Specifiche tecniche[32]

- CPU: Arm Cortex-A8 32 Bit RISC 1Ghz
- NEON™SIMD Coprocessor
- L1 Cache (32 kb Data, 32 kb Instruction) Parity single error
- L2 Cache 256 kb ECC
- 176 kb BootROM
- JTAG, cJTAG (Boundary Scan, IEEE1500)
- mDDR2,DDR2,DDR3 (200-800 MHz)
- PRCM Module
- RTC
- CryptoHardware Accelerators (AES, SHA, PKA, RNG)
- PowerVR SGX530

- General Purpose Memory Controller (8/16 bit)
- 2 High Speed USB2 OTG port
- 2 Gigabit MAC Ethernet (10/100/1000Mbit) MII, RGMII, RMII, MDIO
- 6 UART (1 full-modem)
- 3 MMC,SDIO,SD port
- 3 I2c, 2 McSPI, 128 GPIO, 1 ADC (12Bit, 200K samples), 3 PWM
- LCD Controller (2048x2048)
- Package 324Pin

A titolo di esempio, verranno mostrati i consumi[11] di un SoC di questa fascia.

Consumi[33]:

- VMIN: 0.95V a 300 MHz
- VMAX: 1.325 V a 1 GHz
- Potenza a 600 MHz in mW:
  min 553.9 (LinuxPSP)
  max 823.21 (3D)
- Potenza a 1 GHz in mW:
  min: 780.54 (Idle)
  max: 1142.02 (Dhrystone)

Configurazioni alternative

Il SoC consente effettivamente di poter utilizzare tutte le specifiche elencate sopra, ma non contemporaneamente. Molti pin di I/O del SoC sono "multiplexed", ovvero sono condivisi tra più possibili configurazioni. Se ad esempio si usa l'LCD controller, l'Ethernet ed un MMC il numero di GPIO disponibile sarà sensibilmente ridotto. In questo, come in molti SoC, esistono quelle che vengono chiamate le "Alternative Configuration" che indicano l'uso che viene fatto dei singoli I/O in base alla configurazione scelta.

Apparentemente una simile scelta potrebbe risultare illogica, ma bisogna sempre ricordarsi che questo tipo di chip è pensato

[11] I consumi sono mostrati sulla base di specifici casi d'uso: Idle e LinuxPSP rappresentano casi a bassa richiesta computazionale, Dhrystone e 3D invece sono test per valutare il massimo delle performance.

per rispondere ad esigenze embedded, quindi con dei forti limiti dal punto di vista dei consumi ma anche delle dimensioni stesse del chip. Inoltre, l'utilizzo di tutte le possibili porte di un SoC è alquanto raro; usando i pin multiplexed, si può scegliere la configurazione adatta alle esigenze ingegneristiche, mantenendo basse le dimensioni di stampa e disabilitando ciò che non sia necessario. Tale scelta impatta positivamente sia sul consumo energetico, sia nella complessità di progettazione del PCB finale del prodotto.

In base alla versione del silicio, il Sitara può essere testato per diversi range di temperatura: commerciale (0-90C), industriale (-40C,90C), estesa (-40C,105C)[34]. Sempre in base alla versione del silicio e al distributore il prezzo può variare tra i 10 ed i 14 dollari al pezzo (per ordini superiori alle 1000 unità).

4.7.7 PIC32

PIC32 è la famiglia di microcontrollori a 32 bit di MIPS Technologies, il top gamma derivato dall'originale PIC degli anni '70. L'architettura della memoria è di tipo Harvard (memoria dati e programma separate), le istruzioni sono in numero limitato e di dimensione fissa (quasi tutte "costano" un solo ciclo di clock), tutte le locazioni in RAM fungono anche da registri, dispongono di cache memory, hanno clock elevati (da 80 MHz in su), supportano debugging via JTAG, forniscono connettività USB, Ethernet, ecc.

L'ambiente di sviluppo visuale si chiama "MPLAB X IDE" e deriva da Netbeans (Oracle) mentre quello più tradizionale "MPLAB XC32/32++". I linguaggi di programmazione sono C e C++.

4.7.8 NodeMCU

NodeMCU[12] è un progetto molto interessante basato su hardware ESP8266. Si tratta di un *firmware* corposo da caricare, via USB o seriale, su una board ESP8266 (sono supportate molte

[12] `http://nodemcu.com/index_en.html` e `http://github.com/nodemcu/nodemcu-firmware`

versioni). Una volta caricato, la board diventa una piattaforma di sviluppo (sebbene un po' goffa, meglio editare i file esternamente e caricarli una volta pronti) e runtime che permette l'esecuzione (anche interattiva) di script LUA[13].

Fornisce una *shell* interattiva per cui è possibile connettersi (USB, seriale, rete) al NodeMCU e interagire in LUA digitando comandi in modalità cosiddetta REPL (*Read Eval Print Loop*).

4.8 Implementazione Hardware

L'implementazione dell'hardware in quanto tale non rientra tra gli scopi principali di questo testo, ciò nonostante vi sono alcune scelte hardware che si riflettono significativamente sulle scelte di sviluppo software, e quindi non trascurabili.

SoC, Licensing e Fab

Dall'esperienza comune di uso di un computer domestico o di una postazione di lavoro, si è portati a credere che il processore sia solitamente progettato, costruito e commercializzato dalla medesima azienda, e lo stesso si può presupporre per tutte le singole periferiche. Nel mondo delle MCU ed ancor di più nei SoC la situazione è ben diversa. Osservando le specifiche di un SoC si possono reperire tutte o quasi le periferiche del dispositivo (che prendono il nome di IP), oltre ovviamente all'unità di elaborazione (il core).

Mentre il core è l'unità che implementa l'architettura, già discussa in questo paragrafo, le IP[14] (*"Intellectual Property", letteralmente le proprietà intellettuali* sono i singoli dispositivi del SoC all'infuori del core (es. audio, gpu, networking, bus, I/O e via dicendo). Si indicano come proprietà intellettuali in quanto esse non sono vendute come componenti fisiche, bensi

[13]LUA (`https://www.lua.org/`) è un linguaggio interpretato, pensato per essere leggero ed efficiente, supporta paradigmi procedurali, a oggetti e funzionali. È "dinamicamente tipato" e gestisce la *garbage collection*. Viene utilizzato ad esempio nel sistema di gestione via web di OpenWrt LuCI (`https://wiki.openwrt.org/doc/techref/luci`).

[14]Da non confondere con *Internet Protocol*

come progetti completi della singola unità che può essere implementata su silicio da aziende terze. Dando una rapida occhiata al mercato, si può notare facilmente quanto sia raro che core e *IP* vengano fornite dalla medesima azienda. All'interno del medesimo SoC convivono e cooperano elementi progettati da aziende differenti. Le *IP* sono state selezionate in base alle loro caratteristiche tecniche ed al loro collocamento sul mercato. Il prezzo assume un'importanza vitale, ed anche una differenza apparentemente irrisoria, ad esempio uno scarto di 0.10$ a unità, può risultare determinante su grandi numeri.

Inoltre, parlare di un SoC ARM o MIPS è in generale poco corretto, in quanto queste aziende si limitano a progettare il core ed alcune delle IP ma, al netto di alcune rare eccezioni, non forniscono il prodotto ultimato. Una volta progettato il core infatti, vendono i progetti, il software, la documentazione e l'assistenza tecnica in licenza ai produttori dei SoC, che in tal modo potranno comporre il loro sistema finale, combinando le ulteriori IP in base alle varie esigenze.

A loro volta, anche le IP possono essere vendute in licenza, fornendo un servizio simile a quello offerto da chi licenzia il core, ma senza effettivamente produrre fisicamente il componente in questione.

Infine, una volta che un'azienda ha composto e progettato il SoC, unendo il core e le singole IP, è alquanto raro che sia la medesima azienda a trasformare tale progetto nel prodotto finale.

A parte alcune eccezioni eccellenti, la stragrande maggioranza delle aziende non dispone della cosiddetta "Fab"[15]. Tali aziende, definite "Fabless (senza Fab) si rivolgono a produttori terzi.

Sopra abbiamo parlato genericamente di *licensing*, senza specificare però che possono essere di due tipi distinti:

Licenza Core: vengono licenziati i progetti e gli strumenti per integrare il core (o la IP) all'interno del proprio SoC, ma senza il permesso di apportare modifiche di nessun tipo al core stesso.

[15]Una "Fab" in questo contesto è il luogo dove vengono stampati i chip.

Licenza Architetturale: vengono forniti gli strumenti e la possibilità di apportare modifiche all'interno del core. In generale assai più costoso sia dal punto di vista delle licenze sia dai costi di sviluppo.

In base a quanto visto sin ora, ed in considerazione del fatto che sia le *IP* che il core possano essere "uniti" per possibilità ed esigenze ingegneristiche in modi differenti, è facile comprendere quale possa essere l'estrema variabilità strutturale dei vari SoC: facendo le dovute eccezioni, è alquanto improbabile una compatibilità totale dal punto di vista software. Questo modello di business ha permesso un rapido sviluppo di queste tecnologie, oltre che offrire un'estrema flessibilità progettuale, fondamentale per venire incontro alle esigenze dei singoli produttori. La possibilità di rivolgersi a *Fab* esterne ha di gran lunga facilitato l'ingresso sul mercato di numerose aziende che altrimenti non avrebbero potuto permettersi di produrre in proprio[16]. Inoltre, la possibilità di poter comprare dei "progetti" e dei *layer* software già pronti per essere adottati, ha ridotto di gran lunga quelli che erano i "Time to Silicon"[17] ed il "Time to Market"[18]. Bisogna inoltre considerare che, come avviene nel mercato dei tablet e degli smartphone, il SoC, a sua volta, potrebbe essere rivenduto ad un produttore terzo, per essere installato su un dispositivo completo, giungendo così ad un prodotto finale.

L'estrema flessibilità ingegneristica e le forti esigenze di ottimizzazione hanno avuto come conseguenza diretta lo sviluppo di software *Open Source*[19], ed in particolare dei sistemi GNU/-Linux (o derivati), tanto da rendere questi sistemi le piattaforme di riferimento e sviluppo.

[16]Lo sviluppo e il mantenimento della produzione del silicio di nuova generazione ha dei costi elevatissimi difficilmente sostenibili da piccole e medie imprese.

[17]Il tempo per arrivare dal progetto al silicio.

[18]Il tempo complessivo che intercorre tra l'inizio della progettazione e l'arrivo del prodotto finito sul mercato.

[19]Nonostante vi siano differenze filosofiche, ai fini di questo testo le licenze "open source" possono essere assimilate a quelle "libere".

4.8.1 Form Factor

Nella progettazione di un dispositivo embedded è importante considerare le dimensioni ed il peso di un oggetto: volendo realizzare un oggetto portabile è necessario assicurarsi che esso non sia ingombrante e non sia pesante, fattori non sempre influenti in caso di dispositivi fissi. In tal senso va considerata la necessità o meno di una batteria (cfr.3.3.6), che sia capace di alimentarlo per un tempo commisurato allo scopo, ma che al contempo possa preservare per quanto possibile la portabilità ed il costo.

Stabilito il consumo del dispositivo finale (operazione non banale, raramente paragonabile a quello della evaluation board), si può scegliere la dimensione della batteria da utilizzare anche in relazione al PCB finale.

4.8.2 PCB

In una evaluation board sono presenti un gran numero di componenti non necessari per il singolo progetto. Ciò concede facoltà di rimuovere i componenti "in eccesso" dal punto di vista hardware, liberare delle porte di I/O e realizzare un PCB più compatto ed ottimizzato, sia per quanto concerne gli spazi che per il consumo.

Questa scelta si riflette anche sulla compilazione del kernel e dei singoli tool software. Prendendo ad esempio un progetto basato su SoC am3358 (ved. 4.7.6) ed utilizzando come base progettuale la BeagleBone Black vista in esempio (ved. 4.7.6), eliminando l'uscita HDMI dal sistema finale e riconfigurando il cosiddetto device-tree (vedere sezione 6.2.2) si liberano un gran numero di pin utilizzabili per altri scopi. Rimuovendo l'uscita video, potrebbero risultare inutili sia i driver dell'acceleratore grafico (quindi disattivabile sul SoC, garantendo un discreto risparmio energetico) sia tutto il software relativo all'interfacciamento grafico (alleggerendo molto il peso del sistema finale).

C'è da considerare anche il fatto che, qualora si stia utilizzando un SoC dotato di GPU in un progetto che non necessita di grafica, probabilmente vi è un errore a monte nella scelta del SoC da utilizzare.

L'organizzazione degli elementi su un PCB è argomento complesso ed articolato che, per quanto interessante, non sarà tema di questa trattazione. Non è infrequente però che molti piccoli produttori prendano spunto dal layout della evaluation board per realizzare la propria, apportando il minor numero di modifiche necessario.

4.8.3 Costo

Se dal punto di vista accademico si tende in genere a trascurare il fattore costo, dal punto di vista produttivo e commerciale esso rappresenta un fattore vitale.

Riuscire ad eliminare componenti non necessarie, o a selezionare componentistica più economica per il prodotto finale, può fare la differenza tra un prodotto in produzione ed uno rimasto sulla carta. Tutto questo però deve essere pensato e calcolato in un giusto compromesso con l'efficienza termica ed elettrica delle componenti e, cosa non secondaria, con la loro affidabilità. A partire dalla scelta del SoC infatti, si cerca il miglior compromesso tra le caratteristiche tecniche offerte ed il progetto finale, minimizzando gli sprechi. Tali considerazioni, unite a fattori di affidabilità e di revisioni del silicio spiegano come talvolta, progetti apparentemente non vincolati strettamente a ragioni di prezzo, siano sviluppati su piattaforme più datate e meno performanti. Bisogna inoltre considerare che per quanto riguarda alcuni dispositivi, oltre al costo di produzione fisico, risultano considerevoli anche le *royalty* (ovvero diritti economici dell'azienda sviluppatrice delle singole IP).

Capitolo 5

Memorie, I/O e comunicazione

5.1 Memorie

Il termine "memoria" è utilizzato genericamente nel mondo informatico per indicare una "spazio" dove memorizzare permanentemente o temporaneamente dei dati o codici. In realtà il significato di questo termine varia estremamente al variare del contesto. In tal senso, l'uso del termine "memoria" a se stante è sempre giusto e contemporaneamente sempre impreciso. Sono molte infatti le caratteristiche che le differenziano: le tecnologie realizzative, le destinazioni d'uso, le velocità, le quantità complessive di spazio, il tipo di interfacciamento e molto altro ancora.

Il primo elemento di divisione tassonomica è la loro persistenza: si parla infatti di memorie **volatili**, ove il dato immagazzinato non rimane memorizzato per lunghi periodi, e di memorie **persistenti** in cui il dato "dovrebbe" rimanere fino ad esplicita cancellazione.

Un altro elemento discriminante nella classificazione delle memorie è il tipo di accesso. Si distinguono le seguenti modalità di accesso:

Memoria ad accesso casuale (RAM): volatile (ma non sempre) e generalmente "veloce", di costo "elevato", in grado di garantire il medesimo tempo di accesso per qualsiasi porzione di essa. Per queste sue caratteristiche è generalmente associata alla memoria di sistema (nei comuni PC è presente nei vari moduli SIMM, DIMM, SODIMM, etc.);

Memoria ad accesso diretto: persistente e generalmente "lenta", ma con un costo relativo più basso delle RAM, offre tempi variabili di accesso in base all'indirizzo di memoria a cui si vuole accedere. Di solito utilizzata per memorie di massa (ad esempio gli hard disk, floppy, dischi ottici);

Memoria ad accesso sequenziale: Molto lenta, persistente, non particolarmente costosa, tempo di accesso molto variabile in base alla posizione attuale di lettura/scrittura. Utilizzata nei sistemi di backup a nastro.

Si badi bene che i concetti di "veloce/lento" e "costoso/economico" sono da intendersi esclusivamente come indicativi. Dall'esperienza quotidiana impariamo che il costo per Mbyte di una RAM (es DDR) è di gran lunga più alto rispetto a quello di un hard disk. A tale differenza di prezzo corrisponde una pari differenza di performance: i tempi di accesso, lettura e scrittura di una RAM sono molto inferiori (e quindi migliori) rispetto a quelli di un hard disk. Per simili risultanze bisogna ovviamente comparare elementi appartenenti alla medesima generazione.

Semplificando molto il concetto, si può dire che all'interno di un sistema embedded è importante poter identificare due memorie fondamentali; la memoria di sistema e la memoria di massa.

5.1.1 La memoria di sistema

Memoria di sistema (o anche memoria primaria) è un termine generico utilizzato per identificare una memoria volatile ad accesso veloce usata dall'unità centrale per il caricamento di dati ed istruzioni da poter processare. In genere, come memorie di sistema vengono utilizzate le RAM (Random Access Memory o Memoria ad accesso casuale). Come già detto per il termine "memoria", anche il termine RAM, se pur usato propriamente,

potrebbe essere sin troppo generico. In base alla tecnologia è possibile distinguere tre tipologie principali di RAM:

SRAM: Definite anche come Static RAM, si tratta di una memoria statica (che **non** necessita di *refresh*[1]) ma comunque volatile, generalmente molto veloce ma anche molto costosa. Dato il costo elevato (sia in termini di spazio che economici), è utilizzata generalmente per delle memorie "cache", ma può essere usata anche in alcuni sistemi embedded di piccola taglia come memoria per esecuzione, o come memoria di scambio tra sottosistemi differenti. Dal punto di vista energetico e termico ha delle ottime prestazioni;

DRAM: Acronimo di Dynamic RAM, si tratta di una tecnologia per memorie volatili assai più economica e semplice delle SRAM (raggiungendo quindi maggiori densità di capacità), ma che richiede il *refresh* (vedi sopra) del dato immagazzinato. Con il termine DRAM si intende la variante asincrona di questo tipo di memorie, ovvero quelle che non devono essere necessariamente sincronizzate con il *clock* del sistema;

SDRAM: E la variante sincrona delle DRAM (sta per Syncronous Dynamic RAM), la cui velocità è quindi collegata al *clock* del elaboratore, ampiamente utilizzata nei sistemi desktop/workstation (ad esempio la famiglia delle DDR[2]), da tempo viene utilizzata in misure e velocità diverse anche nei sistemi embedded di vario genere.

A seconda del sistema in utilizzo velocità e quantità possono variare sensibilmente.

MMU

L'unità di gestione della memoria (Memory Management Unit, o MMU) è una particolare unità che si occupa prevalentemente

[1] Il termine *refresh*, riferito alle memorie ed in particolare alle DRAM, indica l'operazione di "rivitalizzazione" (lettura e riscrittura immediata) dei contenuti in memoria per non perdere l'informazione mantenuta nei "condensatori" che compongono le celle. Per maggiori informazioni consultare la sezione 3.3.5.

[2] DDR sta per "Double Data Rate".

di tre compiti:

Traduzione indirizzi virtuali:
alcuni sistemi operativi non concedono l'utilizzo degli indirizzi fisici della RAM da parte delle applicazioni (vedere 6.2.2). In tal caso, gli indirizzi utilizzati sono insiemi separati di indirizzi virtuali (o logici). La MMU fornisce un servizio di traduzione tra indirizzi logici e fisici, in modo che il sistema possa offrire contestualmente l'astrazione necessaria e l'accesso e l'uso della memoria;

Protezione della memoria: il meccanismo di traduzione della memoria fa in modo che processi diversi non possano accedere alle medesime aree di memoria;

Gestione della cache: in alcuni casi la MMU si occupa anche di gestire la memoria cache.

5.1.2 La memoria di massa

Le memorie di massa (inteso come spazio di conservazione) di dati persistenti, possono essere suddivise in base al tipo di supporto fisico utilizzato. Si possono infatti distinguere i seguenti supporti:

ROM: Memorie in sola lettura, informazione contenuta nei transistor (o in alcuni primi esemplari nei diodi);

Flash Memory: memorie NAND/NOR e derivati (MMC, SD, USB Key, eMMC), dischi a stato solido;

Supporto Magnetico: Floppy Disk, Hard Disk, Nastri magnetici;

Supporto Ottico: CD, DVD, Blue-Ray;

Con l'eccezione di alcuni casi particolari, lavorando su sistemi embedded, raramente si avrà a che fare con supporti magnetici e supporti ottici che quindi non rientrano nella trattazione specifica di questo testo. Assai più probabilmente, si dovrà interagire direttamente con memorie di tipo Flash e memorie ROM (ROM/PROM/EPROM/EEPROM).

ROM

Fin dai primordi delle tecnologie embedded, si è sempre fatto un utilizzo massivo delle cosiddette memorie ROM (*Read Only*

Memory o memoria in sola lettura). In principio si trattava di memorie in cui le informazioni erano *hard-wired* ovvero codificate direttamente sul silicio, e non riscrivibili successivamente. Col progredire della tecnologia però, le ROM si sono evolute in diverse forme, aggirando, almeno in parte, il limite della impossibilità di riscrittura.

PROM (*Programmable read-only memory*): Presenti sin dal 1956, le PROM sono memorie programmabili solo una volta tramite l'uso di corrente elettrica ad alto potenziale (da cui il termine "*burn the ROM*": "bruciare" la ROM). Una volta scritte non possono essere riscritte in nessun modo;

EPROM (*Erasable Programmable read-only memory*): A differenza delle PROM, le EPROM sono riscrivibili, ma solo un numero limitato di volte (in molti casi si parla di circa un migliaio di volte). La programmazione avviene previo "azzeramento" tramite luce ultravioletta, per questo motivo è possibile riconoscere questi memorie dalla presenza di una cosiddetta "finestrella" di cancellazione (una parte del *package* è ricoperta in vetro per esporre alla luce la superficie del chip);

EEPROM (*Electrically Erasable Programmable read-only memory*): Queste memorie sono cancellabili e riscrivibili elettricamente un numero limitato di volte.

Le ROM possono essere considerate estremamente veloci in lettura, ma, nel caso delle EEPROM, decisamente molto lente in scrittura, limitando spesso il loro ambito applicativo ad usi di *bootstrap* (avvio del sistema) e di funzioni base del sistema. Nonostante molte tipologie siano cancellabili, non è infrequente che, in dispositivi completi, per motivi legati alla produzione o alla sicurezza, siano protette dalla scrittura per evitare manomissioni o utilizzi non previsti dal costruttore del prodotto.

Non tutti le fonti[64] sono concordi nel classificare le Flash Memory separatamente dalle ROM, come fatto in questo testo, ma, in favore di semplicità e chiarezza e per esaltare le enormi differenze applicative e di sviluppo, è stato scelto di porle in un'altra categoria.

Flash Memory

Le Flash Memory, di diretta derivazione delle EEPROM, si distinguono da queste ultime per la loro capacità di essere cancellate e riscritte a blocchi o unità più piccole e non interamente. Nate agli inizi degli anni 80, hanno nel tempo trovato grande applicazione nel mercato embedded come in quello desktop/-workstation e di recente anche server (basti pensare all'utilizzo di disco a stato solido in molte *"server farm"*[3]). Sebbene questa tecnologia sia decisamente migliorata negli ultimi anni, è giusto ricordare che uno dei limiti maggiori per questo tipo di memoria è che un blocco può essere "scritto" un numero limitato di volte, rendendole, fino a pochi anni fa, inadatte a contenuti che dovessero essere aggiornati spesso.

In base al tipo di collegamento delle celle, varia la modalità operativa della memoria, che può comportarsi in modo simile ad un "gate" NAND o ad uno NOR. Sulla base di questo principio si distinguono due tipi di memoria flash:

NAND Memory: I transistor sono collegati in un modo da creare un collegamento in serie tra le celle. Tale tipo di collegamento permette di ridurre molto lo spazio occupato su silicio. Sono memorie di gran lunga più economiche di quelle NOR, ed unendo questo al minor spazio occupato, si può capire facilmente come le NAND siano generalmente reperibili di dimensioni sensibilmente maggiori e a costi inferiori rispetto alle NOR. La struttura di una NAND è organizzata a pagine (generalmente di dimensioni che oscillano tra i 512 ai 4096 byte) riunite in blocchi di un numero fisso di pagine (da 32 a 128 pagine). Le memorie NAND sono quelle maggiormente reperibili nei dispositivi di storage removibili;

NOR Memory: Tutte le celle sono collegate a massa ed in parallelo alla bit line, comportandosi come un NOR Gate. In questo modo, ogni cella è leggibile e scrivibile singolarmente, ma occupa più spazio rispetto ad una cella di una memoria NAND. Le NOR sono state progettate per

[3] Con il termine *Server Farm* ci si riferisce genericamente ad un luogo dove operano numerosi elaboratori di potenza elevata per gli usi più svariati. Talvolta il termine è intercambiabile con la dicitura *"Data Center"*.

prendere il posto delle EEPROM, quindi per contenuti che dovevano essere letti di frequente, ma scritti di rado, per questo motivo, è stato preferito ottimizzarle per le lettura, con un notevole decadimento di performance sulla scrittura. Una caratteristica delle NOR è che, nei processori che lo supportano, eventuale codice oggetto contenuto su di esse può essere eseguito sul posto (*Execution in Place*, XIP), ovvero senza il bisogno di essere caricato su una memoria esterna. Ciò la rende ideale nelle fasi di *boot*, dove la memoria di sistema potrebbe non essere ancora accessibile). Inoltre accade di frequente siano messe in modo da essere completamente mappate sulla memoria di sistema (*memory mapped*). Le NOR possono anche essere programmate per fornire un "*random-access*", come le RAM.

In base alle loro caratteristiche si può intuire come sia assai più probabile trovare una memoria di tipo NAND come storage principale di un sistema embedded (basti pensare alla memoria di un telefono) ed una memoria NOR utilizzata per funzioni a basso livello e di boot.

Le memorie NAND sono spesso "nascoste" da alcuni specifici *controller* (dispositivi con funzioni a basso livello che ci mostrano la memoria NAND con un livello di astrazione più alto) sotto forma di chiavi USB, schede SD/MMC o dispositivi con eMMC. Questi controller, oltre ad assicurare il cosiddetto *wear-leveling*[4], permettono l'utilizzo di filesystem ideati per dispositivi a blocchi quali gli hard-disk (FAT, NTFS, ext2/3/4). Può succedere però che in alcune board, per motivi pratici o economici, tali controller vengano rimossi; in tal caso è necessario utilizzare dei tipi speciali di filesystem, ideati per gestire strutture basate sui cosiddetti "EraseBlock", come ad esempio:

jffs1/2: un "Log Structured Filesystem" disegnato per lavorare su NAND, ha alcuni limiti strutturali legati alla necessità di eseguire le fasi di *clean* ogni qual volta montato;

[4]Sistema per evitare che una memoria sia scritta sempre sullo stesso blocco. Senza questo sistema si potrebbe verificare la fine (per "consumo") prematura del blocco stesso e di conseguenza la perdita di affidabilità della memoria.

yaffs1/2: concettualmente simile ai filesystem JFFS, con il quale condivide alcuni limiti. Molto utilizzato dai produttori dei primi smartphone Android;

ubifs: di recente adozione, Ubifs introduce molte ottimizzazioni legate al "wear-leveling" e con minori tempi di accesso in fase di mount.

Il comportamento dal punto di vista fisico/elettrico di queste memorie è senza dubbio materia affascinante, ma va ben oltre gli scopi di questa trattazione.

5.2 Bus e periferiche

In questa sezione verrà fornita una breve panoramica introduttiva sui bus e le periferiche utilizzate in dispositivi embedded, e comunemente presenti sotto forma di IP4.8 all'interno dei vari MCU e SOC. Alcuni di questi dispositivi sono gli stessi utilizzati in ambiente desktop/workstation (come ad esempio Ethernet, USB, IEEE802.11 WiFi, SD/MMC Controller), altri invece, come quelli elencati in seguito, sono assai più utilizzati in ambito embedded (ciò non vuol dire che non vengano utilizzati anche altrove).

5.2.1 RS-232 Serial port

In gergo normalmente, riferendosi alle porte seriali RS232 si usa genericamente il termine "seriale"; tale termine, per quanto comunemente accettato è di per se errato, in quanto identifica solamente la modalità di comunicazione, senza però fornire alcun tipo di informazione sul protocollo di comunicazione utilizzato. In questo caso invece faremo espressamente riferimento allo standard RS232 e alle sue implementazioni.

Lo standard di riferimento per lungo tempo è stato il RS-232-C, risalente al 1969, evoluzione del primo standard RS-232 presentato nel 1962. Ad oggi, l'ultima versione di questo standard è marchiata con il nome TIA-232-F, anche se i cambiamenti nei confronti della versione "C" sono limitati quasi solo ai segnali e al *timing* dei dispositivi.

Figura 5.1: Connettori DB9 e DB25 sul retro di un terminale "vintage"

Le porte seriali, nel corso degli anni, sono state utilizzate come interfacciamento per un grandissimo numero di dispositivi, quali terminali, stampanti, mouse, tastiere, scanner, rete, e fino a non molto tempo fa anche modem. Con l'avvento delle porte USB, le porte seriali sono progressivamente sparite dall'hardware desktop/workstation, ma sono rimaste sia in ambito embedded che in quello server, dove ad esempio l'uso di un terminale grafico a schermo non è sempre possibile o comunque è poco conveniente[5].

Per fare una grossolana semplificazione, si può dire che su una linea seriale venga trasmesso un solo bit alla volta, e che il valore di questo singolo bit sia stabilito sulla base della differenza di potenziale rispetto alla massa (GND). Nel caso di valori da +3V a +15V, si assume che il bit sia fissato a 0, se invece il voltaggio e nell'intervallo tra -3V e -15V il bit è impostato ad 1. Da notare che l'intervallo che va da -3V a +3V è visto come segnale non valido (viene quindi ignorato), per eliminare il "rumore di fondo". Ad oggi tale precauzione può sembrare eccessiva, così come può sembrare esageratamente alto il valore di differenza di potenziale utilizzato, ma ciò risultava necessario molti anni fa, con cablature molto lunghe e con circuiterie energeticamente meno efficaci.

Dato il vasto utilizzo esistono un gran numero di varianti allo standard, sia per quanto concerne i connettori sia per quanto riguarda i cavi. Per semplicità, in questa sezione si utilizzerà come esempio il *pinout* del cavo seriale a 9 pin (figura 5.1):

[5]E molte piattaforme di gestione virtualizzano le console seriali dei server in *farm* incanalandole su protocolli di rete come TELNET o più recentemente SSH.

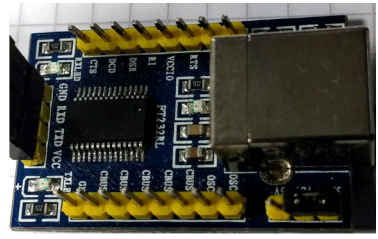


Figura 5.2: Usb2Serial converter, 3V3 e 5V

TxD (Trasmitted Data): Linea per i dati in uscita
RxD (Received Data): Linea per i dati in ingresso
DTR (Data Terminal Ready)
DCD (Data Carrier Detect)
DSR (Data Set Ready)
RI (Ring Indicator): Utilizzato per sollevare un interrupt nel caso il dispositivo stia ricevendo una chiamata in ingresso (modem)
RTS (Request To Send): Controllo di flusso (uscita)
CTS (Clear To Send): Controllo di flusso (ingresso)
GND (Ground): massa.

Questo schema viene spesso definito comunemente con il nome di "seriale completa". In realtà, nei dispositivi embedded delle ultime generazioni, è raro che vengano esposte seriali complete. Nella maggior parte dei casi si tratta di schemi detti a "tre fili" (TxD, RxD e GND) o a "5 fili" (i precedenti più i *flow control* RTS e CTS).

E' inoltre importante tenere presente che al giorno d'oggi è assai raro che si usino seriali standard, più comunemente vengono utilizzate le cosiddette seriali TTL[6]. Il vantaggio di queste seriali sta nel fatto che utilizzano valori molto bassi di differenza di potenziale (0 logico per valori minori 0.8V ed 1 logico per valori tra i 2 ed i 5 volt), e sono assai compatte (di solito sono esportate sotto forma di pin sul circuito stampato). Per poter collegare una seriale TTL con un comune computer Desktop, esistono degli adattatori che prendono il nome di USB2TTL

[6]**T**ransistor-**T**ransistor **L**ogic.

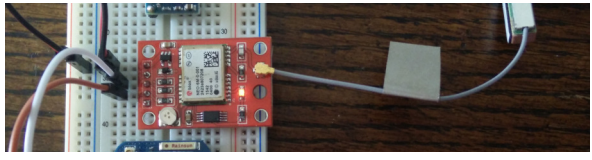


Figura 5.3: Scheda GPS embedded commerciale

(figura 5.2). Tali seriali richiedono la precauzione di verificare quale sia il valore di VCC utilizzato in uscita; in molti dispositivi embedded si utilizzano seriali TTL a 3V3 e a 5V ma non sempre questi dispositivi sono compatibili tra di loro (detti anche "tolleranti"). Utilizzando un adattatore TTL 3V3 su un dispositivo a 5V o viceversa si potrebbero verificare problemi di comunicazione o, in casi peggiori, anche guasti ad entrambi i dispositivi.

5.2.2 NMEA 0183

NMEA (National Marine Electronics Association `https://www.nmea.org`) è un ente di normazione famoso nel mondo embedded per il noto[7] standard NMEA0183. Risale agli anni '80 anche se nel corso del tempo è stato rinnovato e aggiornato a tecnologie più recenti, oggigiorno viene infatti man mano sostituito dal NMEA2000.

E' uno standard sia elettrico (cioè specifica i livelli di tensione da utilizzare sui cavi usati per la comunicazione) sia "protocollare" (si potrebbe dire "applicativo") in senso stretto, cioè specifica il formato della trasmissione per inviare dati "marini", è infatti nato per inviare i dati degli strumenti di bordo quali GPS, bussole, profondimetri, etc. a bordo di un'imbarcazione.

La specifica elettrica descrive un *bus* seriale EIA-422, simile a RS-232 (trattato in Sezione 5.2.1) ma con portata maggiore[8].

[7] Almeno tra gli appassionati di mare tecnologici.

[8] RS-422 infatti usa un segnale elettrico "differenziale", quindi con due cavi, per ogni via (TX e RX) mentre RS-232 usa un solo cavo per via. Ciò gli permette di essere più "robusto" (insensibile al rumore di fondo) e di poter raggiungere distanze maggiori senza perdere informazione.

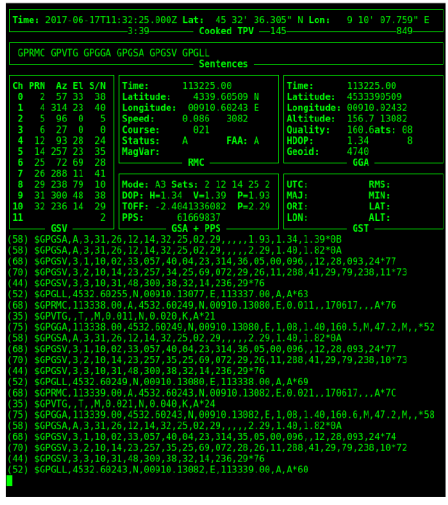


Figura 5.4: Esempio di output NMEA (catturato via `gpsd`)

Attualmente, nel mondo embedded, esistono schede (ad esempio la scheda GPS visibile in figura 5.3) che per semplicità trasmettono utilizzando RS-232 con livelli TTL.

La velocità ufficiale originale è 4800 *baud*[9], oggi portata a 38400 *baud*) in cui un solo nodo (detto *talker*) può trasmettere e molti nodi (detti *listener*) possono ricevere.

I dati vengono trasmessi in ASCII (quindi *human readable*) e sono organizzati in stringhe (messaggi) ben formattate (si veda un esempio in figura 5.4). Le stringhe hanno un formato vincolante definito sinteticamente come segue:

- iniziano tutte con "$" (*start*)
- i vari campi sono separati da ","
- i caratteri CR/LF[10] terminano una stringa/messaggio
- la lunghezza massima è 82 caratteri
- i primi 5 caratteri dopo lo *start* rappresentano il nome del *talker*

Ad esempio la stringa:

[9]Numero di "simboli" trasmessi al secondo.

[10]Carriage Return (ASCII 13), Line Feed (ASCII 10).

```
$GPGGA,123519,4807.038,N,01131.000,E,1,08,0.9,545.4,M
    ↪ ,46.9,M,,*47
```

Rappresenta l'invio, da parte di un GPS, della posizione attuale: il valore 4807.038 è la latitudine[11], N indica Nord, 01131.000 è la longitudine, E indica Est.

Oltre all'uso marino incontriamo l'NMEA nel contesto embedded volendo usare ad esempio le già citate schede GPS (figura 5.3). Per fortuna degli sviluppatori non è necessario scriversi software per il *parsing* delle stringhe dato che ogni ambiente di sviluppo, specie quelli "liberi"[12] e "vivi"[13] come Arduino, forniscono molteplici librerie pronte. In ambiente Arduino è molto usata la libreria TinyGPS (`http://arduiniana.org/libraries/TinyGPS`) che permette di scrivere *snippet* di codice tipo il seguente:

```
// inclusione della libreria
#include "TinyGPS++.h"

// istanziazione del parser TinyGPS
TinyGPSPlus gps;

// collegamento con la seconda seriale
// (la prima serve per la console)
// deve essere collegata alla scheda GPS
SoftwareSerial ss(4, 3); // pin 4=RX, pin 3=TX

...

// ciclo di lettura dei dati seriali
while (ss.available() > 0){
 // il parser legge i dati dalla seriale
 gps.encode(ss.read);
}

...

// ora si possono estrarre le componenti
Serial.print("LAT=");
Serial.println(gps.location.lat(), 6);
```

[11] Da tradurre in gradi, minuti e secondi.
[12] Intesi come Free/Libre/OpenSource software.
[13] Cioè molto diffusi e animati da comunità di sviluppatori attive.

```
Serial.print("LONG=");
Serial.println(gps.location.lng(), 6);

Serial.print("ALT=");
Serial.println(gps.altitude.meters());
```

Che, come il lettore potrà notare, evita in toto l'obbligo di gestire le stringhe `$GP...` esplicitamente.

5.2.3 I2C

I2C (più correttamente I^2C, si legge I-quadro-C o I-square-C), acronimo di *Inter-Integrated Circuit*, è un bus seriale inventato da Philips Semiconductor (ora NXP). Tale bus è stato ideato per comunicazione a breve distanza, e in generale all'interno dello stesso computer o sistema embedded. Infatti, pur fornendo la possibilità teorica di comunicazioni ad alcuni metri di distanza, è assai raro che ciò avvenga nella pratica.

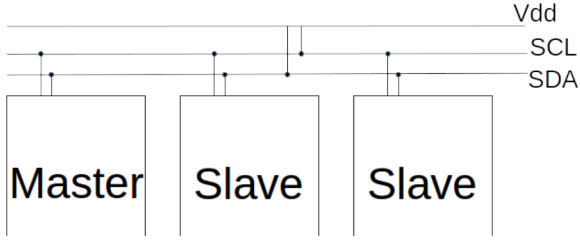


Figura 5.5: I2C bus

Nato nel 1982 con una frequenza di esercizio di 100 kHz, superato successivamente da 400 kHz e 3.4 MHz (rispettivamente dalla versione 1 e 2) e arrivato ad oggi alla versione 6 che da standard può supportare frequenze di 5 MHz. Tali frequenze sono da considerare solo a titolo indicativo, in quanto è comunque possibile, in base alle specifiche del device in uso, variarle in modo arbitrario. Anche dal punto di vista del voltaggio, pur essendo tipicamente reperibili a 3.3 V o 5 V, sono comunque consentite altre modalità operative. Nonostante tutte le possibili variazioni del caso, sono definite alcune velocità di trasferimento dati "usuali":

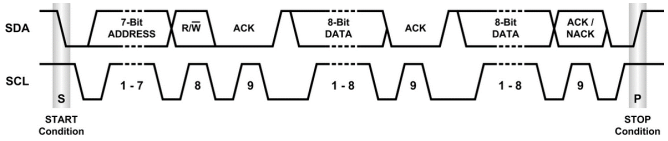


Figura 5.6: Esempio di comunicazione su I2C (da [6])

low-speed mode 10 kbit/s
standard mode 100 kbit/s
Fast mode 400 kbit/s
Fast mode plus (Fm+) 1 Mbit/s
High Speed Mode 3.4 Mbit/s

Le velocità più alte sono disponibili solo nelle ultime versioni del BUS, e sono tipiche dei sistemi embedded.

I2C utilizza due linee bidirezionali (open-drain) note come SDA (Serial Data Line) e SCL (Serial Clock Line) per interconnettere tra di loro alcuni nodi. I nodi all'interno del bus possono avere due distinti ruoli (cfr. figura5.5):

Master: Genera il clock e fa partire la comunicazione con lo slave;

Slave: Riceve il clock e risponde quando indirizzato dal master.

I2C è un bus "multi-master", ovvero dove è possibile la presenza di più nodi master. Inoltre il ruolo master/slave può essere variato tra un messaggio e l'altro.

Nella figura 5.6 è possibile vedere come sia strutturata una comunicazione su I2C: il segnale del clock (SCL) viene abilitato ogni volta che si intenda mandare un singolo bit. Sulla linea dati (SDA), dopo la condizione **S** di *start* (SCL attivo, SDA disattivato) viene specificato un indirizzo di destinazione da 7 bit più un ulteriore singolo bit atto ad indicare se si tratti di una operazione di *read* (lettura) o *write* (scrittura). A questo deve seguire un segnale di ACK[14] da parte dello *slave*. A questo punto avviene la trasmissione del dato vero e proprio, impacchettato in una o più trame da 8 bit l'una, seguite ogni volta da un altro segnale di ACK o NACK[15] sempre da parte dello sla-

[14] Dall'inglese *acknowledgement*, tr. riconoscimento.
[15] Inteso come *not acknowledgement*.

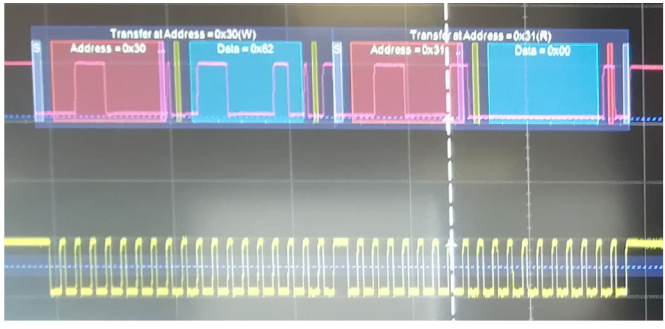


Figura 5.7: Scrittura Registro su I2C vista su oscilloscopio

ve. La fine della trasmissione è marcata dalla cosiddetta "stop condition" **P**, che è l'inverso dal punto di vista dei segnali di quella di start.

Come si può apprezzare in figura 5.7 (visualizzazione su oscilloscopio di un segnale I2C) con riferimento alla linea superiore (SDA), il registro `0x30` viene dapprima scritto e poi riletto successivamente. Apparentemente potrebbe sembrare che il registro letto in seguito sia diverso (dalla configurazione binaria risulta l'indirizzo `0x31`), ma in realtà ciò è frutto del bit di *read/write*: dato che il valore indirizzo è letto ad 8 bit, se si tratta di una operazione di scrittura, l'ottavo bit e impostato a 0, mentre in caso di lettura è impostato ad uno. Da notare come il segnale di CLOCK (SCL), visualizzato nella traccia sottostante, venga abilitato ogni qual volta si trasmetta un bit, sia esso di indirizzo, ACK o dato. Una condizione abbastanza comune, che potrebbe risultare strana ad una utenza non pratica di segnali, è che il segnale SCL è nella modalità definita comunemente *"attivo-basso"*, ovvero che viene considerato attivo quando questo è sul valore elettrico più basso, mentre è considerato non attivo quando è sul valore elettrico più alto. Intuibilmente, la modalità opposta è definita *"attivo-alto"*. Nei comuni bus I2C, il clock è attivo solo in fase di trasmissione.

5.2.4 I2S

I^2S[60], da non confondere con I^2C, è l'acronimo di Inter-IC Sound (si pronuncia I-square-S). Già dal nome è facilmente intuibile che sia stato progettato per "trasportare" dati audio (PCM) e controllare gli apparati relativi alla registrazione e riproduzione del suono all'interno di un dispositivo embedded.

Introdotto nel 1986 da Philips, è rimasto formalmente immutato a partire dal 1996. Semplice dal punto di vista architetturale (un collegamento basato su almeno tre linee), è tutt'oggi molto utilizzato in una moltitudine di dispositivi embedded e mobili (quali ad esempio telefoni e tablet).

Le linee del bus (standard) sono:

Continuous Serial Clock (SCK) noto anche come Bit clock line (BCLK): Ad ogni impulso del bit clock corrisponde un bit sulla linea SD;

Word Select (WS): Noto anche come Left-Right Clock (in base al valore selezione il canale destro (0) o il sinistro (1);

Serial Data (SD): Questo canale può opzionalmente avere più linee, ma in generale lo si considera un cosiddetto "linea multiplexed", in cui, in base al valore del WS e alla frequenza del SCK, vengono trasmessi i dati del canale audio selezionato.

La frequenza del bit clock equivale al prodotto del "sample rate", del numero dei canali e del numero di bit per canale. Un esempio classico è quello del CD, con un "sample rate" di 44.100 Hz, un bit rate da 16bit e su due canali (stereo). Questo ci da complessivamente un valore di $44.100Hz \cdot 16 \cdot 2 = 1411200Hz$ (1.4112 MHz).

Lavorando su un progetto reale, si potrà notare come in alcuni casi, rispondendo a particolari esigenze tecniche, questo standard sia stato implementato con delle modifiche, notando ad esempio, la presenza di un cosiddetto Master Clock (utilizzato per la sincronizzazione di DAC/ADC) o di linee multiplexed per il ritorno.

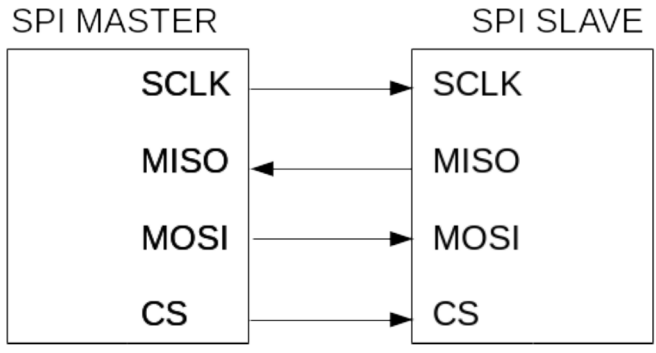


Figura 5.8: Schema di collegamento SPI

5.2.5 SPI

SPI (acronimo di Serial Peripheral Interface)[5] è la specifica di un bus per comunicazione seriale sincrona ampiamente utilizzato all'interno dei sistemi embedded per comunicazioni a breve distanza (all'interno della stessa board) e a velocità relativamente bassa.

Basata su 4 segnali (motivo per il quale viene anche chiamata "4-wire" bus) il cui nome può variare a seconda del costruttore, SPI è un bus di tipo master/slave che, a differenza di quanto già visto con I^2C non è multimaster (quindi un solo master per slave multipli):
SCLK: Serial Clock
MISO: Master Input Slave Output
MOSI: Master Output Slave Input
CS: Chip Select

Come si vede in figura 5.8 il Serial Clock (SCLK o SCK) è un segnale inviato dal master e controlla lo spostamento di bit tra master e slave. Il segnale MISO (SOMI, SDI, SO, DI) è collegato all'input del master e all'output degli slave. Il suo opposto, MOSI (SDO, SIMO, DO, SI) è invece collegato all'uscita del master e all'ingresso dello slave. Infine il CS (o SS, nCS, nSS, STE) è utilizzato per selezionare il chip slave che deve essere abilitato alla comunicazione dal master. In base al tipo di

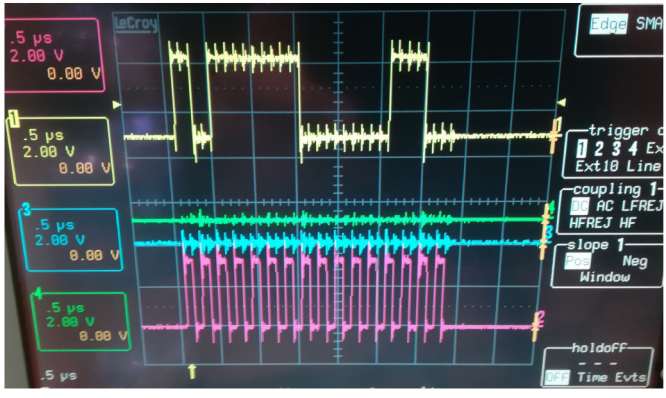


Figura 5.9: Trasmissione 16bit SPI

configurazione, questo segnale può essere unico (collegamento *daisy chain*) o multiplo (uno per ogni slave).

Si osservi ora (figura 5.9) la visualizzazione di una comunicazione SPI su oscilloscopio: partendo dall'alto, sono mostrate in ordine MOSI, MISO, CS e SCLK. Comparato a quanto visto in precedenza con I2C, SPI risulta assai più semplice; quando il clock viene abilitato dal master, vengono trasmessi i bit in modo seriale. In questo caso, sia SCLK, che MISO e MOSI sono impostati come *"attivo-alto"*. Va inoltre aggiunto che, se non specificato da protocolli di più alto livello, o driver specifici per particolari dispositivi SPI, non ci sono né condizioni di inizio e fine trasmissione, né segnali di ACK. Va aggiunto che per quanto riguarda il segnale di *Chip-Select* (in questa configurazione impostato come *"attivo-basso"*), per poterne apprezzare la variazione si dovrebbe allargare di molto la scala dei tempi, in quanto, la sua attivazione/disattivazione avviene rispettivamente molto precedentemente e molto successivamente alla trasmissione vera e propria. Una tale visualizzazione però non avrebbe permesso di vedere gli altri segnali distintamente. Dando una veloce occhiata alla scala dei tempi, molti lettori potranno intuire in modo autonomo quale sia la velocità del clock in uso in questo esempio.

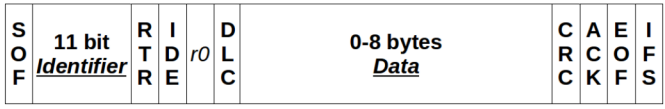


Figura 5.10: Schema di un frame base CAN

5.2.6 CAN-BUS

Il CAN (Controller Area Network), è uno standard di comunicazione (ISO 11898:2015, cfr. [20]) seriale basato sui messaggi sviluppato da BOSCH ed è particolarmente utilizzato nel settore automotive o ambienti industriali. A discapito di velocità massime non particolarmente elevate (1 Mbit/s) e di un raggio di azione limitato (sui 40 m a 1 Mbit/s, ma si può allungare abbassandone drasticamente la velocità) viene prediletto per la sua estrema semplicità di implementazione e per la sua alta affidabilità: in ambienti critici o in cui la comunicazione può essere compromessa da interferenze elettromagnetiche, CAN offre maggiori garanzie sulla trasmissione rispetto ad altri standard. CAN è uno standard master-slave con possibilità di configurazioni multi-master: a differenza di quanto avviene con altri standard di comunicazione, CAN invia piccoli messaggi (e non grandi frame) a tutti i nodi collegati al BUS.

CAN inoltre è un cosiddetto CSMA/CD+AMP (Carrier Sense Multiple Access, Collision Detection and Arbitration Message Priority) o, detto più semplicemente, ogni nodo deve aspettare un tempo prestabilito prima di inviare il proprio messaggio e nel bus vengono rilevate le potenziali collisioni tra i frame, decidendo in modalità bitwise (un confronto bit a bit sui valori di priorità contenuti nell'ID dei singoli messaggi) il messaggio da far arrivare.

In figura 5.10 viene mostrato un esempio di frame il cui contenuto è:

SOF Start of Frame: Inizio del messaggio

11-bit identifier Indica con un numero progressivo il messaggio (o la priorità)

RTR Remote Transmission Request: Se il bit RTR è abilitato tutti i nodi ricevono la richiesta, ma l'ID identifica il nodo.

IDE Identifier Extension: Un ulteriore bit di identificazione
r0 Bit riservato (possibili scopi futuri)
DLC Data length code: 4bit con cui sono indicati il numero di byte trasmessi.
Data 64 bit di campo dati.
CRC Cyclic Redundancy Check: 16 bit di ridondanza ciclica
ACK 2bit per indicare il ricevimento corretto di un messaggio.
EOF End Of Frame: 7bit che indicano la fine del messaggio.
IFS InterFrame Space: 7 bit di cuscinetto tra un messaggio ed un altro.

Oltre al "base frame" esiste anche l'extended frame, che si contraddistingue per avere un ulteriore campo identificativo dopo IDE di 18 bit, l'aggiunta di un campo SRR (Substitute Remote Request) per ragioni di compatibilità con la posizione RTR della frame base, un ulteriore elemento riservato *"r1"* dopo *r0*.

I frame (o messaggi) possono essere di 4 tipi distinti

Data Frame Il messaggio più comune, utilizzato per il passaggio di dati;
Error Frame Messaggio inviato dai nodi che ricevano frame mal formattati;
Overload Frame Simile al frame di errore, inviato dai nodi per prendere tempo tra un messaggio ed un altro;
Remote Frame Inviato per sollecitare l'invio di un messaggio.

5.2.7 Ethernet

Per quanto sia raro ad oggi trovare prodotti finali che contengano anche una cosiddetta "presa ethernet" (oggi identificata per lo più con il connettore RJ-45), eccezion fatta per i dispositivi con un target di rete quali router, IP-camera o simili, è assai comune trovare board di sviluppo/validazione per SoC di fascia alta che la utilizzano. Per questo motivo si eviterà di approfondire sugli aspetti teorici/elettronici, concentrandosi esclusivamente su un approccio più pratico.

E' necessario premettere che, nonostante il nome Ethernet sia estremamente diffuso per identificare questa tipologia di connessione cablata, in realtà questo nome fa riferimento alla prima definizione dello standard, risalente al 1978; tale standard, rea-

lizzato in collaborazione tra le altre da Xerox, Intel e DEC, fu poi successivamente rivisto nel IEEE802.3 del 1985, che pur basandosi sul precedente, ha delle differenze. Ad oggi, i prodotti chiamati Ethernet, sono in realtà dei dispositivi conformi al IEEE802.3 (Per maggiori informazioni sullo standard e sul suo funzionamento, è possibile consultare le pagine del sito IEEE[29].

Dal punto di vista dei sistemi embedded la scheda di rete Ethernet non è considerabile un unico elemento hardware: è assai comune infatti che tale componente sia in realtà diviso in due parti separate e fornite da due produttori differenti:

MAC: Il Media Access Control, comunemente chiamato MAC, rappresenta il livello 2 della pila ISO/OSI (vedere sezione 10).

PHY: Il Physical Layer, o in gergo solo "il fisico", rappresenta il livello 1 della pila ISO/OSI.

Sono abbastanza comuni configurazioni in cui il MAC si trovi all'interno del SoC, mentre il PHY sia esterno, posizionato sulla board. Ciò offre un notevole vantaggio pratico, in quanto consente di utilizzare un SoC dotato di rete durante tutte le fasi dello sviluppo per poi essere rimosso in fase finale di prodotto, rimuovendo solo il PHY, componente assai ingombrante.

MAC e PHY hanno diverse modalità standard di collegamento, che genericamente prendono il nome di Media-Independent Interface:

MII: Acronimo di Media-independent Interface, richiede 18 segnali ed è può raggiungere i 100 Mbit, con un clock di 25 MHz;

RMII: Acronimo di Reduced Media Independent Interface, richiede 9 segnali, ma aggiunge qualche difficoltà nella identificazione degli errori di trasmissione. Per raggiungere le stesse prestazioni del MII necessita di un clock doppio (50 MHz per 100 Mbit);

GMII: Acronimo di Gigabit Media Independent Interface, Retrocompatibile con MII, utilizzando clock più veloci (125 MHz), raggiunge velocità di un 1 Gbit;

RGMII: Acronimo di Gigabit Media Independent Interface, Versione con numero di segnali ridotti rispetto al GMII.

5.2.8 GPIO

Un GPIO[42] (General Purpose Input/Output) è la rappresentazione di un *pin* (o di una cosiddetta *ball*[16] su package) e del segnale digitale che transita su di esso.

Semplificando molto l'idea, si immagini un GPIO come un interruttore[17] digitale, che può essere acceso o spento. Analogamente all'interruttore, dove in base alla stato vi è o meno passaggio di corrente, è possibile definire i due stati di un GPIO, "alto" o "basso" in base alla presenza di una determinata tensione sul pin. Per chiarire meglio il concetto, si pensi ad una determinata tensione di riferimento V_{ref} (tipicamente, ma non necessariamente in valori compresi tra $1V8$ e $5V$, spesso $3V3$), e si immagini che sul *pin* associato al *GPIO* vi possa essere una tensione che varia da $0V$ a V_{ref}. Quando la tensione sul *pin* è prossima ad un intorno di V_{ref}, il GPIO è nello stato "alto", per contro quando è in un intorno di 0V il GPIO è nello stato basso. Intuitivamente, parlando di segnali digitali, si può associare allo stato "alto" il valore di **1 logico** e allo stato "basso" lo **0 logico**. Tale associazione tra valori logici e stati è vera qualora i GPIO siano configurati nella cosiddetta modalità "attivo alto" (o *active-high*). Esistono però configurazioni di GPIO che utilizzano la "logica negativa", definita anche "attivo-basso" (*active-low*): in tale modalità i valori di 0 e 1 corrispondono rispettivamente allo stato "alto" e "basso".

Frequentemente, durante la progettazione di un SoC, i pin non utilizzati da altri device vengono allocati come GPIO; esaminando gli schematici dei SoC o le specifiche delle evaluation board è infatti possibile sapere a che cosa siano collegati i singoli pin. In molti casi è inoltre possibile, tramite software (vedere ad esempio i *device-tree* in 6.2.2), imporre all'hardware una configurazione per i singoli *pin*, potendo scegliere se associarli ad alcune Intellectual Property, o destinarli come GPIO. Per la loro semplicità e versatilità, i GPIO sono quasi onnipresenti all'interno delle *evaluation board* dei vari SoC e MCU. In taluni casi,

[16] Il nome "ball" si riferisce alla struttura dei pin sul package dei chip. Parlando della loro disposizione esterna, ci si riferisce al "Ball Grid Array".

[17] Per un maggiore approfondimento in materia si consiglia la lettura del capitolo 3.

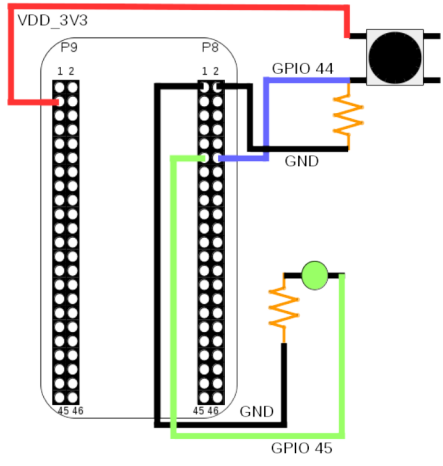


Figura 5.11: Esempio di collegamento di un pulsante ed un led su BBB

laddove per limiti fisici non sia possibile disporre di un numero sufficiente di *pin* da associare come GPIO, si possono utilizzare dei *"GPIO Expander"*[18].

Le modalità di utilizzo dei GPIO sono le seguenti:

Input: Il GPIO è utilizzato per leggere valori binari dal pin

Output: Il GPIO è utilizzato per scrivere sul pin valori binari

IRQ: Si tratta di una particolare modalità di *input* in cui la variazione di valore sul pin viene utilizzata per scatenare interrupt all'interno del sistema, come ad esempio per gestire un evento di "risveglio" (*WakeUp*) di un dispositivo da una fase di *standby*.

Esistono GPIO che possono essere configurati sia come *input* che come *output*, ma in alcuni dispositivi vengono configurati dal costruttore per utilizzare una sola di queste modalità.

All'interno di un sistema embedded, i GPIO possono essere utilizzati per il monitoraggio di presenza per una scheda SD/MMC, il controllo di led e pulsanti (come nell'esempio in

[18]Dispositivi hardware collegati tramite bus I2C o SPI, mettono a disposizione batterie di GPIO.

figura 5.11), bit-banging su linee seriali, o l'invio di segnali a degli *hardware watchdog*[19].

5.2.9 Bit banging

Il "bit banging" è una tecnica per implementare comunicazioni seriali generando treni di impulsi via software, evitando qualunque tipo di hardware specifico. Si basa sul fatto che i processori odierni, anche per l'embedded, sono abbastanza veloci da poter "emulare" un circuito dedicato alla generazione di segnali non eccessivamente complessi.

Intuitivamente: per inviare su una linea seriale un treno di impulsi corrispondente ad un byte è sufficiente "ciclare" su tutti i bit[20] del byte stesso alzando (HIGH) il piedino di output utilizzato ogniqualvolta si incontra un "1". Ovviamente bisogna essere abbastanza sicuri che il ciclo venga eseguito con una velocità definita, nota e stabile, altrimenti si andrà incontro a errori di trasmissione, oppure sarà necessario disporre di una linea separata per il "clock" (per avvisare che un bit è pronto da leggere).

Il vantaggio del *bit banging* è nell'economicità dell'implementazione (non c'è hardware dedicato) e nella flessibilità sul protocollo di comunicazione: è molto facile infatti aggiornare il device per supportare nuovi protocolli semplicemente aggiornando il software. Lo svantaggio risiede nella minore robustezza: l'implementazione software viene infatti influenzata da ciò che sta girando sul device, ad esempio la gestione degli interrupt potrebbe "distrarre" il processore abbastanza a lungo da far perdere dati. Inoltre la soluzione software costa tempo di CPU.

[19] Un *watchdog* (letteralmente "cane da guardia") è un meccanismo che permette il "monitoraggio" grossolano di un device. E' tipicamente un circuito elettronico, esterno e indipendente al device, che deve essere "stimolato" periodicamente, ad esempio mediante un segnale su un GPIO. In assenza di stimolo il watchdog resetta il device. La semantica del segnale sul GPIO verso il watchdog è "sono vivo, tutto bene". La *ratio* (ma si potrebbe dire: la speranza) del reset da parte del watchdog è che un reboot risolva eventuali problemi. Gli appassionati di una famosa serie televisiva inglese ricorderanno il "Have you tried turning it off and on again?".

[20] Utilizzando una funzione di *shift* dei bit di un byte.

5.2.10 JTAG

Pur non rientrando a pieno titolo tra i dispositivi di I/O il JTAG è uno strumento fondamentale per lo sviluppo di dispositivi embedded, adattato per lo più universalmente in tantissime *development board* e *validation board* per il testing e validazione sia hardware che software.

Fino agli anni 80, il test degli elementi interni dei chip era eseguito tramite pin e test-point attraverso la tecnica definita *"bed of nails"*[21] Con lo sviluppo di circuiti integrati sempre più complessi, articolati ed avanzati, questa soluzione non risultava più praticabile costringendo i progettisti a predisporre delle soluzioni alternative. Nel 1985 il consorzio di aziende nord americane ed europee chiamato ***Joint Test Action Group*** (da qui l'acronimo JTAG) propose la metodologia definita come ***Boundary Scan***[22] Ridefinita e puntualizzata dal *Institute of Electrical and Electronics Engineers* è stata formalizzata nello standard IEEE 1149.1[28] nel 1990 intitolato ***Standard Test Access Port and Boundary-Scan Architecture***. Ad oggi i termini *Boundary Scan* e JTAG sono utilizzati per lo più come sinonimi.

Per semplificare molto il concetto, si può pensare al JTAG come una linea di celle interconnesse messa tutta intorno alla logica input/output del chip. Queste celle, in condizioni normali rimangono non operative, e lasciano transitare i segnali di input/output. Quando, attraverso il cosiddetto *TAP*[23] *Controller* si abilita il bypass, le celle possono raccogliere le informazione sullo stato complessivo dei registri e della memoria, permettendo un controllo totale su quanto eseguito all'interno dell'elaboratore.

Tutto questo di solito è sfruttato attraverso specifici connettori JTAG[24] e le cosiddette *probe*[25] di collegamento tra la board

[21] Tradotto letteralmente sta per letto di chiodi, indica l'utilizzo di una fitta serie di pin collegati all'interno del chip alla logica di quest'ultimo.

[22] Testualmente "Scansione dei confini", indica sia il tipo di test che la modalità di collegamento (agli estremi esterni del circuito stampato).

[23] **T**est **A**ccess **P**ort.

[24] Esistono tipologie di connettori JTAG diversi tra loro.

[25] Una probe (sonda) è un dispositivo indipendente vero e proprio, in grado di tradurre quanto rilevato sul JTAG in un qualcosa comprensibile

di sviluppo ed il computer utilizzato per effettuare il debug. All'interno del computer deve essere in esecuzione lo specifico software relativo alla *probe* utilizzata.

Data la estrema varietà legata a questo argomento, in parte dovuta alla architettura su cui si sta operando, ed in parte relativa al debugger[26] in utilizzo, si raccomanda il lettore di leggere attentamente la documentazione relativa al tipo di apparato che si vuole sfruttare. C'è inoltre da far notare che, se pur presente in moltissime *evaluation board*, in alcuni esemplari di piccole dimensioni, potrebbe essere assente il connettore JTAG; ciò può avvenire anche qualora il SoC o MCU in utilizzo disponga al suo interno della componentistica per il *Boundary Scan*. I motivi di tale assenza potrebbero essere legati ad una scelta specifica di sicurezza (esempio si vuole evitare il "*reverse engineering*", la ingegnerizzazione inversa), o per risparmio economico o semplicemente per ridurre lo spazio occupato sulla board.

Nota bene: in generale, sebbene estremamente potente, l'utilizzo del JTAG è raccomandabile solo per testare l'hardware o il software che lavora a basso livello (come ad esempio i driver).

per il software di debug utilizzato all'interno del computer. Ne esistono differenti modelli, sia per potenza che per modalità di collegamento, ed in molti casi si tratta di costose implementazioni proprietarie.

[26] In questo contesto inteso come insieme di *probe* ed software di debug in esecuzione sul computer.

Capitolo 6

Il sistema operativo

Lo scopo di questo capitolo è quello di dare una panoramica generale sui sistemi operativi utilizzati in molti sistemi embedded, approfondendo in particolare i sistemi GNU/Linux.

6.1 Introduzione

Il sistema operativo è un elemento software atto a gestire le risorse hardware del sistema e offrire servizi agli altri software. Frequentemente il termine "sistema operativo" è abbreviato con il suo acronimo inglese OS (*Operating System*). Il termine può indicare un'enorme varietà di software differenti: da sistemi semplici composti da qualche centinaia di righe di codice, a sistemi assai complessi che arrivano tranquillamente a decine di milioni di linee. A causa di un uso talvolta improprio del termine, in particolare legato a produttori di alcuni sistemi operativi, nel gergo comune si tende a fare una certa confusione tra quello che è realmente il sistema operativo ed il software applicativo. In letteratura si tende comunque ad attribuire due significati alla dicitura OS: la prima prevede solo la parte del "nucleo" (o kernel), la seconda comprende alcuni software aggiuntivi quali binari, librerie, file di configurazione e script essenziali per l'avvio del sistema.

6.1.1 Classificazione degli OS

È necessario premettere che, data la grande varietà di OS esistenti e il fatto che siano progettati, sviluppati e rilasciati nei modi più svariati, non è sempre possibile applicare dei parametri di classificazione tassonomica con assoluta precisione. Tuttavia è possibile stabilire dei criteri che consentono di identificare la tipologia di OS analizzato.

Licenza

Il sistema operativo, così come la stragrande maggioranza del software comunemente distribuito, è soggetto a rilascio secondo licenza. Una licenza è una accordo (esplicito o implicito) tra coloro che scrivono e pubblicano il software ed i destinatari designati (maggiori informazioni in merito sono disponibili nella sezione 2.7). Sulla base di come il sistema operativo venga rilasciato, ha senso suddividerli in macro-categorie:

Licenza libera: si tratta di quei sistemi operativi di cui sono disponibili i codici sorgente e l'uso per ogni scopo. La tipologia di licenza consente di sviluppare e rilasciare le proprie copie del prodotto sviluppato. Un esempio tipico di questo tipo di OS sono i sistemi GNU/Linux, rilasciati sotto licenza GPL.

Licenza proprietaria: si tratta di quei sistemi per cui ogni istanza del software deve essere stata rilasciata secondo i dettami di una EULA (End User License Agreement), solitamente legate a un corrispettivo economico. Non è possibile distribuire, sviluppare o redistribuire tali sistemi operativi se non attraverso particolari accordi con il produttore. Un esempio tipico sono i sistemi Microsoft Windows.

Licenza ibrida: sono quei sistemi che sono proprietari, ma che hanno alcune componenti che sono rilasciate con licenza libera o quei sistemi liberi con componenti di rilievo rilasciati con licenza proprietaria.

Risulta evidente come i sistemi a licenza libera offrano più possibilità di accesso e di sviluppo su progetti di varie dimensioni, come nei comparti dell'industria *Mobile*, *Infotaiment* ed

IoT. Nei settori con una elevata richiesta di certificazioni (quali *Industrial*,*Automotive*,*Biomedical*,*Aerospace*) si può constatare invece la frequente adozione di sistemi proprietari ed ibridi. Non va ignorato inoltre che in progetti di notevoli dimensioni, laddove vi siano più sistemi embedded in cooperazione, ci possono essere sistemi liberi che lavorano al fianco di sistemi proprietari.

Struttura del kernel

Nel corso dello sviluppo degli OS, vi è stata una continua evoluzione per quanto concerne la struttura (o architettura, anche se tale termine può generare confusione) del kernel.

Monolitici: un kernel monolitico è caratterizzato della presenza all'interno del nucleo, oltre che dei driver e delle strutture di multiplexing dell'hardware, anche di una serie di servizi per la gestione dei processi e della comunicazione. Tale approccio è stato utilizzato a partire dai primi kernel BSD, ed è tuttora utilizzato nei kernel Linux e FreeBSD. Benché questo modello abbia degli svantaggi dal punto di vista della manutenzione, estensione e dimensione del kernel, è oggi molto adottato per la facilità di sviluppo che si traduce in costi di impiego complessivamente più bassi;

Microkernel: un microkernel è caratterizzato dal contenere la quantità minima possibile di codice che viene eseguito con accesso "supervisore". Per raggiungere tale obiettivo, si tende a portare fuori dal nucleo principale molti componenti "esterni" (es. *driver* periferiche). Ciò è possibile grazie alla aggiunta di uno strato tra il kernel minimale ed il resto, composto da elementi software definiti "server". Tali server sono i tramite della comunicazione tra il nucleo principale e tutto il resto del codice. La comunicazione tra questi elementi avviene con un meccanismo definito IPC (Inter Process Comunication). A fronte di una grande sicurezza e robustezza, si contrappone una notevole complessità di sviluppo ed una certa rigidità comportamentale complessiva. Un esempio classico di tale approccio è Minix[1] o, parlando di sistemi embedded molto utilizzati in

[1] `https://www.minix3.org`

ambito MCU, FreeRTOS e QNX;

Ibridi: sono quei kernel che pur avvalendosi di un'architettura simile a quella microkernel, per motivi di semplicità o di miglioramento delle prestazioni, integrano all'interno del kernel minimale parti considerate "non essenziali" nella classica struttura "microkernel". Questo modello ha trovato grande riscontro in sistemi desktop/workstation commerciali quali XNU (Kernel di MacOS) e WindowsNT di Microsoft;

ExoKernel: si tratta per lo più di un nuovo approccio accademico, volto a ridurre il kernel al solo multiplexing delle risorse hardware. In pratica tutte le operazioni del sistema operativo vengono poste all'infuori del kernel in una o più *libOS* (librerie con compiti di sistema operativo). Tale approccio, almeno in teoria, permetterebbe la coesistenza di più libOS (e di conseguenza di più sistemi operativi) al di sopra del medesimo Exokernel.

Per quanto alcuni microkernel abbiano trovato ampio mercato all'interno dei sistemi embedded, sono in genere utilizzati su CPU di piccole/medie dimensioni (es. le MCU) e per applicativi real-time. All'aumentare delle dimensioni e della complessità della CPU (come avviene nei SoC) è di gran lunga preferibile utilizzare un kernel monolitico (migliori tempi di sviluppo) e libero (dovendo rispondere a forti necessità di personalizzazione e di contenimento dei costi). Per questi motivi, per buona parte dei SoC di fascia alta presenti sul mercato, siano essi ARM o MIPS, Linux risulta essere il sistema operativo di riferimento. Nel settore mobile, per SoC dotati di GPU e/o dispositivi di Encoding/Decoding video si predilige Android[2] rispetto a Linux. Tale scelta, più che per ragioni tecniche, è legata a fattori di collocamento sul mercato e di maggiore standardizzazione relativa alla gestione dei flussi video e delle applicazioni 3D.

[2]Che è comunque una derivazione di GNU/Linux.

6.2 I sistemi GNU/Linux

6.2.1 Storia dei sistemi GNU/Linux

Nel 1983, Richard Stallman, attivo sviluppatore presso i laboratori del MIT, lanciò il progetto GNU (acrostico ricorsivo, sta per "GNU is Not UNIX"): l'intento di questo progetto era di realizzare un sistema operativo simile a Unix, ma rilasciato con licenza libera. A seguito del progetto GNU, fu fondata la Free Software Foundation, una fondazione nata per portare avanti le finalità del progetto GNU. Nonostante i risultati incoraggianti dal punto di vista di molti componenti software (uno su tutti il compilatore C, ad oggi uno dei più utilizzati in assoluto), il cosiddetto "sistema operativo GNU" risultava essere ancora privo di un kernel completo. Il kernel in sviluppo era quello che prenderà il nome di Hurd, un microkernel basato sul core Mach, ritenuto interessante dal punto di vista architetturale, ma mai giunto a maturità.

Nel 1992 Linus Torvalds, uno studente dell'università di Helsinki rilasciò con licenza libera il suo kernel (inizialmente nominato FreeX, in seguito ribattezzato Linux), ideato per essere eseguito su sistemi Minix[3]. In breve periodo il kernel fu adattato per poter coesistere con l'ambiente *userspace* del progetto GNU, dando vita ai sistemi GNU/Linux. Ad oggi, per semplicità, si tende ad indicare con il solo termine Linux tutto il sistema operativo completo... mandando in escandescenze Richard Stallman.

La licenza libera del kernel e dell'intero progetto GNU, ha "concesso"[4] a milioni di sviluppatori la libertà di realizzare proprie versioni personalizzate dei singoli software o dell'intero sistema operativo, dando vita al fenomeno delle distribuzioni GNU/Linux[5].

Le "distribuzioni" (gergalmente definite anche "distro") sono dei sistemi operativi completi, realizzati partendo dai medesimi

[3] Il sistema operativo basato su Microkernel scritto dal professor Tanenbaum dell'università di Vrije.

[4] Difficile tradurre bene la semantica di "empower" in italiano.

[5] Si veda il sito `http://distrowatch.com` per un elenco commentato delle distribuzioni.

codici sorgente, ma con differenze dovute a personalizzazioni e ottimizzazioni.

È possibile definire tre categorie distinte di distribuzioni:

Hobbistiche: team di sviluppo limitato a poche persone (o una sola), create e mantenute per soddisfare esigenze particolari, personalizzazioni dettagliate o per scopi didattici.

Community: realizzate attraverso il lavoro di sviluppo collettivo di "comunità" di volontari di medie/grandi dimensioni.

Enterprise: sviluppate principalmente all'interno di aziende da professionisti dedicati per sopperire alle richieste di sviluppo, mantenimento ed assistenza d'imprese, istituti finanziari ed enti pubblici.

In alcuni casi le "realtà community" sono sovvenzionate dal punto di vista economico dalle medesime aziende che producono prodotti enterprise; tale comportamento, apparentemente incompatibile con i modelli di business tradizionali, crea un flusso bidirezionale di idee e sviluppo software fra queste due realtà, migliorandosi vicendevolmente.

In un primo momento si poteva avere la percezione che questa tipologia di sistemi avesse un ristretto pubblico d'elezione ma, già a partire della fine degli anni '90, i sistemi GNU/Linux hanno ricevuto l'attenzione delle grandi compagnie hardware e software, arrivando ad avere una predominanza nel settore "enterprise" (server/workstation) ed in quello "embedded" (mobile, automotive, industrial, home automation), ma non in quello desktop, dove Linux è rimasto sempre, purtroppo, ai margini del mercato.

6.2.2 Il kernel Linux

Linux è un kernel monolitico rilasciato con licenza libera (GPLv2). Il kernel, creato inizialmente per un'architettura x86 (per un Intel i386 a 32 bit) ad oggi supporta numerose architetture hardware differenti (x86, x86_64, IA64, arm, aarch64, alpha, mips, s390, PowerPC) nelle loro varianti a 32 o a 64 bit.

Linux è scritto in prevalenza con linguaggio C, anche se a tutt'oggi sono presenti ancora molte parti scritte in linguaggio

assembly[6] (Sintassi GAS, AT&T). Per molti anni l'unico compilatore utilizzabile per il kernel Linux è stato GCC (anche grazie a delle opzioni speciali aggiunte appositamente nel compilatore per il kernel), ma negli ultimi anni sia Intel che Clang (LLVM) stanno sviluppando in questa direzione.

Alcune caratteristiche del kernel linux:

Preemptive Multitasking: attraverso un meccanismo basato su interrupt, viene sospeso il processo corrente, invocato lo scheduler e sulla base di un diverso livello di priorità, viene eseguito un determinato processo. In questo modo si cerca di garantire a tutti i processi un accesso pesato alle risorse hardware.

Memoria Virtuale: i processi utilizzano un spazio di indirizzamento virtuale che viene "mappato" su quello fisico, attraverso un meccanismo di traduzione indirizzi hardware (memory management unit o MMU). Semplifica le gestione della memoria da parte delle applicazioni, migliora sicurezza e isolamento. Pur essendo possibile fare il mapping con aree di memoria non presenti sulla RAM fisica proprio grazie al meccanismo di VM, il termine "memoria virtuale" è stato per anni erroneamente scambiato per la tecnica di swapping della memoria su disco fisso.

Loadable Kernel Module: trattandosi di un kernel monolitico è lecito aspettarsi che tutta la parte dei driver si trovi nello stesso spazio di memoria (il cosiddetto *kernel space*, o Ring 0). Se in passato era necessario eseguire il caricamento di tutto il kernel durante la fase di boot, con l'avvento dei "moduli kernel caricabili dinamicamente", ovvero dei file oggetto compilati con il medesimo compilatore e *header* del kernel, è oggi possibile aggiungere in un momento secondario all'avvio il supporto per alcune periferiche. Tale strumento ha permesso di ridurre le dimensioni dei kernel caricati in fase di boot, e di selezionare semplicemente i driver necessari una volta avviato il sistema.

[6] Linguaggio a basso livello molto vicino al linguaggio macchina (talvolta con un rapporto 1 ad 1 tra istruzione assembly e istruzione macchina).

Symmetric Multiprocessing (SMP): la sigla SMP definisce un sistema multiprocessore omogeneo (ovvero dotato di più processori identici) in grado di condividere la stessa system-ram e i medesimi I/O, coordinati dalla stessa istanza del sistema operativo. Discorso analogo vale per i recenti sistemi multicore, che vengono trattati come singoli processori distinti.

Copy on Write: è assai frequente che più processi lavorino, in lettura e scrittura, su copie di dati. Con il metodo COW (Copy on Write) si alloca un dato una prima volta in memoria, poi, quando altri processi devono averne delle copie (ad es. per un passaggio di parametri) si crea uno speciale puntatore senza duplicare effettivamente i byte. Se poi un processo tenterà una modifica del dato allora, e solo in quel momento, il sistema operativo effettuerà la vera e propria copia.

Kernel API

Il kernel, per potersi interfacciare con il resto del sistema, mette a disposizione una serie di API (Application Program Interface), suddivise in due tipi distinti: le ***In-kernel*** api e le ***kernel-to-userspace*** API.

Le In-kernel sono quelle utilizzate dai cosiddetti sottosistemi: attraverso queste API è possibile standardizzare il comportamento e la scrittura dei driver per determinate periferiche. Importante far notare che tale standardizzazione non vale al proseguire delle versioni di kernel: un driver scritto per la versione 3.X del kernel potrebbe essere non compatibile con la versione 3.X+1 e nemmeno retro-compatibile con la versione 3.X-1.

Alcuni esempi delle In-kernel API:

Bluez: comunicazioni bluetooth
Mac80211: per le interfacce wireless
Direct Rendering Manager (DRM): acceleratori grafici
Kernel Mode Setting (KMS): display controller
Video4Linux (V4L): sottosistema di cattura video per Linux

Advanced Linux Sound Architecture (ALSA): sottosistema per le schede audio

Le API kernel-to-userspace invece rappresentano l'interfaccia del kernel per le cosiddette "*syscall*" (o chiamate di sistema). In particolare definiscono come alcune librerie interagiscano con il kernel, cercando di offrire per quanto possibile gli standard POSIX (Portable Operating System Interface) e Single Unix System Specification (IEEE 1003[30] , ISO/IEC 9945). L'elenco delle syscall è assai vasto e non rientra nell'interesse specifico per la trattazione qui svolta.

Il device-tree

Il "device tree" è una struttura dati atta a descrivere particolari configurazioni hardware, utilizzata in alcuni dispositivi embedded di recente sviluppo, dotati di kernel Linux versione 3.5 o successiva. In sostanza un device-tree sotto forma di sorgente può essere visto come una descrizione dettagliata di come l'hardware presente sia configurato, indicando mappature tra device e indirizzi di memoria, gestione dei pin di uscita, configurando dei "clock" o cambiando valore a particolari registri.

Come già accennato nella descrizione dell'hardware, i dispositivi embedded odierni sono in grado di trasformarsi e utilizzare non simultaneamente dispositivi diversi sui medesimi *pin* a velocità differenti. Prima dell'avvento del device-tree, tali capacità erano però limitate al fatto che ogni qualvolta si doveva riconfigurare qualcuno dei parametri sopracitati, era necessaria una riscrittura di un driver e una successiva ricompilazione.

Affidando queste parametriche al device-tree, tali operazioni di riconfigurazione possono essere fatte al netto di un riavvio di sistema. In taluni casi è possibile vedere come, in base al riconoscimento di alcuni specifici segnali, un sistema possa caricare un device-tree differente, cambiando di fatto il comportamento complessivo del sistema. Il risultato finale è una maggiore libertà di sviluppo, senza la necessità di dover scendere nei dettagli dell'implementazione del codice.

Il device-tree è presente nel sorgente del kernel come file di testo, ma viene compilato sotto forma di file binario per essere

caricato dal bootloader nelle prime fasi del boot. Nei primi sistemi con device-tree, i boot loader non disponevano ancora del supporto per il device-tree: in questi casi la versione binaria del device-tree veniva aggiunta in fondo al file immagine compresso del kernel[7].

6.3 Root Filesystem

Come è facilmente intuibile, il sistema non è composto esclusivamente dal kernel che, per quanto indispensabile, da solo non è sufficiente. Altri elementi vanno a comporre quello che è l'insieme del software essenziale alla definizione del sistema: la *libc* (la libreria C) e l'interprete di comandi (una *shell* nella maggior parte dei casi) e alcuni *tool* essenziali. Questi elementi sono collocati all'interno del cosiddetto "*root filesystem*" (o rootfs). Contrariamente a quanto il nome possa suggerire, con questo termine non si specifica nessuna tipologia di filesystem, ma il contenuto. Con "rootfs" infatti si indica la partizione (o sistema di storage locale/remoto) all'interno della quale è possibile identificare gli elementi necessari all'avvio del sistema. Tale partizione viene solitamente montata come primo nodo dell'alberatura delle cartelle, che prende il nome di *root* (dall'inglese radice, indicata nel sistema con la "/"). Occorre precisare che si tratta di un collocamento assolutamente convenzionale e che, con le opportune modifiche, si potrebbe avere un sistema le cui componenti siano altrove rispetto alla posizione della partizione "/".

6.3.1 La libreria C

La "libreria C" è un elemento fondamentale all'interno del sistema operativo, in quanto fornisce tutti gli header e le funzioni base utilizzate dal linguaggio di programmazione C (alla base dei sistemi GNU/Linux). Per fare alcuni esempi concreti, basti considerare alcuni degli header forniti:
***stdio.h*:** funzioni basilari di input/ouput (es. *printf*, *scanf*);

[7]Per quanto tale metodo sia tutt'ora utilizzabile è considerato obsoleto e poco raccomandabile.

***stdlib.h*:** funzioni di allocazione memoria e sistema (es. *malloc*);
***signal.h*:** funzioni controllo segnali (es. *kill*, *raise*);
***string.h*:** funzioni di manipolazione stringhe (es. *strcpy*, *memcpy*);
***math.h*:** funzioni matematiche (es. *sqrt*).

Nella maggioranza dei casi, un sistema GNU/Linux privato della libreria C, non è in grado di funzionare correttamente. Esistono più tipi di *libc*, alcune più generiche, altre dedicate a dei settori specifici.

GNU C Library (glibc): in assoluto la libreria C di riferimento nei sistemi GNU/Linux, la più completa e di conseguenza anche la più ingombrante;
eglibc: variante della nota glibc pensata specificatamente per ambienti embedded. Lo sviluppo è fermo al 2014;
musl: disegnata per essere particolarmente pulita ed efficiente, è stata progettata per preservare la possibilità di una compilazione statica (ambienti real-time). Compatibile con lo standard Posix 2008 e C11, è utilizzata in sistemi Linux pensati per i router (ad es. OpenWrt[8] e il suo *fork* più recente LEDE[9]);
uClibc: libreria C pensata appositamente per essere estremamente compatta (la u sta per μ, ad indicare "micro"). Inizialmente era stata progettata per essere la libreria C di $\mu-$Linux, la versione ridotta pensata per sistemi non dotati di MMU;
Bionic: è la versione libc ideata da Google per i sistemi operativi basati su Android; dalle dimensioni ridotte e dal codice ottimizzato, al momento non risulta essere compatibile con gli attuali standard POSIX.

6.3.2 Init

Nei sistemi operativi di derivazione diretta ed indiretta di Unix, "init" rappresenta il primo processo del sistema operativo (il suo

[8] `https://openwrt.org`.
[9] `https://lede-project.org/`.

PID[10] è convenzionalmente uguale a 1). Nel modello gerarchico "padre/figlio" dei processi, *init* risulta essere il processo "antenato" di tutti i processi in esecuzione. Tale processo rimane attivo dall'avvio del sistema, sino alle fasi di riavvio o spegnimento. Il tentativo di chiudere questo processo, porta ad un errore denominato "kernel panic"[11].

Un sistema di init è invece l'insieme di strumenti software e meccanismi di configurazione che comprende anche il processo init: tale sistema consente l'avvio di tutti gli applicativi e demoni previsti dal sistema. In alcuni casi limite è possibile non utilizzare un vero e proprio processo di init, usando in alternativa una shell minimale.

Tra le caratteristiche comuni ai sistemi di init, esiste il concetto di *runlevel*, ovvero l'identificazione di diverse modalità operative con cui chiamare il processo: tali runlevel, solitamente indicati con un numero progressivo da 0 a 6, possono assumere un significato differente sulla base della configurazione, anche se come standard de-facto (non sempre rispettato alla lettera) si identificano nel seguente modo.

0 (Halt): è il runlevel impostato per eseguire lo spegnimento del sistema;

1 (Single): il runlevel impostato per avviare il sistema in "Single User Mode", ovvero caricando il numero minimo di servizi indispensabili per offrire un'interfaccia testuale non multiuser;

2: non specificato. In alcuni sistemi è la modalità multiuser senza il supporto di rete;

3 (Multiuser): in genere il sistema avviato nella sua interezza, multiutente, servizi di rete e demoni, ma senza interfaccia grafica;

4: non specificato;

5 (GUI): equivalente alla modalità multiuser, con l'aggiunta dell'interfaccia grafica;

6 (Reboot): il runlevel impostato per eseguire il riavvio del sistema.

[10] **P**rocess **I**dentification **N**umber.

[11] Un "kernel panic" è un errore irreversibile nei sistemi operativi Unix-like che comporta lo stop forzato di ogni processo di sistema e del kernel.

In alcuni sistemi di init, pur non esistendo un vero e proprio supporto al meccanismo dei runlevel, questo viene implementato attraverso degli script atti ad emularne il comportamento (come avviene ad esempio nel Busybox Init).

Nel corso degli anni vi sono state numerose variazioni ai sistemi di init utilizzati dalle distribuzioni GNU/Linux; per quanto possa sembrare una scelta ininfluente sull'utente finale, la scelta del sistema di init ha rappresentato un fattore di caratterizzazione delle distribuzioni stesse. Al fine di comprenderne le differenze verranno presentati alcuni esempi di init.

BSD Init: di derivazione dai sistemi BSD Unix è basato su una serie di script presenti nella cartella "/etc" (rc). Il processo init esegue sequenzialmente le operazioni descritte all'interno dei vari script. Inizialmente non disponeva del supporto per i runlevel, aggiunto nelle versioni più recenti;

Sys-V init: di derivazione dai sistemi Unix System-V da cui prende il nome, ha introdotto il sistema dei runlevel e, nei sistemi moderni, dispone di una cartella specifica per ogni runlevel all'interno della cartella "/etc". Anche in questo caso si tratta di esecuzione sequenziale;

systemd: systemd è un complesso sistema di librerie, tool e file di configurazione che, tra le altre cose comprende anche il sistema di init. Di recente concezione, e scelta predefinita per le maggiori distribuzioni Linux desktop/server, offre meccanismi di caricamento dei servizi in parallelo e la gestione di dipendenza ed ottimizzazioni specifiche per le architetture multiutente. Il suo utilizzo nel settore embedded è ancora in discussione, anche se la sua maggiore complessità nella gestione e l'inefficacia di alcune ottimizzazioni lo renderebbero poco adatto ai sistemi di piccole dimensioni;

Busybox Init: integrato all'interno del binario di busybox, offre un meccanismo semplice, basato su script, per l'avvio dei servizi e del software. Ideale per piccoli sistemi senza grandi modifiche a tempo di esecuzione, mostra forti limiti di gestione su sistemi di dimensioni maggiori.

6.3.3 La shell

Parlando d'interazione con l'utente si possono distinguere due differenti tipologie di interfacce: le cosiddette CLI (Command Line Interface) strettamente testuali o quelle di tipo grafico (Graphical User Interface o GUI). Nella pratica comune con il termine *shell* si tende a indicare la prima tipologia, ovvero quella testuale. È necessario sottolineare l'opportuna distinzione tra shell e terminale, usati erroneamente come sinonimi. Nel primo caso si tratta di uno software atto all'interpretazione dei comandi, parlando di terminale si fa invece riferimento ad un dispositivo hardware (della tipologia dei cosiddetti dispositivi a caratteri, come ad esempio la tastiera o le porte seriali) di comunicazione, sia esso reale o emulato. Fatta questa opportuna distinzione, rimane comunque valida l'accezione "comandi da terminale" per indicare l'uso di comandi all'interno di una CLI.

A partire dai primi sistemi UNIX, gli OS sono stati sempre dotati di una (o più) shell per interfacciarsi con l'utente e permettere di eseguire task in modo interattivo (lanciando i comandi manualmente) e non interattivo (ponendo i comandi in dei file di testo per un'esecuzione automatica). In molti casi le shell definiscono un vero e proprio linguaggio (definito "di *scripting*") che possa essere interpretato durante l'esecuzione. I file contenenti i comandi da interpretare sono definiti *script*.

In ambienti Unix/Linux possono essere presenti una o più delle seguenti shell:

Bourne shell: scritta da Stephen Bourne e rilasciata nel 1977, apparve per la prima volta all'interno di Unix versione 7, era indicata generalmente con il nome di "sh" (il nome del file binario per richiamarla).

C-shell: rilasciata alla fine degli anni '70 per i sistemi BSD (Berkeley Software Distribution), C-shell deve il suo nome allo sforzo compiuto dagli sviluppatori per far si che il suo linguaggio di scripting fosse il più possibile aderente alla sintassi del più noto linguaggio di programmazione C, considerato più leggibile[40].

Bourne again shell: nata nel 1989 come sostituzione della Bourne shell (da cui prende il nome), ha trovato grande applicazione in buona parte dei sistemi Unix/Linux esi-

stenti. Ad oggi è considerata la shell predefinita dei sistemi GNU/Linux, ma non in ambito embedded, dove risulta sovradimensionata. Viene indicata con il nome *bash*.

ash: nata alla fine degli anni '80 per sostituire la Bourne shell e sviluppata da Kenneth Almquist (da cui prende il prefisso 'a'). Nonostante non sia completa come bash, è stata particolarmente apprezzata nei vari ambienti embedded per la sua compattezza, tanto da essere la scelta predefinita nei sistemi basati su *busybox*.

Korn shell: basata su Bourne shell, molto completa ed estensibile, ha trovato applicazioni in ambienti server, in particolare per il suo linguaggio di scripting, che ben si sposa con ambienti DB.

6.4 Bootloader

Un *bootloader* è un software creato per consentire il caricamento del kernel, la selezione dei parametri di avvio e l'avvio vero e proprio del sistema. Per adempiere a tale scopo è necessario che questi siano in esecuzione prima del sistema operativo stesso. In sistemi di elaborazione di complessità superiore alle MCU, o comunque che eseguano sistemi operativi completi, la presenza di un bootloader è da considerarsi necessaria. Nei personal computer, così come nei sistemi workstation e server, è assai comune che ogni sistema operativo abbia un proprio bootloader. All'occorrenza e se opportunamente configurato, un bootloader può prestarsi anche all'avvio di sistemi operativi differenti da quello "predefinito". Nel mondo GNU/Linux sono famosi Grub[12] e LILO[13], mentre i sistemi Apple e Microsoft utilizzano un loro software proprietario.

Un bootloader è a sua volta un piccolo "sistema operativo", in quanto deve disporre degli elementi software necessari (potremmo definirli impropriamente dei driver) per poter accedere ai dispositivi di storage che contengono il sistema operativo (o

[12] **GR**and **U**nified **B**ootloader.
[13] **LI**nux **LO**ader.

parti di esso) ed in alcuni casi anche dispositivi di rete, porte seriali ed altre porte di comunicazione.

Nella maggioranza dei casi, nei PC (o in elaboratori accostabili a tale categoria), si dispone di un BIOS[14] (ovvero di un firmware residente in una memoria FLASH/EEPROM della scheda madre) atto ad inizializzare l'hardware nel modo corretto. Tra i compiti di un BIOS c'è la configurazione dei clock di sistema (velocità CPU, RAM, BUS), il rilevamento dei dispositivi di storage, l'inizializzazione di dispositivi di comunicazione, schede grafiche e gestione energetica. In sostanza un bootloader per un computer tradizionale trova parte del lavoro già svolto, limitandosi ad attivare il corretto interfacciamento con l'utente per consentire la scelta del sistema e il passaggio dei parametri. Nei sistemi di ultima generazione i classici BIOS sono stati sostituiti dai più moderni UEFI[15], ancora più completi e complessi, tali da richiedere dello spazio sul sistema di storage principale (invece che risiedere nella sola EPROM).

Nei sistemi embedded la situazione è decisamente più complicata, in quanto, ad esclusione di casi eccezionali, il più delle volte non dispongono di veri e propri BIOS, bensì di assai più limitati e compatti *BootROM*.

Un BootROM esegue il numero minimo di operazioni per poter "puntare" [16] verso una specifica area di storage. Questo vuol dire che il bootloader deve farsi carico di tutte le operazioni necessarie per la prima configurazione dell'hardware al fine di poter caricare il sistema operativo. Tra le altre cose c'è da considerare anche il fatto che per determinate esigenze ingegneristiche, questi dispositivi devono essere in grado di poter avviare il sistema utilizzando sorgenti assai diversificate tra di loro quali memorie NAND/NOR/FLASH, SDCard, USB (host, device ed OTG), SATA (assai raramente), ethernet o addirittura delle connessioni lente quali serali UART ed SPI.

La complessità di tale compito ha portato molti produttori a suddividere il bootloader in due componenti separati, defini-

[14] **B**asic **I**nput **O**utput **S**ystem.

[15] **U**nified **E**xtensible **F**irmware **I**nterface.

[16] Termine gergale derivato dalla programmazione imperativa, sta ad indicare l'operazione di impostare il *program counter* della CPU su un indirizzo di memoria.

ti a loro volta pre-bootloader e bootloader: il primo si occupa delle inizializzazioni basilari quali la CPU, la RAM, i BUS ed un primo dispositivo di storage per "puntare" al bootloader completo. Quest'ultimo, una volta caricato, oltre ad avere un maggior supporto per i vari dispositivi presenti, offre anche una CLI[17], che può essere interattiva o meno, per poter selezionare il dispositivo e i parametri di boot.

I bootloader, per evidenti restrizioni dal punto di vista della dimensione del codice, non possono essere flessibili come un kernel completo. Ciò comporta la necessità di avere un bootloader specifico per ogni differente dispositivo, ottimizzato e specializzato nella configurazione hardware presente.

6.4.1 U-boot

U-boot (nome completo "Das U-boot", testualmente "Il bootloader universale") è un progetto rilasciato con licenza libera GNU GPLv2. Solitamente classificato nella categoria "firmware", è uno dei bootloader più utilizzati nel mondo dei SoC. Esistono numerosi porting di U-boot per un ampio insieme di architetture e SoC differenti. La licenza aperta e la notevole semplicità dal punto di vista della programmazione ne hanno consentito l'integrazione in numerosi progetti, rendendolo de-facto il termine di riferimento nel suo settore.

Le grandi possibilità di modifica hanno reso questo software molto suscettibile, dal punto di vista dell'utilizzo, alle modifiche apportate dai produttori dei vari SoC. Nonostante sia possibile riconoscere alcuni elementi comuni, è assai probabile che una serie di comandi funzionanti sull'istanza del bootloader presente su una eval board di Texas Instruments, non sia eseguita correttamente su una board Freescale o ST.

In alcune implementazioni, un sottoinsieme degli elementi di U-boot è stato utilizzato per costruire il pre-bootloader; sfruttandone in parte il codice a basso livello, il pre-bootloader viene compilato internamente al sorgente di U-boot (es. MLO di Texas Instruments).

[17] **C**ommand **L**ine **I**nterface.

In base a quanto abilitato dal produttore del singolo SoC, U-boot consente di caricare un'immagine di sistema (vedere sez. 7.1.2 e del device-tree da diverse fonti.

All'interno di U-boot, se abilitato, esiste un ambiente CLI per l'esecuzione di comandi, script e per le impostazioni di variabili di ambiente per gestire la fase di boot. U-boot mette a disposizione un tool userspace per la creazione di file di immagine kernel (mkimage) conformi allo "U-boot image format".

U-boot Image Format

Lo U-boot Image format è un insieme di dati, solitamente contenuti nei primi 72 byte del file immagine, contenente alcuni informazioni utili a U-boot per eseguire la fase di caricamento.

A seguire una rapida spiegazione dei parametri configurati nello "U-boot image format"[69].

Target Operating System: specifica il sistema operativo utilizzato (es. Linux, NetBSD, VxWorks, QNX, RTEMS, ARTOS, Unity OS, Integrity);

Target CPU Architecture: specifica l'architettura (es. ARM, AVR32, BlackFin, M68K, Microblaze, MIPS, MIPS64, NIOS, NIOS2, Power Architectureő, SuperH, Sparc, Sparc 64 Bit, Intel x86);

Compression type: specifica la compressione (uncompressed, gzip, bzip2, lzo);

Entry point: il punto in cui "salterà" il bootloader per avviare il kernel;

Image name: campo opzionale;

Image Timestamp: data e ora relative al momento della creazione dell'immagine.

6.5 La fasi di boot

Quanto visto sin ora ci permette di valutare come e quando gli elementi presentati sinora (pre/bootloader, kernel, rootfs, init) interagiscano tra di loro per arrivare a un sistema funzionante.

Si può considerare la fase di boot come una sequenza di attivazioni di più software (anche se per alcuni di essi potrebbe

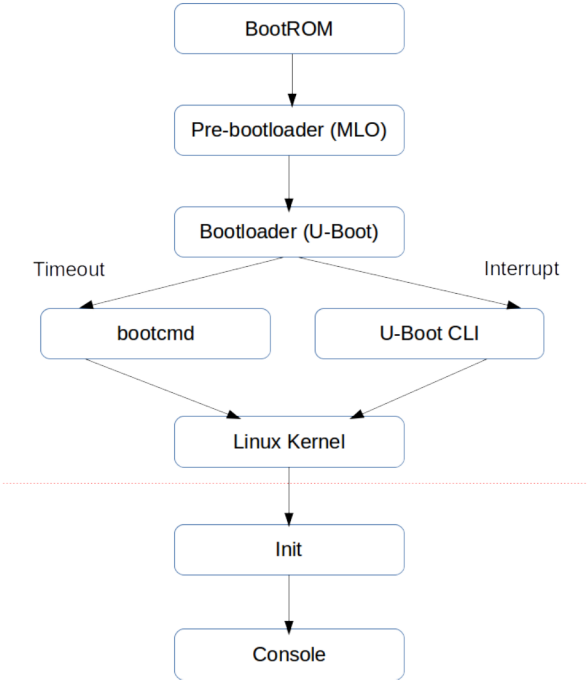


Figura 6.1: Schema basilare delle fasi di boot

essere più attinente la definizione di firmware), ognuno dei quali con determinati compiti specifici (ma non unici), al fine di arrivare ad un sistema funzionante. Dopo le prime tre fasi, quelle inerenti a BootROM, pre-bootloader e bootloader (vedere sezione 6.4), avviene il caricamento del kernel. Il bootloader provvede al passaggio di alcuni parametri di configurazione (in alcuni sistemi tali parametri sono inclusi all'interno della configurazione del kernel). Il kernel, eseguite le operazioni d'inizializzazione e configurazione, "monta"[18] il rootfs, collocato solitamente nella posizione radice "/". Il passo successivo è quello di cercare

[18]Termine appartenente al gergo informatico, indica l'azione di mappatura di un filesystem esterno all'interno dell'alberatura principale del rootfs.

il binario di “init” ed eseguirlo. Da questo momento in poi il controllo rimane ad init che, come già visto in precedenza, effettuerà l'inizializzazione del sistema userspace sino al raggiungimento di una shell o di un'interfaccia grafica. Si tratta per lo più di uno scenario tipico, in quanto, in casi estremi, si può saltare il processo di init per arrivare direttamente ad una shell (tramite parametri di boot). In altri casi, in sistemi progettati per non avere alcun interprete di comandi attivi per motivi di sicurezza, il processo di init porta direttamente all'esecuzione di un particolare software selezionato.

Capitolo 7

Configurazione GNU/Linux

In questa sezione verranno mostrate alcune delle metodologie per arrivare a produrre il software necessario ad un sistema embedded basato su sistema GNU/Linux, attraverso esempi semplificati e, per quanto possibile, generali. Alcuni contenuti di questo capitolo sono scritti considerando da parte del lettore una conoscenza generale sui sistemi GNU/Linux e sulla compilazione di codici sorgente. Come avviene nella maggior parte dei casi pratici, l'ambiente di sviluppo utilizzato negli esempi, è a sua volta un sistema GNU/Linux.

Le evaluation Board utilizzate per questi esempi sono la TI BeagleBone Black (cfr. sezione 4.7.6) e la Imgtec Ci40 (cfr. sezione 4.7.5). Trattandosi di esempi pratici, è doveroso far notare come, al variare delle board, vi possano essere molte differenze nelle procedure da eseguire. Per questo motivo, l'attenzione del lettore deve essere focalizzata sul metodo e non sui singoli comandi, che potrebbero non essere validi in caso di board differenti o di versioni aggiornate dei medesimi hardware e software.

7.1 Ambiente di sviluppo

Per realizzare il nostro sistema esempio è necessario identificare i macro-elementi che andranno a comporre l'ambiente di sviluppo.

Toolchain: Serie di strumenti generici per la compilazione del codice sorgente.

Kernel: il sorgente del kernel e di eventuali driver aggiuntivi;

Buildsystem: Sistema di costruzione del sistema. Tale sistema può contenere anche il kernel e la toolchain.

Nonostante per molte evaluation board esistano numerose "immagini"[1] già pronte di kernel e rootfs, al fine di fornire una maggiore conoscenza e autonomia di sviluppo, si mostreranno alcuni esempi di modifica/compilazione da sorgente.

7.1.1 Toolchain

Trattandosi di compilazione da codice sorgente, è necessario disporre di un compilatore efficiente e completo. Una toolchain è solitamente composta dai seguenti elementi:

compilatore: un software che trasforma il codice sorgente (solitamente testuale, ad esempio in C) nel cosiddetto codice oggetto (comunemente binario ma non ancora eseguibile, spesso riconoscibile dall'estensione ".o");

linker: questo elemento collega (*linking*), fonde i vari codici oggetto prodotti dal compilatore e le librerie utilizzate, trasformandoli in codice eseguibile;

librerie: librerie necessarie alla fase di linking, in particolare per quanto riguarda le librerie d'interfacciamento con il sistema operativo;

debugger: opzionale in alcune toolchain, è lo strumento atto ad eseguire il debug software[2] dei software compilati.

Una toolchain inoltre deve essere "compilata" per l'architettura selezionata, in quanto, in generale, non è possibile utilizzare

[1] Gergale: con il termine "immagini" o "immagini di sistema" si intende una serie di file binari contenenti versioni compilate del software, già testate e pronte (da "bruciare"/"flashare") all'utilizzo

[2] Individuazione dei problemi tramite esecuzione passo passo e *inspection* variabili

la medesima toolchain su architetture differenti. Nel caso dei sistemi embedded la questione si complica ulteriormente, dato l'elevato livello di differenziazione tra le varie architetture e tra i singoli SoC. La necessità di avere una toolchain specifica per l'architettura "obiettivo" (comunemente definita con il termine "target"), porta ad un'ulteriore distinzione.

In tema di compilatori per embedded, sin dagli esordi, si è assistito ad un dominio quasi incontrastato di GCC (GNU C Compiler) e solo di recente i compilatori di nuova generazione (ad esempio LLVM/Clang) hanno cominciato ad offrire soluzioni per questo settore.

Per quanto riguarda la la libc6.3.1 è possibile, durante la fase di compilazione, selezionare quella che verrà utilizzata nel sistema finale. Per la scelta del linker invece, ad eccezione di alcuni casi particolari, si fa in generale riferimento al pacchetto *GNU binutils*[3].

Tra gli elementi di *GNU binutils* si citano:

ld: il linker;

as: assemblatore, traduce linguaggio macchina (o assembler) in codice oggetto.

La fasi di compilazione e linking sono in realtà molto più complesse ed articolate di quanto presentato rapidamente in questo paragrafo. Per una maggiore comprensione, si suggerisce la lettura di [64].

Oltre a quanto detto sopra in merito alle differenti toolchain per differenti architetture, in base a come sono state configurate e compilate, esiste una ulteriore distinzione fra le toolchain: le toolchain *native* e le *cross-toolchain*.

toolchain native: una toolchain nativa è a sua volta stata compilata per un sistema con la medesima architettura hardware/software del target; se nel campo dei sistemi desktop workstation questa è la scelta predefinita, risulta abbastanza nuova e controversa nei sistemi embedded. Da un lato offre una maggiore semplicità per i sistemi di compilazione (a partire dai cosiddetti *autotools*[4]), al con-

[3] Una collezione di strumenti per maneggiare file binari.

[4] Strumenti atti all'automazione della configurazione, compilazione ed installazione software da codice sorgente

tempo però richiede una notevole potenza computazionale non sempre propria dei sistemi embedded, oltre che di un sistema operativo già funzionante sull'architettura selezionata, requisito non sempre facile da soddisfare. Al momento un simile approccio è utilizzato, oltre che per i sistemi Desktop, Workstation e Server, solo per il mantenimento di grandi distribuzioni GNU/Linux per SoC di fascia molto alta, non infrequentemente utilizzando batterie di SoC dedicati solo alla compilazione.

cross-toolchain: una cross-toolchain è costituita dagli stessi elementi di una toolchain nativa, compilati per una architettura ospite (detta anche "host"), ma configurata per produrre binari per una architettura differente. Tale scelta è assai più comune in ambito embedded, in quanto, pur richiedendo un maggior lavoro di configurazione, permette di sfruttare sistemi dotati di maggiore potenza per produrre i binari. Inoltre, non ha come prerequisito un sistema già funzionante per l'architettura target. Per alcune architetture di fascia minore è l'unica alternativa. In molti casi, i produttori del SoC forniscono delle cross-toolchain già pronte, configurate ed ottimizzate per il sistema selezionato.

La compilazione di una cross-toolchain è una operazione complessa, non priva di insidie, costosa dal punto di vista del tempo di sviluppo e non sempre redditizia sul piano delle performance. Si parla di una grande quantità di codice, che deve essere compilato nel corretto ordine, con le versioni corrette ed i giusti parametri di configurazione. Volendo esemplificare, si può pensare ad una grande orchestra dove l'errore di una singola nota di un singolo strumentista può rovinare l'intera esecuzione. Al medesimo modo, un singolo errore di configurazione e compilazione, non facile da individuare e correggere, può compromettere la costruzione della toolchain.

Può risultare una scelta vantaggiosa invece, quando possibile, l'utilizzo di una cross-toolchain precompilata; oltre agli evidenti vantaggi dal punto della semplicità e del minor tempo impiegato, si tratta di toolchain già abbondantemente testate da chi sviluppa l'architettura e successivamente da una larga

base di utenti. Parlando di architetture ARM a partire dalle ARMv7 in poi, le toolchain prodotte da Linaro[5] sono tra quelle più utilizzate. Per quanto riguarda le architetture MIPS, Imagination rilascia diverse versioni del *Codescape GNU Tools*.

Una parametro importante da identificare nel caso delle cross-toolchain è la cosiddetta *target triplet*, ovvero una tripletta di valori testuali che identifica il target di compilazione. In teoria, ma non è da considerarsi come una regola assoluta, segue il *pattern*:

$$< machine > - < vendor > - < operatingsystem > -$$

Dove *machine* rappresenta l'architettura, *vendor* chi rilascia la toolchain e *operating system* il sistema operativo del target. In realtà a volte il pattern non viene seguito, come è visibile in questi esempi:

arm-linux-gnueabihf: Linaro Toolchain 4.9 per ARMv7. La triplet indica un compilatore per architettura ARM 32bit, per sistemi Linux, ed utilizza EABI per hardware dotato di FPU.

arm-linux-androideabi: Android 7 AOSP. La triplet indica il supporto per una architettura ARM 32bit, sistema Linux, con le EABI specifiche per il sistema Android.

mips-linux-gnu: Codescape GNU tools. La triplet indica il supporto per architettura MIPS, su sistema Linux, e tool GNU.

x86_64-linux-gnu: Ubuntu 16.04. Toolchain nativa, per x86_64

Il corretto valore della triplet risulta fondamentale in fase di cross-compilazione, perché, in molti casi, deve essere specificato nella variabile di ambiente **CROSS_COMPILE**, utilizzata in seguito.

[5]Linaro è un gruppo di lavoro che nasce dalla collaborazione di diverse aziende operanti ad alto livello nel settore embedded legate alle architetture ARM (es. ARM, HISILICON, SPREADTRUM, ST, Texas Instruments, Mediatek, ZTE.) per condividere e standardizzare quanto più possibile gli strumenti di sviluppo.

Utilizzo di una toolchain precompilata

Molte distribuzioni GNU/Linux offrono cross-toolchain direttamente scaricabili, ma può essere molto saggio utilizzarne una esterna al sistema in un percorso fuori dai *path* di sistema in modo da poter installare il software necessario senza impattare sul sistema operativo su cui si sta sviluppando. Inoltre, è possibile installare versioni differenti della medesima cross-toolchain, selezionandole di volta in volta senza dover operare con il gestore dei pacchetti del sistema. Questo approccio, relativamente semplice nel caso si utilizzi una toolchain precompilata, richiede solo alcuni accorgimenti in fase di cross-compilazione, ma permette di avere un numero a piacere di cross-compilatori per architetture e revisioni differenti. A seguire un esempio con una cross-toolchain Linaro per architetture ARMv7 e supporto floating-point hardware:

```
$ mkdir tool
$ cd tool
$ wget http://releases.linaro.org/components/
    ↪ toolchain/binaries/4.9-2016.02/arm-linux-
    ↪ gnueabihf/gcc-linaro-4.9-2016.02-x86_64_arm-
    ↪ linux-gnueabihf.tar.xz
$ tar -xvJf gcc-linaro-4.9-2016.02-x86_64_arm-linux-
    ↪ gnueabihf.tar.xz
```

In sostanza, in un percorso a piacere (posto di averne accesso) si crea una cartella dove si scarica e si decomprime la toolchain precompilata.

A questo punto è necessario impostare le necessarie variabili di ambiente per poterla utilizzare:

```
$ export PATH=/shared/projects/example/tool/gcc-
    ↪ linaro-4.9-2016.02-x86_64_arm-linux-gnueabihf/
    ↪ bin:$PATH
$ export CROSS_COMPILE=arm-linux-gnueabihf-
$ export ARCH=arm
```

Con il primo export si aggiunge il percorso dei binari del compilatore nella variabile PATH[6]. Sebbene in molti casi non sia strettamente necessario anteporre il percorso della toolchain ai

[6]Questa variabile contiene i percorsi dei binari utilizzabili all'interno dell'interprete dei comandi

percorsi già presenti nella variabile, ciò può rendersi necessario in caso di più toolchain simili (in particolare con la medesima triplet) presenti nello stesso sistema.

Definendo CROSS_COMPILE, oltre ad indicare la triplet del compilatore in uso, si specifica anche il prefisso nel nome dei file binari della toolchain, molto utile in caso si utilizzino script o Makefile per la cross-compilazione (es. arm-linux-gnueabihf-gcc). Con la variabile ARCH si imposta l'architettura di riferimento del cross-compilatore.

In generale, può essere confortevole unire questi tre *export* in uno script da invocare prima di iniziare la cross-compilazione (es. "toolchain.source").

7.1.2 La compilazione del Kernel Linux

Nonostante i vari buildsystem e BSP offrano sovente una modalità interna ed automatizzata per configurare e compilare il kernel per le specifiche board (come avviene con i layer aggiuntivi per Yocto o sui defconfig specifici di buildroot), è assai frequente che tale configurazione faccia riferimento a kernel collaudati (non recenti, quindi privi degli ultimi aggiornamenti disponibili), testati sulla piattaforma, ma solo con lo stretto necessario per far funzionare l'hardware presente sulla board o il SoC in oggetto. Nel caso di modifiche hardware, quale la aggiunta o la disabilitazione di alcune componenti o nel caso si necessiti di alcuni protocolli di rete non presenti nella configurazione iniziale, è opportuno modificare la versione preconfigurata per generare un nuovo file di configurazione ".config"[7].

Con il medesimo metodo è possibile generare un "_defconfig"[8]; l'utilità di creare e testare un "_defconfig", è che tali file sono facilmente importabili all'interno di un qualsiasi buildsystem, rendendo assai semplice tutti i passaggi di compilazione, di creazione delle immagini di sistema e di test.

[7] Utilizzando gli strumenti di configurazione del sorgente del kernel, il risultato viene salvato, se non differentemente specificato, in un file chiamato ".config", senza nome prima della estensione

[8] File con struttura in tutto e per tutto simile al file ".config", solitamente presente all'interno della cartella "configs" per la specifica architettura (es. "arch/arm/configs/bb.org_defconfig").

Vanilla, Mainline, Longterm e SoC version

La compilazione del kernel richiede la presenza del codice sorgente. Contrariamente a quanto si potrebbe pensare, non esiste un unico sorgente di riferimento. Come è noto, il kernel Linux è rilasciato con licenza GNU GPLv2 dal sito ufficiale degli sviluppatori del kernel (`http://www.kernel.org`).

Sebbene il kernel rilasciato da *kernel.org*, che prende il nome di *Vanilla*[9] sia il punto di riferimento, ovvero quello nel quale vengono definite sia il versionamento che l'architettura generale, è assai raro che venga utilizzato direttamente all'interno di un sistema Desktop o Embedded. Sia i produttori dei BSP (Board Support Package) delle board, sia gli sviluppatori delle distribuzioni Linux Desktop e Server, aggiungono altro codice al kernel vanilla in base ai loro scopi (possono essere sia driver per periferiche non supportate ufficialmente all'interno del kernel, sia aggiornamenti di sicurezza o modifiche funzionali).

Va inoltre fatto notare che *kernel.org*, oltre a rilasciare le versione attuale (prende il nome di *stable* o *last-stable*) o quella nella quale confluiscono tutti i nuovi sviluppi odierni (prende il nome di Mainline) o quelle future non ancora considerate pronte[10], mantiene un certo numero di versioni precedenti del codice fornendo patch di sicurezza e correzioni. Tali versioni, scelte di volta in volta dagli sviluppatori, prendono il nome di *"longterm"*; con tale termine si indicano proprio quelle versioni del kernel che riceveranno aggiornamenti per un periodo prolungato. Per maggiore chiarezza, una longterm riceverà aggiornamenti nel caso venga scoperta una vulnerabilità di sicurezza o un errore di programmazione, ma nessuna nuova funzione o funzionalità proveniente da versioni successive. Alcune versioni mainline, possono diventare delle last-stable, e a loro volta alcune last-stable possono successivamente diventare delle longterm, ma ciò è deciso esclusivamente dal team degli sviluppatori, così come la durata di manutenzione di una longterm. Parlando di evaluation board, o genericamente Soc, data la loro particolare

[9] Con il termine Vanilla si tende ad indicare un sorgente uguale a quello rilasciato, senza ulteriori modifiche.

[10] Possono prendere il nome di *rc* o *Release Candidate*, associata ad un numero progressivo che ne indica il livello di "maturità".

natura hardware, è alquanto improbabile che sia possibile utilizzare la versione vanilla (potrebbero mancare alcuni driver o addirittura potrebbe non essere in grado di ultimare le fasi di boot[11]). Nella maggioranza dei casi, gli sviluppatori del BSP, prendono una versione longterm di riferimento e la modificano con una serie di patch funzionali solo ed esclusivamente per lo specifico SoC. C'è da considerare però che tale codice sorgente non è sempre accessibile dagli utilizzatori finali, o se accessibile, non sempre è in forma completa (alcuni driver potrebbero essere stati rimossi per motivi di licenza o proprietà intellettuale).

Va fatto notare inoltre che tra quando il SoC viene progettato e quando il prodotto basato su quel SoC viene lanciato sul mercato, può passare un lungo periodo di tempo; ciò comporta in molti casi che il prodotto finale utilizzi un kernel di gran lunga più datato (e potenzialmente meno aggiornato) rispetto al livello di sviluppo attuale del kernel.

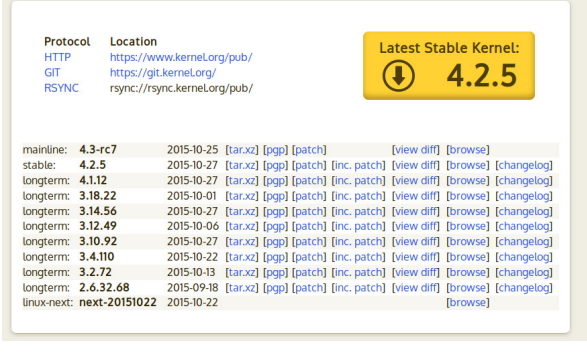


Figura 7.1: Pagina del sito kernel.org

L'utilizzo di longterm è fondamentale in campo industriale, perché, dal punto di vista dello sviluppo di un prodotto, non sarebbe possibile seguire costantemente il continuo sviluppo del

[11] Le motivazioni di ciò sono per lo più da ricercarsi nell'uso di IP con forti limiti dal punto di vista contrattuale, dallo scarso interesse del produttore del SoC a mantenere aggiornati i driver o per problemi di qualità dei driver considerata insufficiente dai manutentori del kernel, impedendone l'inserimento nel kernel mainline.

kernel e delle API. Per chiarire ulteriormente questo concetto, basti pensare che un sistema sviluppato su una longterm più datata potrebbe non funzionare perfettamente se eseguito con un kernel più recente e viceversa. In fase di sviluppo di un sistema ex-novo, i kernel-headers (gli header file del sistema operativo) devono essere della medesima versione del kernel utilizzato.

Download ed ambiente di compilazione

```
$ source tool/toolchain.source
$ git clone https://github.com/beagleboard/linux.git
$ cd linux
```

La chiamata del comando "source" consente di richiamare lo script descritto durante l'installazione della toolchain, esportando le variabili nella shell corrente. Con il comando `git`[12] `clone`[13] viene scaricato il codice sorgente del kernel (una versione community derivata da una release stabile) con modifiche specifiche per questa evaluation board dal noto repository pubblico github. Questa versione non gode dello sviluppo e della fase di testing del kernel rilasciato ufficialmente con la board.

L'utilizzo di una versione presa da un *repository* pubblico, oltre ai naturali vantaggi derivati dall'uso di un SCM, faciliterà notevolmente l'opera di integrazione successiva con i build system.

Una volta finita la fase di clone (assai più onerosa in termini di download rispetto al semplice pacchetto compresso), verrà creata una cartella "linux".

Configurazione

[12]Git è un software della categoria SCM (Source Code Management) sviluppato inizialmente da Linus Torvalds, l'autore del quasi omonimo kernel. Come altri software di questa fascia consente di controllare lo sviluppo del software, con una gestione avanzata su più rami di sviluppo. La sua natura volutamente distribuita, particolarmente adatta a codici sorgenti a sviluppo condiviso, lo ha reso la scelta predefinita nel mondo del software libero.

[13]Il comando 'git clone' crea una copia locale del repository sul branch predefinito, solitamente etichettato "master".

```
$ make bb.org_defconfig
```

La base di partenza è un file defconfig. Questo specifico defconfig non si trova nella versione "mainline" del kernel, ma è una specifica aggiunta degli autori di questa versione relativa all'hardware utilizzato. Una volta eseguito il comando, il file di configurazione finale verrà salvato nella directory corrente nel file "nascosto"[14] ".config". A seguire, tale configurazione può essere modificata con il seguente comando:

```
$ make menuconfig
```

Si presenta un menu sviluppato con `ncurses`[15] per la selezione dei parametri di configurazione. Non è banale stabilire una linea guida valida per tutte le possibili casistiche, ma è tuttavia possibile però definire dei comportamenti "tipici" sulla base delle scelte svolte durante questa fase.

Approccio Monolitico o Modulare[16]:

Monolitico: con questo termine si indica le tendenza a porre i driver e i supporti correlati all'interno dell'immagine del kernel, limitando o eliminando completamente l'uso dei moduli. Fino all'avvento del Linux Kernel 2.2, con l'introduzione dei moduli esterni, questa era l'unica scelta possibile. Nel settore embedded è generalmente preferita questa modalità, in quanto riduce fortemente i problemi relativi al caricamento dinamico dei moduli (ad esempio la necessità di accedere al *rootfs* in fase di boot). Come conseguenza ovvia di questa scelta risultano le maggiori dimensioni dell'immagine del kernel.

[14] La dicitura nascosta potrebbe essere fuorviante: nei sistemi Unix/-Linux in genere, anteponendo il carattere "." ad un nome file/cartella quest'ultimo non verrà visualizzato nelle viste relative a file (es. comando `ls`), a meno di opzioni. Ciò non ne impedisce l'accesso o la visualizzazione diretta.

[15] `https://www.gnu.org/software/ncurses/ncurses.html`

[16] Tale terminologia può risultare ingannevole, in quanto è la medesima utilizzata per descrivere il modello architetturale del kernel. In questo contesto si riferisce semplicemente alle scelte relative alla compilazione dei moduli, ma Linux rimane un kernel basato su architettura monolitica, vedere sezione 6.1.1.

Modulare: con "modulare" si intende invece la scelta di porre driver e supporti, per quanto possibile, all'esterno dell'immagine del kernel, sfruttando il meccanismo dei moduli caricabili dinamicamente. Tale metodologia, solitamente tipica per i sistemi desktop, offre una maggiore flessibilità nella fase di caricamento (un modulo viene caricato solo se necessario) oltre ad una minor dimensione dell'immagine del kernel, ma richiede alcune particolari attenzioni. In generale, non si può affermare che il kernel e il root-filesystem contenente i rispettivi moduli si trovino nella stessa area di storage; è importante evitare che il driver atto all'uso da parte del kernel del root-filesystem non risieda nel filesystem stesso (una dipendenza circolare dalla quale ne risulterebbe un "kernel panic"). Alcuni supporti del kernel non sono comunque compilabili come moduli.

Sempre in tema di tipologie di configurazione di configurazione, si parla genericamente di due differenti approcci che prendono il nome di "Custom" e "General Purpose":

Custom: con questo termine si indica la scelta di inserire all'interno del kernel esclusivamente i supporti considerati essenziali all'hardware o ai protocolli previsti. Ciò produce un kernel più leggero, ma contemporaneamente poco flessibile all'aggiunta di nuovo hardware o alla richiesta di nuovi protocolli. Tale scelta è generalmente legata a contesti embedded, o comunque molto specializzati. Non è infrequente che questo modello vada di pari passo con una compilazione di tipo "monolitica";

General Purpose: modello antitetico al "custom", è il tipo di scelta che viene fatta per massimizzare la compatibilità del sistema con il maggior numero possibile di hardware e protocolli supportati. Tipico delle distribuzioni Linux per sistemi desktop, questo tipo di impostazione viene spesso associata al modello di configurazione modulare.

Compilazione

```
$ make uImage LOADADDR=80008000
```

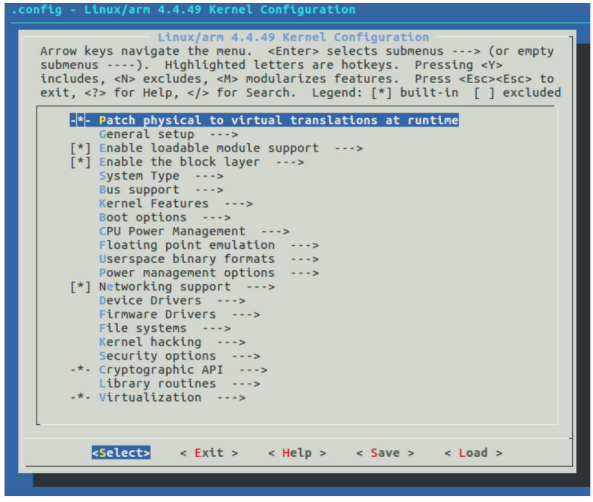


Figura 7.2: Linux Kernel Menuconfig

Con questo comando si avvierà la compilazione del kernel (ma non dei moduli) specificando il tipo d'immagine da utilizzare ed eventuali parametri. L'immagine prodotta non sarà quella utilizzata in fase di rilascio, ma sarà utile per la fase di testing iniziale e di creazione del "defconfig" definitivo.

Il LOADADDR è un valore assai importante, che indica il punto in memoria da cui il kernel inizierà la sua esecuzione. Tradizionalmente è posto 32Kb (0x8000) al di sopra della prima area di memoria disponibile, per lasciare spazio per l'inserimento dei parametri ATAGs[17]. In generale è raro trovarsi a calcolare il LOADADDR, ed è di solito fornito dal produttore del SoC o della Board nelle guide di riferimento o negli esempi di configurazione di U-boot.

```
$ make modules
$ make modules_install INSTALL_MOD_PATH=$TEMP_ROOTFS
```

[17] Simili a variabili di ambiente, passate dal bootloader al kernel. Tale sistema non è universalmente supportato.

Con questa fase si lancia la compilazione dei moduli, e successivamente l'installazione dei moduli compilati in un percoso specificato dalla variabile "TEMP_ROOTFS"[18]. TEMP_ROOTFS è in realtà una variabile d'ambiente preimpostata contenente il percorso dell'ambiente di test (vedere sezione A.1).

Le immagini del kernel

Si possono suddividere le tipologie d'immagine del kernel in base alle informazioni per specifici bootloader o al tipo di compressione utilizzata:

Image: immagine binaria non compressa del kernel. L'uso di questo tipo di soluzione è scarsamente raccomandabile se non per motivi di debug. Il bootloader non è in grado di sfruttare le massime potenzialità del sistema, ciò comporta che la maggiore dimensione dovuta alla mancata compressione impatterebbe sensibilmente sui tempi di avvio. Nel caso di dispositivi embedded di piccole dimensioni ciò è da escludere completamente;

zImage: immagine binaria compressa auto-estraente[19] con algoritmo gzip. zImage è utilizzato come scelta di default per molti sistemi embedded: sebbene gzip non garantisca i medesimi risultati di altri software di compressione, risulta computazionalmente semplice, e quindi non pesante dal punto di vista dei tempi di esecuzione;

bzImage: medesimo meccanismo di zImage, ma con algoritmo bzip2: più efficace dal punto di vista della compressione, risulta più oneroso computazionalmente. È stata la scelta di default per molti sistemi desktop/server in molte distribuzioni;

uImage: uImage è il formato predefinito per le immagini kernel da utilizzare con il bootloader U-boot (vedere la sezione U-boot image formats 6.4.1). Di per se non specifica il

[18] Senza il parametro INSTALL_MOD_PATH i moduli non verrebbero installati nel sistema target ma in quello host, che essendo un'architettura differente potrebbe subirne un danno dal punto di vista OS.

[19] I file compressi auto-estraenti sono da considerarsi dei file che al momento dell'esecuzione decomprimono il loro contenuto in un'aria di memoria.

tipo di compressione, ma aggiunge un header di informazioni all'immagine utili allo stesso u-boot per il corretto caricamento ed esecuzione dell'immagine. In base al makefile, questo tipo di immagine è solitamente compresso in gzip.

Nell'ottica di avere un'immagine kernel funzionante, con i relativi moduli, possono essere necessarie numerose ripetizioni della procedura, con un numero considerevole di prove nell'ambiente di test. Ottenuta un'immagine funzionante si può passare alla costruzione del rootfilesystem.

Nella fase iniziale di configurazione dell'ambiente sono già stati impostati i parametri fondamentali del sistema (compilatore, libreria C, impostazioni di architettura). In seguito è necessario specificare tutti gli altri parametri integrando la versione del kernel appena testata.

Anche in questa fase è necessario operare un'accurata selezione sui pacchetti a disposizione per il sistema: nella maggior parte dei casi, lo spazio a disposizione per lo storage risulta essere un parametro fondamentale quanto i limiti computazionali del dispositivo.

7.1.3 Build system

Un build system può essere definito come l'insieme di tool, sistemi di revisione codici sorgente, script, Makefile, eseguibili e librerie atti alla compilazione di pacchetti software che andranno a comporre il sistema embedded finale. Un build system può comprendere o meno la toolchain, il kernel ed i componenti del root filesystem: si tratta di una visione piuttosto ampia di quello che può fare un build system, ma ciò è dovuto prevalentemente al fatto che sono assai differenziati tra loro.

LFS: acronimo di *Linux From Scratch*, tende ad indicare tutti gli script e le procedure operative per compilare un sistema operativo GNU/Linux e i tool necessari alla compilazione senza un vero e proprio build system. Didatticamente molto valido, non rappresenta un approccio realmente utilizzabile per ottenere un rootfs utilizzabile su un prodotto finale in tempi contenuti. Pur non essendo utiliz-

zato, la conoscenza delle procedure operative di LFS può essere di grande aiuto in fase di risoluzione problemi negli altri build system.

OpenEmbedded: un sistema di build, basato su *layer* (strati) e su *recipes* (ricette). All'interno dei vari layer sono presenti le varie recipes, sotto forma di file testuali, contenenti le istruzioni ed i parametri per compilare i vari elementi del sistema. Avendo una struttura molto flessibile, si presta particolarmente alla personalizzazione ed al lavoro in team. Da OpenEmbbeded sono derivati progetti quali SHR, Angstrom (utilizzata per anni da costruttori come Texas Instruments come ambiente di default) e Yocto.

Yocto: lo Yocto Project[58], promosso dalla Linux Foundation e fortemente basato su OpenEmbbeded, nasce con lo scopo di semplificare la creazione di distribuzioni linux per sistemi embedded, indipendentemente dall'architettura di destinazione. Yocto si basa sul concetto di "Layer" ovvero di una serie di strati non necessariamente in gerarchia verticale, contenenti della meta informazioni sulla compilazione del sistema e dei suoi componenti. Se da un lato quindi può risultare più onerosa la scelta di yocto rispetto ad altri sistemi di build, dall'altro Yocto è quello che si presta maggiormente ad essere utilizzato con software SCM (Software Control Management), quindi adatto per quei progetti realizzati in team di sviluppo numerosi.

Buidroot: buildroot[12] è un insieme di script, Makefile a patch pensati per generare un sistema operativo Linux su piattaforme embedded. Buildroot può generare la cross-toolchain, il root filesystem, le immagini del kernel e dei vari bootloader. Nell'ottica di sviluppo di buildroot ci sono da sempre i sistemi embedded più piccoli e semplici (inizialmente era utilizzato come semplificazione per gli sviluppatori che lavoravano su ambienti uKernel e ulibc, per sistemi privi di MMU) e la possibilità di offrire un valido ambiente di sviluppo manutenibile da un singolo sviluppatore. Supporta una grande varietà di architetture, quali ARC, ARM (32/64 LE/BE), Blackfin, Microblaze, Mips

(32/64 BE/LE), Nios, PowerPC (32/64), SuperH, Xtensa ed ovviamente le architetture x86(32/64)[20].

AOSP: Android utilizza un suo sistema di build (definito AOSP, acronimo di Android Open Source Project), utilizzabile esclusivamente per realizzare immagini di sistema Android. Anche a causa delle enormi dimensioni del progetto, ad oggi la personalizzazione di un build di Android può risultare macchinosa e complicata. La compilazione del sistema, ma non del kernel, viene eseguita con una serie di script interni e, salvo eccezioni, delle cross-toolchain precompilate.

gbs: acronimo per git-build-system è sistema di build che utilizza un approccio completamente differente dai suoi predecessori: basato su git, permette di fare compilazione "nativa su architettura differente" (es. compilazione per ARM su PC x86_64) grazie all'uso intensivo di un emulatore software (qemu). La fase di compilazione produce una serie di pacchetti "rpm"[21] sulla base di "ricette" scritte su file testuali con estensione ".spec"[22]. Tale approccio, apparentemente molto versatile, soprattutto per le ottime possibilità di *versioning* legate a git e per la semplificazioni di una compilazione nativa, risulta però essere molto lento in fase di compilazione, proprio in relazione all'enorme richiesta computazionale dovuta ad una compilazione eseguita su un emulatore. Gbs è utilizzato come build system predefinito per Tizen[23].

Questi build system si distinguono, oltre che per target, anche per l'hardware necessario per compiere il loro scopo. Se

[20] Con la dicitura 32/64 si intende il supporto per architetture a 32bit e a 64bit. Con la dicitura BE/LE si intende il supporto per le architetture Big Endian e Little Endian.

[21] Acronimo di Red Hat Package Manager, che rappresenta contemporaneamente la tipologia di pacchetto, l'estensione del file e il tool di installazione dei pacchetti

[22] ciò è molto comune nelle distribuzioni GNU/Linux dette "rpm based", quali RedHat, SUSE, Fedora ed openSUSE

[23] Sistema opensource di derivazione GNU/Linux, legato all'ecosistema Samsung, era stato ideato inizialmente per smartphone, per poi essere adattato as utilizzi Automotive (IVI, In-Vehicle-Infotainment), Wearable (come smartwatch e simili) e Notebook di piccola taglia

Buildroot e OpenEmbbeded possono essere eseguiti su Hardware più modesto, ma comunque non troppo vetusto, lo stesso non si può dire per AOSP e gbs, che necessitano di grandi quantità di potenza computazionale, memoria RAM, e spazio di storage.

Yocto per Beaglebone

A seguire, un esempio pratico di generazione di un rootfs con il buildsystem Yocto. Utilizzando la versione scaricabile dal sito, le board della famiglia Beaglebone sono già supportate nativamente, quindi non è necessario aggiungere alcun layer. Ciò non è vero nella maggior parte dei casi, dove invece è necessario scaricare ed aggiungere ulteriori layer contenenti sia elementi della toolchain, che elementi legati ai driver.

In una prima fase è necessario creare una cartella di lavoro, scaricare il file compresso (o alternativamente la versione git) e decomprimerlo al suo interno.

```
$ mkdir yocto
$ cd yocto
$ wget http://downloads.yoctoproject.org/releases/
    ↪ yocto/yocto-2.0.3/poky-jethro-14.0.3.tar.bz2
$ cd poky-jethro-14.0.3/
```

A questo punto, saranno visibili le cartelle contenenti i vari layer, tutte caratterizzate dal prefisso *"meta"*, ma non la cartella *build* necessaria per la configurazione e la compilazione. In questa fase iniziale è possibile aggiungere layer supplementari o modificare quelli esistenti. La cartella build vine infatti creata al momento dell'esecuzione dello script *oe-init-build-env*, che oltre alla creazione della cartella, copia i file di configurazione standard al suo interno e imposta alcune variabili di ambiente necessarie.

```
$ source oe-init-build-env

### Shell environment set up for builds. ###

You can now run 'bitbake <target>'

Common targets are:
```

```
    core-image-minimal
    core-image-sato
    meta-toolchain
    meta-ide-support

You can also run generated qemu images with a command
    ↪ like 'runqemu qemux86'
```

Da notare come l'output dello script suggerisca quali siano i target disponibili per l'attuale configurazione:

core-image-minimal: compilazione degli elementi base della toolchain e di un rootfs minimale;

core-image-sato: oltre a quanto già fatto nella *core-image-minimal*, aggiunge altri software tra cui una interfaccia grafica personalizzabile.

Eseguito lo script, ci si troverà direttamente nella directory *build*. A questo puntò è necessario scegliere la configurazione: con un qualsiasi editor di testo, va modificato il file *local.conf* presente in *textbfbuild/conf*. Da notare in particolare la sezione relativa alla scelta della *"machine"*.

```
#MACHINE ?= "qemuarm"
#MACHINE ?= "qemuarm64"
#MACHINE ?= "qemumips"
#MACHINE ?= "qemuppc"
#MACHINE ?= "qemux86"
#MACHINE ?= "qemux86-64"
#
# There are also the following hardware board target
    ↪ machines included for
# demonstration purposes:
#
#MACHINE ?= "beaglebone"
#MACHINE ?= "genericx86"
#MACHINE ?= "genericx86-64"
#MACHINE ?= "mpc8315e-rdb"
#MACHINE ?= "edgerouter"
#
# This sets the default machine to be qemux86 if no
    ↪ other machine is selected:
MACHINE ??= "qemux86"
```

La scelta di default è di compilare il tutto per un ambiente emulato con qemu e per architettura x86 a 32bit, ideale per testare, ma non adatto per avere un rootfs funzionante sulla

board. Per cambiare, è sufficiente "commentare"[24] con il carattere "#" la linea ***MACHINE ??= "qemux86"*** e decommentare quella relativa alla board in utilizzo (in questo caso ***MACHINE ?= "beaglebone"***. Così come le *recipes*, anche le *machine* sono definite all'interno dei layer. Tali definizioni, oltre a comprendere i dettagli legati all'architettura, possono anche definire alcuni pacchetti aggiuntivi necessari alla generazione di una immagine funzionante, oltre che il tipo di immagini prodotte.

A seguire è possibile osservare in quella relativa alla *beaglebone*, che si trova nel file *beaglebone.conf* nella cartella *meta-yocto-bsp/conf/machine/beaglebone.conf*:

```
#@TYPE: Machine
#@NAME: Beaglebone machine
#@DESCRIPTION: Machine configuration for http://
    ↪ beagleboard.org/bone and http://beagleboard.org
    ↪ /black boards

PREFERRED_PROVIDER_virtual/xserver ?= "xserver-xorg"
XSERVER ?= "xserver-xorg \
           xf86-input-evdev \
           xf86-input-mouse \
           xf86-video-modesetting \
           xf86-input-keyboard"

MACHINE_EXTRA_RRECOMMENDS = "kernel-modules kernel-
    ↪ devicetree"

EXTRA_IMAGEDEPENDS += "u-boot"

DEFAULTTUNE ?= "cortexa8hf-neon"
include conf/machine/include/tune-cortexa8.inc

IMAGE_FSTYPES += "tar.bz2 jffs2 wic"
EXTRA_IMAGECMD_jffs2 = "-lnp"
WKS_FILE = "sdimage-bootpart.wks"
IMAGE_INSTALL_append = "kernel-devicetree kernel-
    ↪ image-zimage"
do_image_wic[depends] += "mtools-native:
    ↪ do_populate_sysroot dosfstools-native:
    ↪ do_populate_sysroot"
```

[24] Termine gergale che indica l'utilizzo di uno o più caratteri speciali che indicano al *parser* o all'interprete di turno di non considerare determinate porzioni di codice.

```
SERIAL_CONSOLE = "115200 ttyO0"

PREFERRED_PROVIDER_virtual/kernel ?= "linux-yocto"
PREFERRED_VERSION_linux-yocto ?= "4.8%"

KERNEL_IMAGETYPE = "zImage"
KERNEL_DEVICETREE = "am335x-bone.dtb am335x-boneblack
    ↪ .dtb"
KERNEL_EXTRA_ARGS += "LOADADDR=${UBOOT_ENTRYPOINT}"

SPL_BINARY = "MLO"
UBOOT_SUFFIX = "img"
UBOOT_MACHINE = "am335x_evm_config"
UBOOT_ENTRYPOINT = "0x80008000"
UBOOT_LOADADDRESS = "0x80008000"

MACHINE_FEATURES = "usbgadget usbhost vfat alsa"

IMAGE_BOOT_FILES ?= "u-boot.${UBOOT_SUFFIX} MLO"
```

Di particolare rilievo, oltre alle configurazioni già elencate in precedenza, anche quella relativa al *tuning*[25] per il Cortex-A8 utilizzato dalla Beaglebone. Curiosando nella cartella ***meta/-conf/machine/include/*** è possibile vedere tutti gli include relativi alle varie architetture supportate. In generale, al netto di alcune eccezioni, può non essere produttivo modificare questi file.

Una volta selezionata la *machine* è possibile procedere alla fase di compilazione e finalizzazione delle immagini di sistema:

```
bitbake -k core-image-minimal
```

L'opzione "-k", consente di continuare, per quanto possibile, la compilazione, anche in caso di errori. Anche se utilizzata di frequente, è in generale poco raccomandabile in fase di sviluppo. Dopo la fase di "build", che può tranquillamente durare diverse ore, si avranno le immagini di sistema all'interno della cartella *build/tmp/deploy/images/beaglebone*. Se la compilazione è avvenuta con successo sarà possibile notare i seguenti file:

- **MLO-beaglebone-v2015.07**: immagine binaria del pre-bootloader

[25] In questo contesto il termine *tune* si riferisce alla generazione di codice binario per una specifica architettura/cpu

- **u-boot-beaglebone-v2015.07.img**: immagine binaria del bootloader (u-boot)
- **zImage-4.1.15-beaglebone.bin**: immagine binaria (formato zImage) del kernel
- **zImage-4.1.15-am335x-boneblack.dtb**: file binario del device-tree
- **modules-4.1.15-beaglebone.tgz**: file compresso contenente alcuni moduli del kernel
- **core-image-minimal-beaglebone.rootfs.jffs2**: file immagine del rootfs in formato jffs2 pronto per essere installato sulla eMMC della board.
- **core-image-minimal-beaglebone.rootfs.tar.bz2**: file compresso contenente tutti i file del rootfs, utilizzabile per usi alternativi (es. NFS)

Nota: alcuni di questi file potrebbero presentare dei prefissi o suffissi indicanti il commit git a cui fa riferimento la *build* (es. +gitAUTOINC+33711bdd4a-r0) o un *timestamp*[26] che indica la data di compilazione dell'immagine (es. -20170214163534)

7.2 Buildroot

7.2.1 Inizializzazione dell'ambiente

```
$ cd ~/
$ mkdir build_esempio
$ cd build_esempio
$ wget http://buildroot.uclibc.org/downloads/
    ↪ buildroot-2015.08.1.tar.gz
$ tar -xvzf buildroot-2015.08.1.tar.gz
$ cd buildroot-2015.08.1
```

I primi tre passaggi sono necessari per la creazione di una cartella di lavoro. In questa cartella verranno sviluppate tutte le singole parti del progetto, per poi essere assemblate in un unico sistema di build. Il comando `wget` consente di scaricare uno specifico pacchetto da Internet all'interno della cartella

[26] Una forma numerica in cui si rappresenta una data in forma non ambigua

di lavoro. Nella specifico, la versione 2015.08.1 in formato compresso "tar.gz". Il comando `tar` utilizzato in questo modo estrae il pacchetto compresso generando la cartella dentro la quale si troverà il buildroot (buildroot-2015.05).

All'interno della cartella di buildroot sono da identificare alcune cartelle che verranno utilizzate successivamente:

dl: la cartella dove vengono scaricati i sorgenti compressi

fs: i makefile per la generazione delle singole immagini dei root filesystems

arch: file di configurazione per le differenti architetture hardware

board: contiene delle sottocartelle dedicate al supporto ad un numero limitato di evaluation board, come script di configurazione e defconfig kernel/uboot

configs: contiene i defconfig per le evaluation board più diffuse, oltre che per alcune architetture specifiche

output: una volta effettuata la fase di "build", contiene le immagini di sistema generate, i binari della toolchain, e il rootfs temporaneo

system: i file di configurazione per la generazione del sistema

toolchain: contrariamente a quanto potrebbe far pensare il nome, questa cartella non conterrà la toolchain, ma solo i file di configurazione per la sua generazione/download

docs: file di documentazione

Buildroot Board Directory

All'interno di buildroot, oltre ai già citati "defconfig", sono presenti alcune cartelle specifiche per ogni differente piattaforma supportata. Si trovano tutte all'interno della cartella "board", e contengono alcuni elementi aggiuntivi e personalizzabili, non presenti nella normale configurazione.

```
$ ls board/beaglebone/
genimage.cfg  linux-4.1-sgx.fragment  patches  post-
    ↪ image.sh  readme.txt  uEnv.txt
```

linux-4.1-sgx.fragment: frammento del .config del kernel per abilitare alcuni componenti (nello specifico la gpu)

uEnv.txt: in questo file è possibile specificare alcune opzioni e variabili di ambiente di U-boot (vedere sezione 7.2.4)

post-image.sh: questo script fa eseguire alcune operazioni a buildroot, dopo la creazione delle immagini di sistema (vedere sezione 7.2.3)

patches: cartella contenente alcune patches per i codici sorgente dei vari elementi del sistema

Non è infrequente, laddove non specificato dal .config di buildroot, trovare in questa cartella anche .config del kernel. Bisogna considerare che questa cartella può essere modificata o utilizzata come base per crearne una versione personalizzata alle esigenze.

7.2.2 Configurazione di buildroot

Una volta entrati all'interno della cartella buildroot è possibile lanciare il comando "make beaglebone_defconfig che preconfigura l'ambiente di compilazione seguendo le specifiche scritte nel file "beaglebone_defconfig":

```
$ make beaglebone_defconfig
$ make menuconfig
```

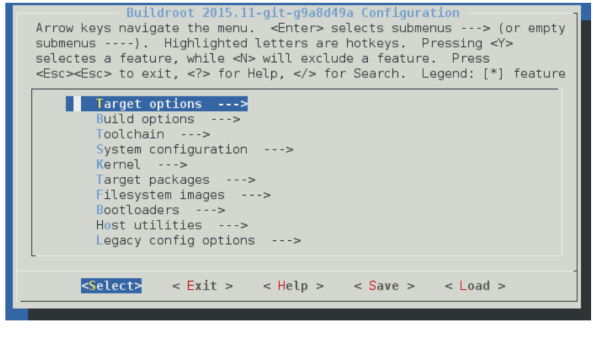


Figura 7.3: Menu principale buildroot

L'uso dei file definiti "defconfig" è abbastanza comune, sia in buildroot, sia nel sorgente del kernel, da cui tale struttura è

stata ispirata: Questi file contengono delle configurazioni basilari (solitamente bastevoli alla creazione di un sistema minimale, ma non sempre sufficienti ai fini di un progetto completo) specifici per alcune evaluation board, device embedded o sistemi di emulazione (es. qemu). È giusto ricordare che in relazione all'hardware presente sul mercato, solo una piccola parte è supportata con un proprio defconfig. In molti casi, in assenza di uno specifico defconfig, è necessario ricreare un file di configurazione da zero.

Inoltre un defconfig può essere una versione generica per un insieme di dispositivi, quindi contenere elementi non utili all'hardware specifico o non perfettamente aderenti al progetto considerato. In generale è considerato fondamentale procedere ad una configurazione manuale di buildroot, eseguendo successivamente il comando "make menuconfig"[27].

Osservando il menu principale di buildroot è possibile distinguere le seguenti voci:

Target options: in questa sezione vengono specificate la caratteristiche dell'architettura e del formato binario

Build options: opzioni relative alle sezioni di download, compilazione e debug del sistema prodotto

Toolchain: in questa sezione viene selezionato il tipo di toolchain, e nel caso di una toolchain costruita da zero anche le versioni della libreria C e delle cosiddette "binutils"

System Configuration: parametri di configurazione base del sistema, tra cui il sistema di init, le opzioni di boot e script di pre e post configurazione

Kernel: creazione di un'immagine del kernel, specificando la locazione del sorgente, uno specifico defconfig ed il tipo di immagine da utilizzare

Target Packages: selezione dei pacchetti che saranno presenti all'interno del sistema finale

Filesystem Images: selezione dei diversi modi di generare le immagini del rootfs.

[27]Sia per quanto riguarda buildroot che per il sorgente del kernel linux, menuconfig è un tool binario compilato al momento dell'esecuzione del comando "make menuconfig". Per essere compilato necessita dei pacchetti ncurses (libreria per menu grafici da console testuale, derivata da curses) ed ncurses-devel (gli header file per lo sviluppo).

Bootloaders: è possibile selezionare la compilazione di uno specifico bootloader (vedere sezione U-boot)

Host utilities: è possibile compilare alcune applicazioni locali per il sistema host (in molti casi non è necessario operare in questa sezione)

Legacy config options: in questa sezione sono presenti alcune opzioni rimosse dalla versione attuale di buildroot, ma configurabili per motivi di retro-compatibilità

Data l'estrema variabilità dei possibili scenari, è assai complicato poter definire delle linee guida universali: in alcuni casi si predilige realizzare una configurazione singola comprensiva di tutto, in altri invece si procede per passaggi successivi, aggiungendo incrementalmente sia i pacchetti e le opzioni di buildroot che sorgenti software esterne da integrare.

Kernel Linux in buildroot

In questa parte del menu di configurazione è possibile utilizzare quanto sviluppato in precedenza, utilizzando gli stessi elementi testati separatamente all'interno di buildroot.

```
[*] Linux Kernel
      Kernel version (Custom Git repository)  --->
(https://github.com/beagleboard/linux.git) URL of custom repository
(94a60fd127ea564a483c577d277d68543b250c21) Custom repository version
()    Custom kernel patches
      Kernel configuration (Using a custom (def)config file)  --->
(board/meshsystem/linux-4.1.2.config) Configuration file path
()    Additional configuration fragment files
      Kernel binary format (uImage)  --->
(0x80008000) load address (for 3.7+ multi-platform image)
[*]   Build a Device Tree Blob (DTB)
        Device tree source (Use a device tree present in the kernel.)  --->
(am335x-boneblack) Device Tree Source file names
[ ]   Install kernel image to /boot in target
      Linux Kernel Extensions  --->
      Linux Kernel Tools  --->
```

Figura 7.4: Menu di configurazione del kernel in buildroot

Come evidenziato dalla figura 7.4, si utilizza il *repository git* visto in precedenza, facendo puntare il sorgente al medesimo ID di commit[28] utilizzando il file di configurazione precedentemente sviluppato e testato e con il medesimo LOADADDR. Tra

[28] Ogni *commit* ha un identificativo unico (*commit id*) composto da cifre e lettere.

le opzioni è possibile anche specificare il "device-tree" (vedere sezione 6.2.2) da utilizzare.

Nel defconfig originale utilizza il tipo di immagine "zImage" che, pur essendo supportato dalle odierne versioni di U-boot, può risultare una scelta meno saggia rispetto al formato nativo di U-boot (uImage), scelto in questa configurazione.

Target Packages

La scelta dei pacchetti da installare è di fatto una delle prime discriminanti nel determinare la dimensione finale del rootfs finale. In questo caso è importante limitare il più possibile il numero di pacchetti installati, senza però sacrificare la funzionalità prevista.

Ultimate le modifiche è necessario uscire dal "menuconfig" salvando il file di configurazione. Il file di configurazione finale (.config), potrà essere copiato all'interno della cartella "config" in modo tale da diventare a sua volta un defconfig riutilizzabile automaticamente per ogni eventuale modifica e personalizzazione:

```
$ cp .config configs/esempio_defconfig
```

7.2.3 Compilazione del rootfs

Una volta che gli step di configurazione sono ultimati, è possibile passare alla compilazione:

```
$ make source
$ make
```

Il comando "make source" scarica automaticamente[29] i sorgenti che dovranno essere ricompilati. Il comando "make" avvia la compilazione del sistema, delle dipendenze e crea una prima immagine del sistema e del kernel.

Una volta ultimata la compilazione, oltre ad una versione preliminare del sistema, sarà disponibile anche la toolchain utilizzabile esternamente. La toolchain binaria non sarà disponibile nei "path" del sistema host: per questo motivo è importante

[29] È necessario un collegamento ad Internet.

creare uno script che al bisogno esporti determinate variabili di ambiente:

```
$ cd ..
$ touch toolchain.source
$ echo "export ARCH=arm" > toolchain.source
$ echo "export CROSS_COMPILE=arm-linux-gnueabihf-">>
    ↪ toolchain.source
$ echo "export PATH=\$PATH:\$HOME/mesh_system/
    ↪ buildroot-2018.02/output/host/opt/ext-
    ↪ toolchain/bin"  >> toolchain.source
```

Una volta ultimata questa fase, in base alle scelte fatte in fase di configurazione, all'interno della cartella "out/images/" saranno presenti alcuni file che rappresentano i vari elementi del sistema:

- uboot.img
- uImage
- rootfs.tar.gz
- rootfs.ext2.gz
- MLO
- uEnv.txt

In questo esempio, oltre agli elementi del bootloader in formato binario (MLO e uboot.img), sono presenti anche una immagine in formato uBoot del kernel (uImage), il file uEnv.txt, già visto in precedenza e contenente parametri aggiuntivi per il bootloader, e altri due file caratterizzati dalla radice "root". ognuno dei quali contenente un root filesystem (uno in formato compresso, l'altro sotto forma di file immagine formattata `ext2`).

7.2.4 Variabili di ambiente U-boot

Durante questa trattazione si è fatto più volte riferimento alle variabili di ambiente U-boot, ed in particolare, facendo riferimento agli esempi su TI Beaglebone, al file uEnv.txt. È in primo luogo importante far notare che nonostante U-boot sia presente in moltissimi sistemi embedded, non è assolutamente certo che le sue variabili vengano salvate su un file di testo esterno: in molti casi possono essere salvati in una porzione specifica del dispositivo di memoria, in un formato misto (un header binario

seguito da caratteri di testo.). Esistono altri sistemi che utilizzano sempre il file di testo per i parametri, ma con nome ed estensione differenti.

Le variabili di ambiente definiscono le modalità ed i parametri operativi con i quali U-boot effettuerà l'avvio del sistema. Le possibilità di configurazione sono elevate, quindi per semplificarne la comprensione viene mostrato un esempio di alcune variabili tipo per un boot con immagini precaricate7.3.1:

```
preload\_args=setenv bootargs console=tty00,115200n8
    ↪ root=/dev/ram0 rw ramdisk_size=196608 initrd=0
    ↪ x81000000,64M
preload\_bootcmd=run preload\_args; fatload mmc 0:1 0
    ↪ x81000000 rootfs.ext2.gz; fatload mmc 0:1 0
    ↪ x80200000 uImage; fatload mmc 0:1 0x80F80000
    ↪ am335x-boneblack.dtb; bootm  0x80200000 - 0
    ↪ x80F80000
```

Analizzandone il contenuto, si può evincere che si tratti della definizione di due variabili d'ambiente, che a loro volta definiscono due distinte azioni da eseguire.

La prima, "preload_args" imposta a sua volta un'altra variabile, nota come bootargs, usata per definire i parametri da passare al kernel. Per fare ciò utilizza il comando uboot *setenv*:

console=ttyO0,115200n8: definisce quale sia la porta seriale e i suoi parametri di velocità e parità, sul quale verrà mostrato il caricamento del kernel; fondamentale durante la fase di test e sviluppo, ininfluente durante l'esecuzione sul prodotto finale

root=/dev/ram0: definisce quale sarà il dispositivo dal quale dovrà essere montato il rootfs; in questo caso di tratta di un dispositivo virtuale definito ramdisk sul quale verrà eseguita una mappatura dell'immagine del filesystem

rw: indica che il dispositivo del rootfs verrà montato in modalità lettura/scrittura (read/write); un parametro piuttosto insolito in sistemi normali, ma necessario per questa specifica configurazione

initrd=0x81000000,64M: questo parametro indica l'indirizzo fisico dove verrà caricata l'immagine del rootfs

La seconda variabile descrive il comportamento di uno script, ovvero di una serie di comandi da eseguire, fino al boot:

run preload_args: esegue la variabile definita in precedenza, impostando fattivamente i valori di bootargs

fatload mmc 0:1 ...: queste tre chiamate caricano nell'ordine il rootfs, la uImage (kernel) e il device tree a differenti indirizzi fisici di memoria; tali indirizzi non sono casuali, e devono essere compatibili con l'hardware in uso[30]; il nome fatload indica che il caricamento avviene da una partizione fat, e l'opzione "mmc 0:1" indica che si sta utilizzando il primo dispositivo mmc (nel nostro caso la microSD) alla prima partizione

7.3 Dispositivo di memoria

La scelta del dispositivo di memoria è particolarmente significativa e può avere grande influenza sia sulla possibilità del prodotto finale sia nelle scelte dello sviluppatore. Come già affrontato nella sezione relativa alle memorie (vedere 5.1), le scelte possono essere molteplici. In base al SoC (o MCU) selezionato, si potrebbe essere limitati all'uso di una memoria flash interna (con o senza controller aggiuntivo), o in caso di dispositivi più completi, possono essere sfruttate memorie non presenti sul PC (tramite USB o SD card). In rari casi, il codice del sistema operativo può non trovarsi su un dispositivo di memoria locale, optando per un caricamento attraverso rete (ma si tratta per lo più di casi specifici o ambienti di test).

Parlando di Linux e di SoC odierni di fascia alta, si può considerare come buona generalizzazione un dispositivo che possieda una memoria dotata di controller interno (es eMMC) o di poter sfruttare un dispositivo di storage esterno (usb o sdcard). Dal punto di vista software si comportano in modo simile, offrendo il medesimo livello di astrazione, con il classico comportamento del dispositivo a blocchi. Per quanto abbastanza comune nelle evaluation board di varie fasce di prezzo, è comunque sempre possibile imbattersi in memorie NAND/NOR senza controller avanzati, ma nel resto di questo capitolo non verrà

[30] Diversamente da quanto si potrebbe credere, la mappatura della memoria non parte sempre dall'indirizzo 0x00000000.

contemplata tale possibilità, anche perché può variare molto sulla base del produttore della memoria e di come essa sia stata collocata nel progetto.

Nell'esempio seguente si suppone di utilizzare una scheda esterna; premesso che esistono svariati scenari di configurazione, per semplicità espositiva ne verranno presentati solo due abbastanza comuni: quello definito come "rootfs su partizione" e quello con "immagini pre-caricate", entrambi basati su un sistema di partizionamento classico.

7.3.1 Il partizionamento classico

Con questo termine si indica l'utilizzo di una schema di partizionamento tradizionale (MSDOS Partition Table[31]) in cui gli elementi del rootfilesystem si trovano all'interno di un classico filesystem collocato in una delle partizioni. Lo schema di partizionamento MSDOS soffre di numerosi limiti, sia per numero di partizioni che per spazio allocabile, ma questi non sono particolarmente significativi in ambito embedded, in quanto si tratta generalmente di dispositivi di memoria dalla ridotta capacità e che non necessitano di un grande numero di partizioni.

Pur esistendo da tempo uno nuovo schema di partizionamento, il GPT, (Guid Partition Table) quest'ultimo non è ancora universalmente supporto all'interno dei bootloader, rendendone di fatto impraticabile o poco probabile l'utilizzo in ambito embedded.

Rootfs su partizione

In questa modalità il filesystem contenente il rootfs viene disposto direttamente su una partizione classica. Trattandosi di un sistema GNU/Linux, il tipo di partizione prende il nome "Linux" (codice 0x83) ed il tipo di filesystem è di tipo ext2/ext4.

[31] Questo schema si basa sulla presenza nel dispositivo di storage di una struttura dati posta all'inizio dello spazio disponibile che prende il nome di Master Boot Record (MBR). Tale struttura è della dimensione di 512 byte, di cui i primi 446 riservati alla fase 1 del bootloader, 64 byte riservati alle informazioni sulle partizioni e 2 byte finali chiamati MAGIC NUMBER come campo identificativo. Questo schema consente l'uso di 4 partizioni primarie o di 3 partizioni primarie e 11 partizioni logiche.

Per motivi di compatibilità con i filesystem della famiglia ext con il bootROM o in generale con i vari bootloader, è assai probabile che **non** sia possibile fare la medesima scelta per la partizione che contiene i file del pre-bootloader, del bootloader e del kernel Linux, dovendo optare per un filesystem di tipo FAT (codice 0x0C) formattato in FAT32. Prendendo ad esempio il dispositivo utilizzato nel progetto (per comodità chiamato con il medesimo nome utilizzato dal kernel per indicare il dispositivo, ovvero "mmcblk0") un possibile schema di partizionamento potrebbe essere il seguente:

mmcblk0: dispositivo

mmcblk0p1 (64MB): tipo 0x0C, FAT32, etichetta "BOOT"; contiene i file immagine del pre-bootloader, del bootloader e del kernel, oltre al file con le variabili d'ambiente per il bootloader

mmcblk0p2 (512MB): tipo 0x83, ext2/4, etichetta rootfs; contiene i file di tutto il sistema userspace oltre che i moduli del kernel

mmcblk0p3 (1400MB): partizione opzionale per i dati da salvare o per altre informazioni; per esigenze particolari tale spazio può essere ridotto in favore del rootfs; può essere formattato sia in FAT che in ext

I pro di tale scelta sono:

- semplicità d'implementazione, si tratta del modo più intuitivo di disporre il rootfs
- semplicità di configurazione, il bootloader è già configurato per sfruttare questo schema
- possibilità di sfruttare tutto lo spazio all'interno della scheda per il rootfs; di conseguenza può aumentare la quantità di software nel sistema
- possibilità di aggiornare in modo permanente il contenuto del rootfs durante l'esecuzione del sistema

I contro di tale scelta sono:

- la copia e l'aggiornamento dei file del rootfs all'interno del dispositivo possono risultare complessi se non utilizzati con un sistema Linux (Windows e MacOS hanno problemi ad interagire con i filesystem della famiglia ext)
- i filesystem della famiglia ext sono pensati e ottimizza-

ti per i dischi rotazionali; alcune di queste ottimizzazioni possono portare ad un maggior deterioramento del dispositivo flash
- in caso di spegnimento improvviso del sistema o di interruzione di corrente si possono verificare perdite di dati o problemi in fase di riavvio del dispositivo
- possibilità di aggiornare in modo permanente il contenuto del rootfs durante l'esecuzione del sistema

Immagini pre-caricate

Questo schema, pur utilizzando il partizionamento classico come base, anche per le sopracitate esigenze del bootROM e/o del bootloader, non lo adotta per il rootfs. In pratica tutti gli elementi del sistema sono salvati come singoli file immagine compressi, compreso il rootfs. Utilizzando la medesima convenzione utilizzata nella precedente sezione è possibile rappresentare un potenziale schema di partizionamento:

mmcblk0: dispositivo

mmcblk0p1 (512MB): tipo 0x0C, FAT32, etichetta BOOT; contiene tutte le immagini di sistema in formato compresso oltre al file con le variabili di ambiente per il bootloader.

mmcblk0p2 (1400MB): partizione opzionale per i dati da salvare o per altre informazioni; per esigenze tale spazio può essere ridotto in favore del rootfs; può essere formattato sia in FAT che in ext

I pro di tale scelta sono:
- ridotto tempo di accesso I/O in fase di esecuzione
- estrema facilità di copia ed aggiornamento dell'intero sistema partendo da immagini pronte su qualsiasi sistema operativo (basandosi sull'elevata compatibilità del filesystem FAT)
- ridotto numero di operazioni sul filesystem che contiene le immagini (opzionalmente utilizzabile anche in sola lettura), limitando le possibilità di guasto e di perdita di dati in caso di problemi
- impossibilità di aggiornare in modo permanente il contenuto del rootfs durante l'esecuzione del sistema

I contro di tale scelta sono:

- le dimensioni complessive del filesystem sono fortemente condizionate dalle dimensioni della RAM
- una maggiore complessità di configurazione del bootloader
- un tempo di avvio aumentato dovuto al caricamento dell'immagine del rootfs nella RAM, ed un conseguente consumo statico della RAM dovuto alle dimensioni del rootfs
- impossibilità di aggiornare in modo permanente il contenuto del rootfs durante l'esecuzione del sistema

Si noterà come la possibilità o meno di poter modificare il contenuto del rootfs sia inteso sia come fattore "pro" che "contro": tale possibilità infatti permette di aggiornare o aggiungere software e configurazioni utili al sistema, rendendolo di fatto più flessibile, ma al contempo rende possibile anche modifiche permanenti non desiderabili all'interno del sistema, come ad esempio codice malevolo o errori all'interno dei file di configurazione. Nel caso del rootfs su partizione, una volta compromesso il filesystem rimane come unica soluzione il ripristino di rootfs stesso. Nel caso delle immagini pre-caricate potrebbe essere bastevole un riavvio di sistema, con conseguente ricaricamento dell'immagine originale.

Nel caso d'immagini pre-caricate c'è inoltre da considerare la possibilità di ridurre sensibilmente l'utilizzo di I/O sulla memoria, con effetti positivi sia sulla longevità del dispositivo che nei consumi energetici.

Minori (o quasi nulli) accessi I/O si traducono anche in una maggiore reattività del sistema. Per come è impostato al momento il sistema di init, sarebbe possibile rimuovere la sdcard una volta terminata la fase di boot e configurazione.

7.4 Esempio di configurazione della SD

Le operazioni di creazione della scheda iniziale deve essere eseguita dal computer "host" attraverso una shell con permessi di root. In questa serie di comandi si assume che il device di riferimento su sistema host sia "mmcblk0" e che il dispositivo sia vuoto:

```
# fdisk /dev/mmcblk0
(n p 1 [Enter] +512MB)
(n p 2 [Enter] +1400MB)
(t 1 0C)
(t 2 0C)
(w q)
```

La sequenza di comandi è da intendersi intervallata dalla pressione del tasto invio tra ogni singola istruzione (ogni singola lettera è un comando). Laddove presente [Enter], indica di lasciare il parametro preimpostato.

La prima sequenza richiede al software `fdisk` di creare una nuova partizione (n), primaria (p), collocata al primo slot disponibile (1), partendo dal primo settore disponibile ([Enter]) e con una dimensione di 512MB (+512MB). La seconda sequenza esegue le medesime operazioni per la seconda partizione (di dimensioni maggiori).

La terza sequenza imposta il tipo di partizione (t) per la partizione numero 1 (1) e la imposta a 0C (FAT). Analoga situazione per la sequenza successiva riferita alla seconda partizione. Infine, l'ultima sequenza scrive la tabella delle partizioni (w) ed esce dal programma (q).

Una volta creata la tabella delle partizioni è necessario formattarle:

```
# mkdosfs -F 32 /dev/mmcblk0p1
# mkdosfs -F 32 /dev/mmcblk0p2
```

Ultimata la formattazione è ora possibile copiare le immagini all'interno del dispositivo di memoria:

```
# mkdir /mnt/disk
# mount /dev/mmcblk0p1 /mnt/disk
# cd $BUILDROOT/output/images/
# cp MLO uboot.img uEnv.txt uImage rootfs.ext2.gz /
    ↪ mnt/disk
# sync
# umount /mnt/disk
```

Queste ultime istruzioni, montano la scheda SD all'interno del sistema "host" e copiano le immagini come se fossero semplici file. In seguito, dopo un'operazione di "sync" (puramente

precauzionale) verrà smontato il dispositivo di memoria al fine di essere reinserito all'interno della board per l'avvio di sistema.

Capitolo 8

FreeRTOS

Nota bene

Per la piena comprensione del seguente capitolo è necessario che il lettore disponga di una pregressa esperienza sia in termini di programmazione (in particolare con il linguaggio C), sia con in concetti di Cross-Compilazione solo parzialmente espressi in questo testo e soprattutto una grande confidenza con la teoria alla base dei sistemi operativi. Prima di affrontare questo capitolo si raccomanda la lettura di testi specifici quali ad esempio *Modern Operating Systems* di A.S. Tanenbaum[65] e *C Programming Language* di B. Kernighan e D. Ritchie[44]. Il contenuto di questo capitolo fa riferimento a quanto espresso nel testo *Mastering the FreeRTOS*([10]) di **Richard Barry** e nel *FreeRTOS Reference Manual* ([3]).

8.1 Introduzione

FreeRTOS[1] è un sistema operativo rilasciato con licenza GNU GPL modificata[2], sviluppato da Real Time Engineers Ltd, nato

[1] `http://www.freertos.org`

[2] Limiti legati al benchmarking, ma maggiore libertà di aggiunte di software proprietario.

con lo scopo di essere molto compatto, veloce e semplice ed in grado di poter essere eseguito e portato su un considerevole numero di MCU.

Come suggerisce l'acronimo RTOS (Real Time Operating System), questo sistema cerca di offrire, per quanto possibile, performance *real-time*. Parlando di FreeRTOS bisogna abbandonare l'immagine onnicomprensiva tipica dei sistemi operativi Desktop: non sono presenti né driver, né API avanzate per la gestione del sistema. È piuttosto da intendersi come una sorta di libreria scritta in C in grado di offrire al programmatore una serie di strumenti basilari quali *threading*, *task*, *mutex*, *semafori* e *timer*.

Un prodotto finale FreeRTOS combina il codice sorgente del sistema stesso con il supporto hardware del MCU in questione (con eventualmente i vari driver) ed il codice applicativo dedicato.

Esistono *porting* FreeRTOS per molte famiglie di MCU quali ad esempio:

- ST STM32;
- Atmel AVR (Arduino);
- Intel 8052;
- Renesas SuperH;
- Microchip PIC32;
- ESP8266[3].

Esistono inoltre porting per CPU e SoC più grandi quali gli x86 di Intel e i Cortex-A9 di ARM.

Della stessa famiglia di FreeRTOS, esistono anche OpenRTOS[4] (Una versione di FreeRTOS con licenza commerciale) e SafeRTOS[5] (variante FreeRTOS pensata per essere compatibile con vari standard di sicurezza).

8.2 Caratteristiche

- Modalità operativa *preemptive* o cooperativa;
- Assegnamento priorità flessibile;

[3] `https://github.com/SuperHouse/esp-open-rtos`
[4] `https://www.highintegritysystems.com/openrtos`
[5] `https://www.highintegritysystems.com/safertos`

- Meccanismo di notifica task leggero e flessibile;
- Code;
- Semafori binari e "counting";
- Mutex (anche ricorsivi);
- Software timer;
- Event group;
- Funzioni Tick hook e Idle hook;
- Controllo Stack overflow;
- Registrazione Trace;
- Gestione statistiche task in esecuzione;
- Possibilità di realizzare sistemi tick-less per dispositivi a basso consumo.

8.3 Concetti di base

Come già anticipato, FreeRTOS può essere visto come una libreria che offre strumenti multi-tasking ad applicazioni o software scritti per essere eseguiti su *bare-metal*. FreeRTOS viene rilasciato come un unico file compresso al cui interno sono presenti cartelle, sorgenti e file di compilazione.

La prima parte da identificare è il cosiddetto **port** dato che FreeRTOS può essere compilato per diverse architetture hardware e con un certo numero di compilatori. La combinazione di una architettura con un specifico compilatore rappresenta il *port*. Alcuni sorgenti presenti nel pacchetto sono generici e relativi all'intero sistema, altri invece sono specifici per singolo port.

L'alberatura delle cartelle del codice sorgente è la seguente:

- FreeRTOS
  - Source
  - Demo
- FreeRTOS-Plus
  - Source
  - Demo

All'interno della cartella Source di FreeRTOS, sono presenti i sorgenti del cosiddetto *core*[6] di sistema.

[6]In questo contesto il termine è riferito al software, e sta per *nucleo*.

tasks.c *(necessario)*: Questo file contiene il necessario per il funzionamento dei *task* all'intero di FreeRTOS;

list.c *(necessario)*: implementazione di liste, specificamente configurato per le necessità dello scheduler, ma utilizzabile anche dalle applicazioni. Semplificando molto, si può dire che sia una sorta di API per gestire oggetti *Lista* simili alle liste dinamiche del C;

queue.c *(opzionale)*: fornisce i servizi per la gestione di code e semafori. Anche se non è considerato strettamente necessario, di rado è possibile farne a meno;

timers.c *(opzionale)*: fornisce gli strumenti per utilizzare i timer;

event_groups.c *(opzionale)*: supporto *event group*. Durante l'esecuzione di un task, possono esistere più "eventi" che devono essere gestiti. Per fare ciò FreeRTOS mette a disposizione gli *event group*, insiemi di bit dove ogni singolo bit rappresenta un evento distinto all'interno del gruppo. Al variare di ogni bit dell'event group si associa una azione da eseguire. Per quanto esista qualche similitudine, gli eventi non sono da confondersi con gli *interrupt*;

croutine.c *(opzionale)*: supporto per le *co-routines*, un tipo di routine utilizzate all'interno di MCU molto piccole ed ormai in disuso.

Oltre ai file del nucleo, all'interno della cartella *Source* sono presenti due cartelle: ***include*** e ***portable***. Se il contenuto di *include* è alquanto ovvio per chi sia abituato a sviluppare con il linguaggio C, ovvero contenere gli *header file* da utilizzare per sfruttare le API di FreeRTOS, il contenuto di *portable* richiede un maggiore approfondimento.

La struttura della cartella *portable* è la seguente:

MemMang: contiene i 5 *heap allocator* disponibili in FreeRTOS (vedere nel seguito);

common: contiene mpu_wrappers.c. implementazione delle funzioni per elevare i privilegi del processore prima di chiamare una qualsiasi API di FreeRTOS;

Compiler1, Compile2, Compiler3...: per ogni compilatore supportato è presente una specifica cartella compiler, chiamata con il nome del compilatore. All'interno del-

la cartella del compilatore sono presenti singole cartelle rappresenti le architetture supportare per compilatore su FreeRTOS.

La combinazione di compilatore su architettura rappresenta il *port*, facilmente identificabile come sotto-cartella di *FreeRTOS/Source/portable*.

I compilatori presenti su *portable*:

- BCC
- CCS
- CodeWarrior
- Common
- GCC
- IAR
- Keil
- MikroC
- MPLAB
- MSVC-MingW
- oWatcom
- Paradigm
- Renesas
- Rowley
- RVDS
- SDCC
- Softune
- Tasking
- WizC

Questa struttura di cartelle, così semplice e precisa, ci fornisce una serie di *include paths*[7] per cominciare a creare un progetto basilare FreeRTOS.

- Percorso degli include del *core* (es. *FreeRTOS/Source/include*);
- Percorso del port selezionato per il progetto (es. per un ARM CortexM4 può essere utilizzato il seguente: *FreeR-*

[7] In questo contesto, la dicitura *include paths* serve ad indicare i percorsi delle cartelle dove si trovano gli *header file* utilizzati all'interno di un progetto. Tali *path* saranno necessari al compilatore per procedere con la compilazione.

TOS/Source/portable/GCC/ARM_CM4F);
- Percorso per il file *FreeRTOSConfig.h*.

Il file ***FreeRTOSConfig.h*** non è presente all'interno della cartella Source, in quanto la definizione di questo file è a carico dello sviluppatore della singola applicazione. Si tratta di un file necessario in cui devono essere rifinite le impostazioni dell'istanza di FreeRTOS attraverso una serie di parametri. Nonostante non sia disponibile nelle cartelle *Source* è possibile trovarne numerosi esempi nelle cartella *Demo*. Gli stessi sviluppatori di FreeRTOS sono inclini a suggerire a coloro che vogliono scriversi la propria applicazione di non scrivere da zero un file *FreeRTOSConfig.h*, ma piuttosto di prendere e modificare una versione di questo file da uno degli esempi presenti in *Demo*.

La cartella ***Demo*** contiene una serie applicazioni dimostrative sotto forma di codice sorgente. Idealmente gli sviluppatori di FreeRTOS hanno cercato di fornire un esempio per ogni *port* supportato.

8.4 Usare FreeRTOS

Data la vastità di applicazioni e ambienti messi a disposizione da FreeRTOS, non è possibile presentare un esempio che valga per tutti, per questo motivo si proverà a dare delle linee guida di programmazione generale.

Fermo restando che è altamente raccomandabile partire dagli esempi presenti nella cartella Demo, è comunque possibile iniziare un progetto da zero, o, come spesso avviene, aggiungere FreeRTOS ad un progetto già esistente. Partendo da un progetto già esistente (come ad esempio il BSP[8] della board di riferimento, ovviamente comprensivo di toolchain) è possibile aggiungere le funzionalità di FreeRTOS aggiungendo alle cartelle di progetto i seguenti componenti:
- Il file FreeRTOSConfig.h;
- I file core di FreeRTOS descritti in precedenza8.3;
- Gli include di FreeRTOS descritti in precedenza;

[8] *Board Support Package*

- Il *port* di riferimento[9];
- Il file di gestione memoria heap (*heap_[1-5].c*)[10];

8.4.1 Naming Convention

FreeRTOS offre un gran numero di funzioni, variabili e tipi di dato. Sarebbe impossibile, anche per lo sviluppatore esperto, memorizzarli tutti e sarebbe molto lento andare ad ispezionare la definizione per poter capire cosa faccia un determinato elemento. Per questo motivo, gli sviluppatori FreeRTOS lasciano alcuni indizi all'interno dei nomi di funzioni e variabili.

Nel caso delle variabili, usano dei prefissi prima del nome della variabile:

c per i tipi *char*

s per i *short int* (16bit)

l per i *long int* (32 bit)

u sta per *unsigned* e può essere a sua volta combinato con gli altri (es. *uc* sta per *unsigned char*)

p indica un puntatore e può essere combinato agli altri (es. *pl* sta per puntatore a *long int*)

v indica il tipo *void*

x per gli altri tipi di dato utilizzati da FreeRTOS

FreeRTOS mette a disposizione alcuni tipi di dato speciali:

TickType_t: in FreeRTOS è configurabile un particolare tipo di *interrupt* che scatta con frequenza determinata. L'intervallo (il periodo della frequenza) prende il nome di *Tick Period*. Questo tipo di dato può essere sia un intero a 16 o 32 bit (entrambi *unsigned*). Tale scelta è impostata all'interno del *FreeRTOSConfig.h* (una *flag* di nome configUSE_16_BIT_TICKS può essere impostata ad 1 per i 16 bit e a 0 per 32 bit);

BaseType_t: in base al tipo di architettura, può essere a 8, 16 o 32 bit. Utilizzato per quei tipi di dato che possono assumere pochi valori, si utilizza la dimensione più efficiente per l'architettura.

[9] *FreeRTOS/Source/portable/[compilatore]/[architettura]*

[10] presente nella cartella *FreeRTOS/Source/portable/MemMang*

Anche le funzioni in FreeRTOS assumono un nome particolare in base al file in cui sono definite e al loro valore di ritorno. Alcuni esempi:

v*Task*PrioritySet(): funzione che ritorna *void* e si trova nel file *task.c*

x*Queue*Receive(): funzione che ritorna un tipo di dato definito in FreeRTOS (nello specifico *BaseType_t*) e si trova nel file *queue.c*

pv*Timer*GetTimerID(): funzione che ritorna un puntatore a *void* e si trova nel file *timer.c*

8.5 Gestione della memoria HEAP

Sebbene a partire dalla versione 9 di FreeRTOS sia possibile gestire tutta la memoria in modo statico, è uso comune configurare il sistema in modo da poter allocare memoria in modo dinamico.

Per coloro che hanno confidenza con la programmazione C, il concetto non si discosta dall'uso in altri sistemi operativi delle chiamate *malloc()*[11] e *free()*[12]: in alcuni casi possono essere utilizzate queste *syscall* anche all'interno di FreeRTOS, anche se il loro uso è decisamente poco raccomandabile (vedere in seguito).

FreeRTOS mette a disposizione 5 diversi modi di allocare la memoria, in modo da rendere più facile adattarsi al tipo di architettura in uso e allo scopo del progetto. La scelta di queste modalità viene fatta sulla base di quale file di *heap* viene incluso nel progetto (*heap_1.c, heap_2.c, heap_3.c, heap_4.c, heap_5.c*) presenti nella cartella *MemMang*.

Per ogni configurazione di *heap* esiste una versione specifica delle funzioni ***pvPortMalloc()*** (omologo della *malloc()*) e ***vPortFree()*** (omologo della *free()*).

[11] **void *malloc(size_t size)**: alloca un valore *size* di byte e ritorna il puntatore alla memoria allocata.

[12] **void free(void *ptr)**: libera la memoria indirizzata dal puntatore *ptr*, precedentemente allocata con *malloc* o similari.

Heap1: questa implementazione fornisce solo la *pvPortMalloc()* e non la *vPortFree()*. In questo caso non è possibile parlare di una allocazione dinamica completa. La memoria viene suddivisa in blocchi più piccoli ed una volta utilizzati non vengono mai liberati. In questo contesto, un *task*, una volta creato, non viene mai più liberato, e con esso la memoria utilizzata. La dimensione dello spazio *heap* è definita all'interno del *FreeRTOSConfig.h*, con il valore di *configTOTAL_HEAP_SIZE*. Viene allocata similarmente ad un array al momento dell'avvio, incrementando il consumo iniziale di memoria;

Heap2: seppur mantenuto per retrocompatibilità, *Heap2* è stato reimplementato con maggiori funzionalità all'interno di *Heap4*, e il suo uso non è in alcun modo raccomandato. Anche in questo caso, si usa la quantità di memoria in *configTOTAL_HEAP_SIZE* suddivisa in blocchi, e a differenza della *Heap1* è implementata la funzione *vPortFree()*. L'array viene dichiarato all'inizio, aumentando il consumo di memoria iniziale. Per la gestione dell'allocazione viene utilizzato un approccio di tipo *best-fit*[13];

Heap3: utilizza la *malloc()* e la *free()* direttamente dalla *C standard library*. Tale scelta, seppur apparentemente più semplice implica conseguenze, come ad esempio l'interruzione dello scheduler ad ogni loro chiamata;

Heap4: uimilmente a *Heap1* e *Heap2* crea il suo array statico di allocazioni (con il solito consumo di memoria iniziale all'avvio), ma è gestito con un approccio *first-fit*[14] e mette in atto il meccanismo del *coalescing*[15]. in modo da minimizzare il fenomeno della frammentazione;

[13] Il termine *best-fit* indica la tendenza ad allocare nell'area di memoria libera con dimensione il più simile possibile a quella richiesta. Riduce la dimensione dei frammenti di memoria liberi.

[14] Il termine *first-fit* indica la tendenza ad allocare nella prima area disponibile di dimensione sufficiente.

[15] *Coalescing*: (in italiano la coalescenza) è l'azione di fondere in uno solo blocco due o più blocchi di memoria libera.

Heap5: l'allocazione e la gestione della memoria libera di *Heap5* è simile in tutto e per tutto a quella di *Heap4*, ma differentemente da *Heap1*, *Heap2* e *Heap4*, in questo caso non è necessario allocare una sola area di memoria contigua, ma è possibile utilizzare più aree non contigue organizzate in una *memory-map*.

Va detto che, nonostante FreeRTOS possa essere eseguito su architettura che dispongono di MMU (ved. 5.1.1), di fatto non ne supporta l'utilizzo. Ciò implica che la gestione della memoria deve essere fatta interamente dal sistema operativo, rendendo particolarmente significativa la scelta del modello heap, in particolare in termini di efficienza e protezione dei dati.

A seguire alcuni esempi di funzioni associate alla gestione della memoria:

size_t xPortGetFreeHeapSize(void): ritorna la quantità di memoria disponibile in bytes (non disponibile in *heap3*);

size_t xPortGetMinimumEverFreeHeapSize(void): ritorna un valore che rappresenta il numero minimo di byte occupati da quando l'applicazione è stata eseguita.

8.6 Task

I task sono da intendersi come dei veri e propri programmi all'interno di un programma principale, solitamente impostati per restare in esecuzione all'infinito senza uscire mai. Un *task* FreeRTOS non è destinato a ritornare nulla al programma chiamante, e può essere chiuso solo attraverso una eliminazione esplicita.

In FreeRTOS ogni task è scritto come una funzione C.

```
void EsempioTask (void *pvParametri){
        int32_t lEsempioVariabile = 0;
        for (;;){
                /* Codice operativo del task */
        }
        vTaskDelete( NULL );
}
```

Il codice deve essere implementato all'interno del ciclo infinito presente all'interno della funzione. In alcuni contesti si tende a voler evitare codice che produca loop infiniti, ma non è quanto avviene in FreeRTOS. Di fatto, senza questa condizione di loop, il task arriverebbe alla sua conclusione senza alcuna possibilità di essere ripreso in seguito (e ciò è contrario alla definizione stessa di task in FreeRTOS).

Un *task* può trovarsi in due stati fondamentali: *Running* (In esecuzione) o *Not Running* (Fermo). Semplificando il modello architetturale, si può dire che una applicazione è composta da più *task* e in un determinato istante solamente uno di questi si troverà in esecuzione, mentre gli altri saranno fermi. In realtà, *Not Running* è un insieme di sotto-stati differenti:

Ready: un task in stato *Ready* non ha condizioni bloccanti, potrebbe essere eseguito al momento, ma non è in esecuzione perché al momento il sistema è impegnato con un task a maggiore priorità;

Suspended: un task *Suspended* è stato esplicitamente fermato da un comando (es. *vTaskSuspend()*), e deve essere esplicitamente riattivato (es *xTaskResume()*);

Blocked: un task in stato *Blocked* è stato fermato ed è in attesa di uno specifico evento, come ad esempio un *delay* (*vTaskDelay()*) o è in attesa per una qualche particolare risorsa come ad esempio una coda (ved. sez. 8.7) o un semaforo (ved. sez. 8.10).

8.6.1 Creazione del task

Per creare un *task* si utilizza la API *xTaskCreate()*:

```
BaseType_t xTaskCreate(
    TaskFunction_t pvTaskCode,
    const char * const pcName,
    uint16_t usStackDepth,
    void *pvParameters,
    UBaseType_t uxPriority,
    TaskHandle_t *pxCreatedTask
    );
```

pvTaskCode: puntatore alla funzione che implementa il *task*;
pcName: nome del *task* (utilizzato in fase di debug);

usTaskDepth: dimensione dello *stack* per il *task*, che non è rappresentato in *byte* ma in multipli del valore *stack wide*. Il prodotto di tale moltiplicazione non può essere superiore al valore rappresentabile da un *uint16_t*;

pvParameter: un puntatore di tipo void che indicherà la locazione di memoria dei parametri;

uxPriority: valore di priorità del task che va da 0 a **(configMAX_PRIORITIES**[16] **- 1)**;

pxCreatedTask: puntatore ad un *handler* del task che permetterà di eseguire operazioni su quest'ultimo durante l'esecuzione (es. cambio di priorità).

Il valore di ritorno di un *task* può essere di due tipi:

pdPASS: il task è stato creato correttamente;

pdFAIL: non è stato possibile creare il task o non è stato possibile allocare memoria a sufficienza per il task.

```
void vTask1( void *pvParametri )
{
    const char *pcStringaTest = "Task␣1␣in␣esecuzione
        ↪ \n";
    volatile uint32_t ul;

    for(;;)
    {
        vPrintString(pcStringaTest);
        for(ul=0;ul<mainDELAY_LOOP_COUNT; ul++)
        {}
    }
}

void vTask2( void *pvParametri )
{
    const char *pcStringaTest = "Task␣2␣in␣esecuzione
        ↪ \n";
    volatile uint32_t ul;

    for( ;; )
    {
        vPrintString(pcStringaTest);
        for(ul=0;ul< mainDELAY_LOOP_COUNT;ul++)
        {}
    }
}
```

[16]Questo valore è espresso nel file FreeRTOSConfig.h

```
int main(void)
{
    xTaskCreate(vTask1, "Task␣1", 1000, NULL, 1, NULL
        ↪ );
    xTaskCreate(vTask2, "Task␣2", 1000, NULL, 1, NULL
        ↪ );
    vTaskStartScheduler();
    for( ;; );
}
```

Nel esempio precedente è possibile vedere la dichiarazione di due funzioni task conformi al prototipo (ritorno *void* che accettano in ingresso un puntatore a *void* per i parametri).

In questo caso è possibile vedere come il task FreeRTOS sia creato con la chiamata *vTaskCreate*. Come si può vedere, entrambi i Task sono creati con la medesima dimensione di stack (a seconda delle configurazioni quel 1000 può corrispondere a 2 o 4 Kb) con il medesimo valore di priorità (in questo caso **1**) e senza specificare alcun puntatore a parametri o puntatore ad *handler*. Tra le due chiamate cambia il nome simbolico che diamo al task, ed ovviamente il puntatore a funzione (che coincide con il nome della funzione stessa).

Da notare la chiamata a ***vTaskStartScheduler()***: questa chiamata avvia lo scheduler che farà girare, presumibilmente all'infinito, i task creati. Se dovesse uscire dal ciclo principale, ciò potrà essere dovuto a qualche problema relativo all'esecuzione dei task o all'allocazione della memoria necessaria.

8.7 Queue

Le *Queue* (in italiano più semplicemente "code"), sono un meccanismo di comunicazione tra processi (o tra processi ed interrupt). Per semplicità, si può pensare ad una coda come un insieme di dati di vario genere il cui numero e dimensione sono prefissati.

Lo scopo di una coda però non è la conservazione dei dati, ma piuttosto la gestione di un flusso di dati in ingresso ed uscita dalla coda stessa.

Una caratteristica tipica delle "code" in ambito informatico è il modo in cui vengono gestiti i dati al suo interno: generalmente si può parlare della cosiddetta modalità ***FIFO***[17]

In una coda si possono identificare due nodi detti "sentinella" chiamati *head* (testa) o *front* e *tail* (coda[18]) o *back*, rispettivamente inizio e fine della coda. In caso di "coda circolare", i due nodi sentinella coincidono.

Vi sono due tipi di implementazioni delle code:

queue by copy (lett. "coda per copia"): i dati vengono copiati all'interno della coda;

queue by reference (lett. "coda per riferimento"): all'interno della coda ci sono solo i puntatori ai dati.

In FreeRTOS, vengono utilizzate le *queue by copy*.

Un'altra caratteristica delle code è l'accesso simultaneo: task differenti (o ISR[19]) possono accedere alle stessa coda.

L'accesso, sia esso in lettura o in scrittura ad una coda può essere definito "bloccante", ovvero il task si ferma fino al momento in cui la coda non consente l'accesso. Tale periodo di attesa bloccante può essere definito a priori.

Ogni coda deve essere esplicitamente creata prima del suo utilizzo. Per la creazione di una coda in FreeRTOS si utilizza la chiamata ***xQueueCreate***.

```
QueueHandle_t xQueueCreate(
    UBaseType_t uxDimCoda,
        UBaseType_t uxDimElementi)
```

uxDimCoda: massimo numero di elementi presenti nella coda;

uxDimElementi: dimensione in byte dei singoli elementi della coda.

Una coda può essere gestita con un particolare tipo di *handler* corrispondente al tipo *QueueHandle_t*. Per semplicità, lo si può considerare un puntatore alla coda. La funzione *xQueueCreate* può ritornare un *handler* per la coda o NULL in caso di fallimento in fase di creazione.

[17] Acronimo inglese che sta per *First In First Out*. Letteralmente il primo che entra nella coda è anche il primo che esce.

[18] La traduzione in italiano può essere fuorviante, in quanto sia *queue* che *tail* possono essere tradotti con la parola "coda".

[19] Interrupt Service Routine

Una volta creata la coda, è possibile inserire nuovi elementi in testa o in coda.

```
BaseType_t xQueueSendToFront(
    QueueHandle_t xCodaEsempio,
        const void * pvElemDaAccodare,
        TickType_t xTicksdiAttesa);

BaseType_t xQueueSendToBack(
    QueueHandle_t xCodaEsempio,
        const void * pvElemDaAccodare,
        TickType_t xTicksdiAttesa);
```

La funzione ***xQueueSendToFront()*** inserisce in testa alla coda *xCodaEsempio* l'elemento *pvElemDaAccodare* attendendo per un determinato numero di *Ticks* pari al valore di *xTicksdiAttesa*. Analoghe considerazioni per l'inserimento in coda possono essere fatte per la funzione ***XQueueSendToBack()***. Il tempo di attesa è da intendersi come il tempo per cui è possibile aspettare la disponibilità della coda prima di far fallire la chiamata. Una coda può non essere disponibile all'istante, ma può esserlo poco dopo, consentendo quindi l'eventuale svolgimento del task che ne ha fatto richiesta. Entrambe le funzioni possono ritornare i seguenti valori:

pdPASS: inserimento avvenuto con successo;
errQUEUE_FULL: inserimento fallito (coda piena).

L'estrazione (o lettura) di elemento dalla coda può essere effettuata con la funzione *xQueueReceive()*.

```
BaseType_t xQueueReceive(
        QueueHandle_t xCodaEsempio,
        void * const pvBuffer,
        TickType_t xTicksdiAttesa);
```

La funzione ***xQueueReceive()*** legge dalla coda *xCodaEsempio* con un tempo di attesa pari a *xTicksdiAttesa* salvando l'elemento estratto (e di conseguenza eliminato dalla coda) all'indirizzo di memoria espresso dal puntatore *pvBuffer*. La funzione può ritornare i seguenti valori:

pdPASS: estrazione avvenuta con successo;
errQUEUE_EMPTY: impossibilità di effettuare la lettura (coda vuota).

Va detto che le chiamate in lettura e scrittura su una coda sono di tipo bloccante, spostando il task in uno stato *Not Running*. Quando l'operazione richiesta sulla coda diventa disponibile, viene scatenato un *interrupt* che riporta il task in stato *Running*. In questo senso diventa fondamentale impostare un tempo massimo in *Ticks* di attesa, in modo da non bloccare i task all'infinito, ma al tempo stesso di tollerare l'indisponibilità temporanea di una determinata operazione.

In FreeRTOS le code possono ricevere dati da sorgenti differenti e contestualmente è possibile fare operazioni di lettura da code differenti.

8.8 Timer

In FreeRTOS è possibile utilizzare i *Timer Software*: non sono legati a particolari risorse hardware (come gli *Hardware Timer*) e sono utilizzati per eseguire delle specifiche azioni ad un tempo determinato. Un *Software Timer* (da qui in avanti verranno chiamati solo *timer*) possono essere legati a delle speciali funzioni *callback* eseguite nel momento in cui il *timer* scatta.

```
void TimerCallback(TimerHandle_t TimerEsempio);
```

Una *Timer Callback* deve necessariamente ritornare *void* e non può mai entrare in stato *blocked*. Questo limite impone che all'interno di queste *callback* non sia possibile chiamare funzioni bloccanti appartenenti alle API di FreeRTOS (come ad esempio le funzioni *xQueueReceive* viste in precedenza). In generale è buona norma evitare tutto quello che impone dei tempi di attesa (es. *vTaskDelay()*). Si definisce **periodo** di un *timer* il tempo che deve intercorrere tra l'avvio del timer stesso e l'avvio della funzione *callback*.

I timer possono essere di tipo ***One Shot*** o ***Auto-Reload***: i primi eseguono la *callback* una sola volta, mentre i secondi ripartono dopo ogni esecuzione della *callback*, trasformandosi in ciclo di esecuzione temporizzato.

Un *timer* può trovarsi in due stati:

Dormiente (*Dormant*): il *timer* esiste e può essere utilizzato attraverso il suo *handler* ma la *callback* non verrà chiamata;

Esecuzione (*Running*): il *timer* è in esecuzione e la callback verrà chiamata dopo il periodo definito.

Un *timer* può essere creato con la funzione API ***xTimerCreate()***.

```
TimerHandle_t xTimerCreate(
        const char* const pcNomeTimer,
        TickType_t xPeriodoinTick,
        UBaseType_t uxAutoReload,
        void * pvTimerID,
        TimerCallbackFunction_t pxCallBack);
```

Parametri della funzione *xTimerCreate()*:

pcNomeTimer: un nome, ignorato da FreeRTOS, ma utile per sviluppo e debug, del *timer*;

xPeriodoinTick: il periodo specificato in *tick*;

uxAutoreload: può avere valore **pdTRUE** se il timer è di tipo *Auto-Reload* o **pdFALSE** in caso contrario (*One-Shot*);

pvTimerID: questo ID è un puntatore a tipo *void* utile per identificare il singolo timer;

pxCallBack: un puntatore a funzione che consente di specificare la funzione da eseguire una volta scattato il *timer*.

La funzione API *xTimerCreate* in caso di successo ritorna un puntatore ad un *handler* del *timer*, o *NULL* in caso contrario.

Una volta creato, il *timer* si trova in stato dormiente. Per farlo passare in stato di esecuzione è necessario chiamare la funzione *xTimerStart()*.

```
BaseType_t xTimerStart(
        TimerHandle_t xTimer,
        TickType_t xTicksToWait );
```

Come è intuibile, *xTimer* è un *handler* di un *timer*, mentre *xTicksToWait* è il numero massimo di Tick da aspettare per lo start del *timer*. In base al successo o meno della chiamata, *xTimerStart()* può ritornare rispettivamente il valore **pdPASS** o *pdFALSE*.

Per far tornare un *timer* in stato dormiente è possibile utilizzare la funzione API *xTimerStop()*. Va detto che normalmente

non è raccomandabile far partire un *timer* da una *Interrupt Service Routine*. La versione *interrupt-safe* di questa funzione è la *xTimerStartFromISR()*.

E' inoltre possibile, indipendentemente dal fatto che il *timer* si trovi in esecuzione o sia dormiente, resettare (o azzerare) il *timer* stesso con la chiamata *xTimerReset()*.

```
BaseType_t xTimerReset(
        TimerHandle_t xTimer,
        TickType_t xTicksToWait );
```

I parametri sono omologhi alla *xTimerStart()*.

8.9 Interrupt

Come ogni sistema operativo, è importante per FreeRTOS interagire con gli *interrupt*. Un *hardware interrupt* (o interruzione) è il modo con cui una periferica di qualsiasi tipo richiede al processore di eseguire delle azioni specifiche, fermandosi (da cui "interruzione"), salvando (tipicamente in *stack*) quello che stava facendo per poi riprenderlo una volta evasa l'azione richiesta. Come già accennato in precedenza, alla ricezione di un *interrupt* vengono chiamate delle speciali funzioni note come *Interrupt Service Routines* o ***ISR***.

In FreeRTOS sia i *task* che le *ISR* hanno dei valori di priorità, ma non sono paragonabili tra loro; per definizione le *ISR*, anche quelle con priorità più bassa, hanno sempre più priorità rispetto a qualsiasi *task*, anche se a priorità alta.

In questa logica, le API FreeRTOS sono sdoppiate, disponendo di versioni per i normali *task* o per le *ISR*[20]). Le chiamate API per *ISR* si distinguono dalle loro omologhe per i *task* dal suffisso ***FromISR*** (es *xTimerStart()* per i *task* e *xTimerStartFromISR()* per le ISR).

[20] *Interrupt Safe API*

8.10 Semafori

In informatica vengono definiti **semafori** delle particolari strutture dati utilizzate per la sincronizzazione di risorse condivise da più *task*[65]. Si può immaginare un semaforo come un struttura dati i cui possibili valori sono limitati (talvolta solo **0** e **1**, in tal casi si definiscono semafori binari) su cui è possibile eseguire un numero limitato di operazioni:
Init: inizializzazione del semaforo;
Wait: decremento del semaforo;
Signal: incremento del semaforo.

Per semplicità, si possono considerare queste operazioni come atomiche, eseguite in un unico flusso di esecuzione senza interruzioni.

In FreeRTOS si usa la dicitura "prendere il semaforo" (o *take*) e "dare il semaforo" (o *give*) rispettivamente per la *wait* e per la *signal*.

Un esempio di creazione del semaforo:

```
SemaphoreHandle_t xSemaphoreCreateBinary( void );
```

La funzione *xSemaphoreCreateBinary* ritorna un handler del semaforo (o *NULL* in caso di fallimento).

Come anticipato in precedenza, se disponibile, un semaforo può essere preso (o ricevuto) e ciò viene fatto con la chiamata *xSemaphoreTake()*:

```
BaseType_t xSemaphoreTake(
    SemaphoreHandle_t xSemaforo,
    TickType_t xTicksDaAttendere );
```

La chiamata accetta in ingresso:
xSemaforo: un handler di semaforo;
xTicksDaAttendere: il numero massimo di tick in cui bloccarsi in attesa di ricevere il semaforo.

Come di consueto, se sarà possibile ottenere il semaforo entro il numero massimo di *tick* specificati la chiamata ritornerà il valore ***pdPASS***, se invece fallirà ritornerà ***pdFALSE***.

Per far si che un semaforo sia disponibile è necessario che una volta preso tale semaforo venga rilasciato (o *give*).

```
BaseType_t xSemaphoreGive(
    SemaphoreHandle_t xSemaforo);
```

Il parametro *xSemaforo* è un *handler* di semaforo. Come già visto in precedenza, esistono versioni specifiche di queste funzioni da utilizzare all'interno delle *ISR*: ***xSemaphoreTakeFromISR()*** e ***xSemaphoreGiveFromISR()***.

8.11 Esempio: LED Blink

A seguire, un esempio semplificato con due *task* per fare accendere e spegnere due LED con tempistiche differenti su una board STM32.

```
1  #include "FreeRTOS.h"
2  #include "list.h"
3  #include "task.h"
4  #include "queue.h"
5
6  /* Headers of Hardware Library */
7
8  #include "stm32f3_discovery.h"
9  #include "stm32f30x.h"
10
11 static uint64_t u64Ticks;
12
13 void vMyTaskLed1(void *pvParameters)
14 {
15         for(;;)
16         {
17                 STM_EVAL_LEDOn(LED8);
18                 vTaskDelay(500);
19                 STM_EVAL_LEDOff(LED8);
20                 vTaskDelay(500);
21         }
22 }
23
24 void vMyTaskLed2(void *pvParameters)
25 {
26         for(;;)
27         {
28                 STM_EVAL_LEDOn(LED3);
29                 vTaskDelay(1000);
30                 STM_EVAL_LEDOff(LED3);
31                 vTaskDelay(1000);
```

```
32          }
33
34  }
35
36  void myInit()
37  {
38          STM_EVAL_LEDInit(LED8);Dal punto di vista
                ↪ pratico
39          STM_EVAL_LEDInit(LED5);
40          STM_EVAL_LEDInit(LED3);
41  }
42
43  void vApplicationIdleHook( void )
44  {
45          STM_EVAL_LEDOn(LED5);
46  }
47
48  int main()
49  {
50          xTaskHandle  rtosTaskHandleArray[2];
51
52          myInit();
53          xTaskCreate(vMyTaskLed1,
54                  (const char *) "LED1",
55                  configMINIMAL_STACK_SIZE,
56                  NULL, 3, &rtosTaskHandleArray[0]);
57          xTaskCreate(vMyTaskLed2,
58                  (const char *) "LED2",
59                  configMINIMAL_STACK_SIZE,
60                  NULL, 3, &rtosTaskHandleArray[1]);
61          vTaskStartScheduler();
62
63          return 0;
64  }
65  void vApplicationTickHook( void ) {
66          ++u64Ticks;
67  }
68  void vApplicationMallocFailedHook( void ) {
69          configASSERT( 0 );
70  }
```

Alcune considerazioni sul presente listato:

- L 8,9: Sono stati aggiunti gli header della board in utilizzo
- L 11: Dichiarazione del Tick counter
- L 13,24: Definizione delle funzioni che diventeranno task
- L 18,20,29,31: ***vTaskDelay()*** ferma il *task* per un numero scelto di millisecondi (simile alle msleep dei sistemi Linux)

- L 17,19,29,31: Funzioni BSP della board per accendere e spegnere i led.
- L 43: Se definito ***configUSE_IDLE_HOOK*** la funzione ***vApplicationIdleHook*** viene chiamata in ogni fase di *idle*[21]
- L 48: Una applicazione FreeRTOS necessita di una funzione *main*
- L 50: A posto di usare un singolo handler per ogni *task* ma si usa un *array* di handler.
- L 53,59: Creazione dei task con la chiamata ***xTaskCreate***
- L 61: La chiamata ***vTaskStartScheduler*** avvia lo scheduler e tutti i task creati.
- L69: ***configASSERT*** è l'omologo della *assert()* macro utilizzata nel linguaggio C, non sempre utilizzabile in FreeRTOS. Se il valore passato è uguale a 0, genera un errore ed interrompe l'esecuzione del programma. In questo contesto, è utilizzata come debug per il fallimento di allocazioni di memoria.

[21] In questo contesto, quando il processore non sta eseguendo nessun task.

Capitolo 9

Arduino e Wiring

La storia[1] di Arduino comincia nel 2003 all'Interaction Design Institute di Ivrea quando lo studente Hernando Barragán inizia a lavorare ad una tesi sotto la supervisione di Massimo Banzi e di Casey Reas (co-autore del linguaggio Processing `http://processing.org/`).

Lo scopo del lavoro era la realizzazione di un ambiente di sviluppo software e hardware economico, semplice e completo per permettere agli studenti (ma anche a progettisti professionisti) di *interaction design* la sperimentazione in progetti integrati interattivi. Il contesto di allora era favorevole perché il mercato non offriva soluzioni a basso costo e a basso *gap* cognitivo di ingresso. Le piattaforme embedded dei primi anni 2000 costavano alcune centinaia di dollari (per le versioni "developer", poi le *board* "production" costavano meno, ma in quantitativi non proponibili per produzioni a tiratura limitata), gli ambienti di sviluppo erano spesso proprietari, i linguaggi usati erano "complessi" (C, C++, assembler!) e la documentazione non era sempre alla portata di tutti.

Barragán propose la creazione di un ambiente completo (hardware+software), che chiamò `Wiring`, composto da un linguaggio derivato da Processing, un ambiente di sviluppo (Integra-

[1] Riferimenti: `http://www.arduino.cc` e `http://arduinohistory.github.io` (una voce parzialmente inascoltata).

ted Development Environment) realizzato in Java (quindi multipiattaforma) e una *board*, inizialmente basata sul *Javelin Stamp microcontroller*[54] di Parallax poi rapidamente sostituito dai componenti Atmel, molto semplice, su cui far girare i programmi, chiamati "*sketch*". *Board* che una volta programmata poteva essere svincolata dal computer per vivere e funzionare indipendentemente dal PC di sviluppo, potenzialmente connessa a dispositivi per l'interfaccia col mondo fisico (sensori, attuatori).

Da quell'embrione nacque, dopo essere passato attraverso alcune fasi di industrializzazione (molte delle quali sono in corso tuttora), il "filone" Arduino. E' riduttivo definire Arduino una semplice "board", oggi infatti al nome Arduino si associa:

- una famiglia di board
- alcuni ambienti di sviluppo
- vari linguaggi di programmazione
- un sistema operativo completo se parliamo di ciò che gira su alcune board (OpenWrt sulla YUN)
- uno standard de facto per l'I/O
- una lunga lista di produttori e venditori di hardware compatibile
- ... oltre ad un *brand* su cui sono state combattute battaglie legali[2]

La maggior parte del software e dell'hardware che gravita nell'ecosistema Arduino è "open"[3] ergo è consentita, quando non addirittura incoraggiata (!), la clonazione, la riproduzione, il miglioramento, la redistribuzione della conoscenza.

Assieme ad alcune caratteristiche interessanti, qui sotto descritte, Arduino è diventato praticamente lo standard di riferimento e confronto per ogni altra piattaforma embedded presente o futura, almeno per il target para-industriale (mondo hobbistico, makers, etc.).

[2] `http://hackaday.com/2016/10/01/arduino-vs-arduino-arduino-won/`.

[3] un misto di Creative Commons (`http://creativecommons.org`), GPL (`http://www.gnu.org/licenses/gpl.html`) e altre licenze libere.

9.1 I punti di forza

- Le già citate licenze, che lo rendono un ambiente a bassissimo *gap* economico di ingresso (oggi si trovano sul mercato schede Arduino compatibili a pochi euro) e ad altissimo livello di *community* (persino i produttori di hardware sono ben contenti di interagire con o addirittura di creare delle *community* di appassionati che lavorano su queste piattaforme perché non temono di perdere proprietà intellettuale dato che è tutto "open by design")
- Una "architettura" KISS (Keep It Simple, Stupid!), sia dal punto di vista hardware (processore 8 bit, almeno il primo della serie, poca rom/ram quindi un solo programma per volta, no MMU, no sistema operativo) che software (linguaggio semplificato ma familiare e processo di sviluppo "mascherato" dall'interfaccia utente).
- Il linguaggio di programmazione ha una sintassi molto simile a C e Java, cioè i costrutti di base (controllo di flusso, dichiarazioni di variabili e funzioni, etc.) sono pressoché identici, mentre vengono mascherati (a meno che non si voglia esplicitamente usarli) i costrutti più complessi (oggetti, puntatori, etc.). Nell'utilizzo standard è un linguaggio imperativo strutturato.
- Nel corso del tempo i produttori e le *community* hanno sviluppato una pletora di librerie per interfacciarsi a sensori/attuatori/etc. di ogni genere e colore, per cui oggi è raro dover ricorrere a qualcosa di più complicato di un "*download library*" dall'interfaccia utente dell IDE (Integrated Development Environment) per far interagire una qualunque periferica con un Arduino.
- Fin dall'inizio, dato che l'oggetto doveva interfacciarsi col mondo fisico, la board originale (e le successive che sono state introdotte fino ad oggi) offriva una serie di "pin" di I/O (Input/Output), sia digitali (anche in PWM, cfr. sezione 3.4) che analogici, disposti secondo una collocazione che col tempo è diventata uno standard de facto (figura 9.1). Paragoniamo questa influenza a quella che scaturì negli anni '80 del secolo scorso quando IBM mise sul

mercato il suo modello PC XT e il successivo PC AT, nel design decisero (e furono i primi ad avere largo successo) di implementare un BUS di interfaccia di cui pubblicarono le specifiche [35] in modo che altri produttori di hardware potessero creare schede "compatibili" e far nascere così un mercato che poi effettivamente semi-monopolizzò il mondo informatico. E così fu anche nel caso di Arduino, nel corso del tempo molti produttori decisero di adattarsi e produrre hardware "Arduino compatibile" fin nella piedinatura. Negli ultimissimi anni c'è stata una lieve controtendenza perché sono state sviluppate board "Arduino compatibili" dal punto di vista software ma con *form factor* molto ridotti[4], abbandonando in questi casi la compatibilità del layout.

9.2 I punti di debolezza

Naturalmente non è tutto "rose e fiori", anche l'ecosistema Arduino ha le sue pecche... In particolare alcuni suoi punti di forza ne causano qualche debolezza.

- La legalità e l'enorme facilità di riproduzione dell'hardware (nella maggior parte dei casi vengono addirittura forniti con licenza libera i file per la produzione dei PCB - Printed Circuit Board) e del software ha creato un'amplissima disponibilità di produttori che vendono a prezzi molto bassi (si arriva ad ordini di grandezza di un decimo dei prezzi applicati ai prodotti "ufficiali"), purtroppo non sempre con alti livelli qualitativi. Non è raro trovare[5]: saldature "fredde" (che non fanno contatto elettrico) o in corto con altre piste del circuito stampato, componenti piegati fino a spezzarsi (a causa del cattivo imballo di spedizione), tempi di spedizione biblici (soprattutto dalla Cina), spese doganali non prevedibili, materiali costruttivi scadenti (specie i cavi e gli adattatori), standard elettrici non sempre rispettati alla perfezione (es. 5 Volt nominali

[4] Ad esempio: Wemos (`http://wemos.cc`) e NodeMCU (`http://nodemcu.com`).

[5] Sono tutti esempi capitati personalmente all'autore.

dichiarati da un alimentatore USB risultano 4.6 alla misura; una coppia di motori elettrici dichiarati uguali ma con evidenti differenze di resa elettrica).

- La proliferazione degli appassionati e il grande fermento delle comunità di utenti e sviluppatori rende molto florida la disponibilità di librerie software per utilizzare ogni tipo di hardware, certo... ma non è infrequente trovare **troppe** alternative sia a causa del sano *forking* tipico del mondo del Software Libero, sia a causa della malsana sindrome del *not invented here* che porta alcuni produttori a realizzare per forza in proprio librerie per l'hardware che costruiscono senza rispettare standard esistenti[6].

9.3 Software: struttura di uno *sketch*

Lo schema base di uno sketch Arduino è molto semplice, nella forma minimale consta di due sole funzioni da implementare: `setup()` e `loop()`. Il concetto da richiamare è il `main()` del linguaggio C, una funzione che viene invocata automaticamente al "lancio" (esecuzione) del programma compilato.

Anche `setup()` e `loop()` funzionano in modo analogo, salvo che:

```
void setup(){
  // codice che viene eseguito all'accensione
  // della board (una volta sola)
}
```

e

```
void loop(){
  // codice che viene eseguito
  // ``in loop'', quindi in continuazione
  // (ripetitivamente)
}
```

[6] Ad esempio alcuni tipi di strisce LED RGB hanno avuto una storia sofferta dal punto di vista della "comandabilità" via Arduino a causa della scelta di protocolli di comunicazione non sempre perfettamente noti e quindi non sempre supportati dalle librerie ufficiali.

Nel `setup()` vanno messe tutte le istruzioni di configurazione iniziale (ad es. l'apertura della seriale, la richiesta di indirizzo IP, etc.) da fare una volta per tutte. Nel `loop()` invece tutte le istruzioni vere e proprie del programma che si vuol produrre. Nel codice mostrato in figura "esempio di compilazione", guardiamo proprio il `loop()`:

```
void loop() {
  digitalWrite(LED_BUILTIN, HIGH);
  delay(1000);
  digitalWrite(LED_BUILTIN, LOW);
  delay(1000);
}
```

e ragioniamo sull'effetto; il LED lampeggia in continuazione pur mancando un qualsiasi costrutto "ripetitivo" (un for/while) perché la funzione `loop()` viene, appunto, chiamata in loop!

Nell'installazione dell'IDE è compresa (voce "Help" della barra dei menu) anche una copia completa della documentazione disponibile sul sito ufficiale: `http://www.arduino.cc/en/Guide/HomePage`, quindi anche essendo *offline* c'è comunque un'ancora di salvezza. Sul sito, oltre alla documentazione ufficiale stabilizzata/matura, v'è un forum molto popolato. Arduino è una piattaforma ormai molto popolare, sono nate negli anni fior di community e di sviluppatori singoli che hanno prodotto librerie (spesso integrate poi nell'IDE ufficiale), sketch completi, hardware, etc.

9.4 Il linguaggio

Wiring è un termine frequentemente utilizzato in maniera erronea per indicare il linguaggio di programmazione di Arduino, in realtà indica l'insieme di:

- ambiente di sviluppo
- alcune librerie C/C++
- piattaforma hardware (l'Arduino vero e proprio) su cui girano i programmi sviluppati

Il linguaggio di programmazione delle *board* Arduino si chiama tautologicamente "Arduino programming language". Ai fini di

questa appendice possiamo comunque utilizzare il termine Arduino per riferirci al linguaggio di programmazione[7] (oltre che alle varie *board*).

La sintassi della programmazione in Arduino è molto simile al linguaggio C o a Java, ma i costrutti più complessi come i puntatori o la programmazione a oggetti, che sono comunque disponibili, vengono messi in secondo piano con grande sollievo degli utenti "base" (e non solo!).

E' un linguaggio tipizzato, nel senso che definisce e supporta la definizione di variabili tipizzate e il compilatore "gestisce"[8] la concordanza dei tipi nelle assegnazioni tra variabili.

Fornisce tutti i costrutti standard di un linguaggio imperativo quindi: variabili ed espressioni, controllo di flusso, definizione di funzioni. In più, dato il contesto applicativo dei sistemi di controllo (sensoristica e attuatoristica), offre funzionalità legate alla gestione dei segnali: I/O digitale e analogico, interrupt su eventi hardware e accesso al clock (inteso proprio come tempo trascorso dall'accensione).

Nota bene: il linguaggio è case sensitive, cioè la sintassi distingue il minuscolo dal maiuscolo, ad esempio `float variabile` è completamente diverso da **Float VARIABILE** e non è detto che entrambe siano sintatticamente corrette.

9.4.1 Tipi di dato e variabili

Ogni tipo qui elencato corrisponde, quando associato ad una variabile dichiarata, ad una zona di memoria allocata e ad un range di valori rappresentabili.

boolean vale solo *true* o *false*, occupa un byte in memoria

char un singolo carattere, occupa un byte, range da -128 a +127 (i valori negativi non necessariamente hanno una rappresentazione ASCII)

unsigned char la versione di **char** senza segno, range da 0 a 255 (il reference consiglia di usare **byte**)

[7] Il termine Wiring per indicare il linguaggio è comunque accettato dagli autori.

[8] Eventualmente operando una conversione, potenzialmente perdendo informazione (troncamenti).

byte numero a 8 bit, da 0 a 255, occupa un byte (coerentemente!)

int intero con segno, a 16 bit sulle piattaforme ATMega e a 32 bit sulle SAMD (es. la Zero), occupa due o quattro byte

unsigned int la versione di **int** senza segno, range da 0 a $2^{16} - 1$ (=65535) o $2^{32} - 1$

word analogo a **int**

long intero con segno a 32 bit, range da -2^{16} a $2^{16} - 1$, occupa 4 byte

unsigned long la versione di **long** senza segno, range da 0 a $2^{32} - 1$

short intero con segno a 16 bit, occupa due byte

float numeri decimali in virgola mobile, range da $(-3.4028235E+38)$ fino a $(3.4028235E + 38)$, occupa 4 byte

double su quasi tutte le piattaforme sinonimo di **float**, su alcune (es. Due) rappresenta numeri in virgola mobile in doppia precisione occupando 8 byte

array vettori di variabili omogenee, vi si accede per indice posizionale

"string" (in realtà char array) non esiste come tipo base, per gestire stringhe di testo di dimensione fissa basta creare *array* (vedi più avanti) di caratteri

String (object) esiste la classe stringa, ma è sconsigliabile abusarne perché comporta allocazione non sempre prevedibile della memoria

Una variabile viene allocata **dichiarandola** (obbligatoriamente) prima di poterla usare:

```
int variabileIntera;
```

da questo momento in poi la variabile (in questo caso di tipo *int*) è disponibile per essere **assegnata**, così:

```
variabileIntera=42;
```

volendo compattare il codice, dichiarazione e assegnamento possono essere combinati:

```
int variabileIntera=42;
```

i valori assegnati alle variabili devono essere compatibili col tipo dichiarato per la variabile stessa, ad esempio la seguente linea di codice è "errata":

```
int variabileIntera=42.42;
```

perché il valore che si tenta di assegnare alla variabile intera è decimale (*floating point*). Notare che la compilazione va comunque a buon fine ma la variabile conterrà un valore troncato e non quello inteso dal programmatore.

Le variabili hanno una "visibilità" (in inglese *scope*) all'interno del codice ben definita: ciò che viene dichiarato dentro uno *statement*[9] non è visibile, quindi non è utilizzabile, al di fuori. Ad esempio il codice qui sotto:

```
funzione() {
   int variabileIntera=42;
}
...
altraFunzione() {
   variabileIntera=42+42;
}
```

non viene compilato e scatena un errore "*variable not declared in this scope*". E' possibile dichiarare delle variabili cosiddette "globali" posizionandone la dichiarazione al di fuori di tutti gli *statement*, tipicamente all'inizio del programma (chiamato *sketch* nella terminologia Wiring/Arduino). Una variabile non può essere dichiarata più di una volta nello stesso scope.

9.4.2 Espressioni e operatori

L'assegnamento delle variabili non avviene solo mediante "valore *hardcoded*" (es. un numero). Nella maggior parte dei casi la "parte destra dell'assegnamento" (ciò che viene dopo l'uguale) è rappresentato da una cosiddetta "**espressione**": un insieme di operandi e operatori che, una volta "valutata", esprime (appunto) il risultato di un calcolo. Ad esempio:

```
int variabileIntera=(42+9)*2;
```

[9] Porzione di codice racchiusa tra due parentesi graffe.

la variabile assumerà il valore 102 a valle dell'assegnamento.

Se non si vogliono ricordare le precedenze tra operatori e/o si vuol forzare l'ordine di valutazione delle sottoespressioni l'autore consiglia di utilizzare le parentesi. Nell'esempio citato, senza parentesi, il risultato sarebbe stato 42+18=60.

Gli **operatori** si dividono in categorie:

Aritmetici "=" (assegnamento), "+" (somma), "-" (sottrazione), "*" (prodotto), "/" (divisione), "%" (modulo)

Confronto "==" (uguale a), "!=" (diverso da), "<" (minore di), ">" (maggiore di), "<=" (minore o uguale a), ">=" (maggiore o uguale a)

Booleani "&&" (and), "||" (OR logico), "!" (NOT logico)

Puntatori [omessi per salute mentale dei lettori]

Bitwise (manipolazione dei bit) "&" (AND bit a bit), "|" (OR bit a bit), "$\hat{}$" (XOR bit a bit), "$\sim$" (NOT bit a bit), "«" (spostamento dei bit verso sinistra), "»" (spostamento dei bit verso destra)

Composti "++" (incremento), "–" (decremento), "+=" (somma), "-=" (sottrazione), "*=" (prodotto), "/=" (divisione), "%=" (modulo), "&=" (and bit a bit), "|=" (or bit a bit)

è possibile condensare un'espressione e un assegnamento usando questi operatori risparmiando un po' di "fatica di battitura". Ad esempio: `variabile+=42;` equivale a: `variabile=variabile+42;`

9.4.3 Array

Un *array* è un vettore di elementi, cioè di "variabili", omogenei (tutti hanno lo stesso tipo) il cui accesso avviene per "indice" (primo elemento, secondo, etc.). Nelle trattazioni usuali viene rappresentato graficamente come un "casellario":

```
[v1]-[v2]-[v3]-[v4]-[v5]-[v6]...
 0     1     2     3     4     5
```

Ogni casella contiene un valore (indicato con "vN") utilizzabile come una qualunque variabile, la riga sotto indica le posizioni

nel casellario, gli indici partono da 0[10] in quasi tutti i linguaggi di programmazione che supportano gli *array*.

La sintassi per la creazione di un *array* è:

```
int arrayDiInteri[6];
```

dove *int* indica il tipo di tutto l'*array*, cioè di ogni elemento (quindi in questo caso abbiamo creato un "casellario" di numeri interi), il numero tra parentesi quadre invece indica la dimensione del casellario, la sua lunghezza, la sua "occupazione" in memoria. Un *array* di 6 elementi fornisce caselle alle posizioni 0, 1, 2, 3, 4 e 5. Per "riempire" (tra virgolette perché in realtà la memoria è già stata allocata) un elemento di un *array* si usa un normale assegnamento di variabile, lievemente adattato:

```
arrayDiInteri[3]=34;
```

In questo caso stiamo assegnando alla QUARTA posizione dell'*array* il valore 34.

Nota bene: come succede in molti linguaggi meno evoluti, anche Arduino NON protegge da errori di programmazione nell'accesso ad un *array*, cioè tentando di accedere ad un elemento OLTRE la fine dell'*array* stesso, ad esempio:

```
arrayDiInteri[666]=34;
```

l'effetto è quello di scrivere in una zona di memoria non nota a priori con conseguenze quasi certamente distruttive per il buon funzionamento del programma e per la salute mentale dello sviluppatore.

9.4.4 Direttive #define e #include

Si trovano frequentemente, nei sorgenti, le cosiddette "direttive al precompilatore", ad esempio:

```
#include <Ethernet.h>
#define DELAY 500
```

[10] Gli indici si dicono in questo caso "0-*based*", altri linguaggi iniziano da 1 e si chiamano infatti "1-*based*".

Si chiamano così perché è proprio la fase precedente la compilazione del codice che le "risolve" (le interpreta, dando semantica alla direttiva stessa) prima di passare il sorgente al compilatore vero e proprio. Il precompilatore è uno strumento che trasforma il sorgente: quando "esegue" l'*include* sostituisce alla riga dell'*include* stesso tutto il contenuto del file indicato tra le "<>" (nell'esempio si tratta del file `Ethernet.h`). Il sorgente trasformato viene poi passato al compilatore, tutto in maniera automatica e trasparente allo sviluppatore.

Date le due direttive sopra esposte, la seconda in particolare, se nel codice dello *sketch* si trovasse una porzione come la seguente:

```
void loop(){
   digitalWrite(13,HIGH);
   delay(DELAY);
   digitalWrite(13,LOW);
   delay(DELAY);
}
```

il sorgente realmente compilato sarebbe:

```
void loop(){
   digitalWrite(13,HIGH);
   delay(500);
   digitalWrite(13,LOW);
   delay(500);
}
```

Queste direttive servono a definire "pezzi" di codice di uso comune, anche a più progetti.

9.4.5 Commenti

E' possibile "commentare" parti di codice sia nel senso proprio di inserire commenti esplicativi sia in quello di marcare parti di codice che il compilatore dovrà ignorare. Un esempio di commento esplicativo è il seguente:

```
// accende il LED
digitalWrite(13,HIGH);
```

Un esempio di codice che va ignorato:

```
/*
  digitalWrite(13,HIGH);
  delay(DELAY);
  digitalWrite(13,LOW);
  delay(DELAY);
*/
```

La sintassi dei commenti prevede il '//' per far ignorare la riga di codice dal '//' in poi oppure la coppia '/*' (inizio commento) e '*/' (fine commento) per racchiudere intere parti di codice marcate come "da ignorare".

9.4.6 Costrutti di "selezione"

Un linguaggio cosiddetto "Turing-completo"[26, 56], oltre alla possibilità di rappresentare un stato (attraverso le variabili), deve fornire anche una serie di costrutti che permettono la "decisione", cioè la possibilità di scegliere rami diversi di esecuzione in funzione di condizioni. Nel caso di Arduino i costrutti sono *if* e *switch-case*. Il primo ha questa forma:

```
if (millis() % 1000 == 0) {
  // azione eseguita
  // se condizione VERA
}
else {
  // azione eseguita
  // se condizione FALSA
}
```

Nelle parentesi tonde dopo l'*if* si può inserire una "espressione" (cfr. sezione 9.4.2) che dia un valore booleano (vero-falso) o numerico (maggiore di 0 equivale a vero). Il ramo *else* è opzionale e si può omettere. Il secondo costrutto, più articolato che permette di scegliere fra più "casi", è lo *switch* e ha la forma di questo esempio:

```
switch(variabileIntera) {
  case 34:
    // codice eseguito
    // se variabile == 34
    break;
  case 99:
```

```
    // codice eseguito
    // se variabile == 99
    break;
  default:
    // codice eseguito
    // in tutti gli altri casi
  break;
}
```

La scelta avviene in funzione del valore dell'espressione nelle parentesi tonde dello *switch*.

Il valore dell'espressione viene confrontato con ogni argomento dei *case*: l'esecuzione del codice inizia dalla prima concordanza, e termina al successivo *break*. Dimenticare un *break* vuol dire eseguire TUTTI i casi dalla prima concordanza in poi.

In questo caso, diversamente dalla condizione del costrutto *if* nei *case* vanno messi valori *hardcoded* (fissati) e non espressioni.

9.4.7 Costrutti di "ripetizione"

Oltre ai costrutti di "selezione" servono ovviamente quelli di "ripetizione", cioè meccanismi che permettano di ripetere un insieme di istruzioni per un dato numero di volte o sulla base del valore di una espressione condizionale (es. fino a che una variabile assume un particolare valore).

Il primo costrutto che esaminiamo è il *for*, eccone un esempio:

```
for (int i=1; i <= 10; i++){
   Serial.print(i);
}
```

Il cuore di tutto risiede nelle parentesi dopo la parola chiave *for*, lì si inseriscono tre direttive chiamate "inizializzazione", "condizione" e "incremento"[11], separate dal carattere ';', sono tutte opzionali (cioè possono essere anche vuote). Nell'esempio sopra riportato il ciclo inizializza la cosiddetta "variabile di conteggio" 'i' al valore 1, esegue l'istruzione *print* fino a che la variabile non raggiunge il valore 10 ed incrementa di 1 la variabile ad ogni iterazione.

[11] Che può essere anche un decremento o una qualunque operazione.

L'altro costrutto interessante è il *while* (assieme al *do...while* che differisce solo per l'ordine di valutazione della condizione). Il *while*, rispetto al *for*, è orientato alla ripetizione "su condizione", cioè un insieme di istruzioni viene ripetuto finché una condizione non si avvera (o si falsifica).

Vediamone un esempio:

```
while(millis() < 60000){
  // il codice qui dentro viene ripetuto per i
  // primi 60 secondi dall'accensione
}
```

9.4.8 Definizione di funzioni

Con questo termine indichiamo la possibilità di associare un insieme di istruzioni ad un nome per poter usare quel "pezzo di codice" più volte nel nostro programma. Una volta definita, la funzione può essere "invocata" ogni volta che si vuole. Una funzione viene definita nel seguente modo:

```
float calcolo(float opA, float opB){
   return (opA+opB)*4;
}
```

Analizziamo la prima riga:

float è il tipo del "valore di ritorno" della funzione stessa, cioè il tipo di variabile che verrà restituito a chi "invocherà" questa funzione

calcolo è il nome associato alla funzione

float opA è la "variabile" che viene utilizzata come primo parametro di "ingresso" per la funzione

float opB è la "variabile" che viene utilizzata come secondo parametro di "ingresso" per la funzione

A questo punto possiamo esaminare un esempio di invocazione:

```
calcolo(5.5,3.2);
```

Che restituirà il valore 34.8, derivato dall'espansione di (5.5+3.2)*4. Nota bene: in Arduino la definizione della funzione può apparire ovunque nel codice, anche successivamente all'invocazione, cosa che non vale in tutti i linguaggi.

9.4.9 Funzioni predefinite

Ora che sappiamo come si definiscono nuove funzioni, per non reinventare la ruota, vediamo prima quelle predefinite! Già, perché Arduino per nostra fortuna offre alcune (nemmeno poche) cosiddette "funzioni di libreria standard" subito disponibili, senza dover aggiungere nulla al proprio codice. In ogni caso è anche possibile includere (con la direttiva `#include` spiegata sopra) altre librerie fornite con l'ambiente di programmazione. Qui di seguito ne elenchiamo qualcuna, i dettagli e il resto delle funzioni sono descritte sulla documentazione ufficiale sul sito `http://www.arduino.cc/en/Reference/HomePage`.

Matematiche

min(), max() minimo e massimo
abs() valore assoluto (toglie il segno)
sqrt() radice quadrata
sin(), cos(), tan() seno, coseno e tangente
random() genera un numero pseudocasuale

Tempo

millis() restituisce il numero di millisecondi trascorsi dall'accensione dell'Arduino
delay() sospende l'esecuzione del programma per un determinato periodo

I/O

Le funzioni di I/O (Input/Output) in questo contesto sono fondamentali, sono il mezzo con cui comunicare con il mondo fisico. I pin (piedini) di I/O dell'Arduino sono divisi in:
digitali solo due valori: HIGH="acceso" e LOW="spento"
analogici gestiscono valori "quantizzati" (256 o 1024 valori discreti) per livelli di tensione tra 0 Volt e una tensione di riferimento (ad es. 5 Volt)

Ogni piedino può essere usato in lettura o scrittura, ma bisogna prima configurarlo con, ad esempio:

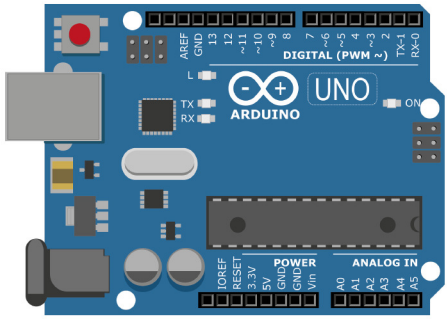


Figura 9.1: Piedinatura Arduino UNO

```
pinMode(5, OUTPUT);
```

che configura il piedino 5 per l'utilizzo in OUTPUT, cioè in "scrittura". "**Scrivere**" un valore su un piedino significa farvi uscire una tensione specifica, ad esempio:

```
digitalWrite(4,HIGH);
```

"alza" il 4 cioè emette 5 Volt sul piedino 4. Se il pin in questione è marcato con l'etichetta '∼' (figura 9.1, piedini 3, 5, 6, 9, 10, 11) allora è possibile emettere una "tensione variabile" (in realtà in PWM - Pulse Width Modulation - cfr. 3.4) usando ad esempio:

```
analogWrite(3,145);
```

che emette una "tensione PWM" equivalente a 145/255 di 5 Volt (circa 2.8 Volt) sul pin 3.

Invece per "**leggere**" informazione esterna si usa principalmente la funzione:

```
boolean valore=digitalRead(4);
```

che legge lo stato elettrico (acceso/spento) del pin digitale 4 e:

```
inf valore=analogRead(2);
```

che legge la tensione (restituendo un numero tra 0 e 1023 corrispondente a tensioni tra 0 e 5 Volt o tra 0 e la tensione presente sul pin AREF) presente sul piedino A2.

9.4.10 Uso della seriale

L'interfaccia USB dell'Arduino viene utilizzata per caricare nella flash memory lo *sketch*, ma può servire anche come canale di comunicazione durante l'esecuzione del programma. Usando il "Serial Monitor" (voce di menu) dell'IDE è possibile aprire un "terminale seriale" che riporta a video ciò che arriva dall'Arduino, inoltre permette di inviare caratteri e stringhe leggibili programmativamente. E' la classe `Serial` a offrire tutte le funzionalità per la comunicazione bidirezionale. La classe si inizializza, normalmente nel metodo setup dello *sketch*, con:

```
Serial.begin(19200);
```

in cui 19200 rappresenta la velocità (*baudrate*) di comunicazione, deve corrispondere all'impostazione nel Serial Monitor. Nel programma è possibile a questo punto inserire combinazioni delle seguenti:

```
Serial.println(``stringa con newline finale'');
Serial.println(valore); // con newline
Serial.print(valore); // no newline
```

Per leggere dati provenienti dall'utente si usa ad esempio:

```
char c=Serial.read();
```

che prende un singolo carattere tra quelli inseriti a terminale, o:

```
String s=Serial.readString();
```

che invece legge un'intera stringa di testo (fino a che trova caratteri nel *buffer* di ingresso) in un colpo solo.

9.4.11 Gestione interrupt

Senza entrare nei dettagli architetturali degli *interrupt* (per i quali si rimanda al classico Tanenbaum [65]) basti dire che il meccanismo in Arduino è molto semplice, sia concettualmente che programmativamente: permette l'attivazione di una "azione" (i.e., invocazione di funzione) in corrispondenza di un "avvenimento" (segnale su un *pin*).

In uno *sketch*, se si vuole gestire un *interrupt* bisogna seguire pochi passi:

- definire almeno una funzione "speciale", cosiddetta ISR (*Interrupt Service Routine*), che non è altro che una semplice funzione `void` (non ha valore di ritorno) e senza parametri in ingresso[12]
- "agganciare" (`attachInterrupt()`) la funzione a uno o più interrupt per attivare la reattività dello *sketch* al segnale in ingresso

Da questo momento in poi lo *sketch* si comporterà normalmente (esecuzione del `loop()`) e, all'arrivo di un segnale su uno dei piedini agganciati ad una ISR, l'esecuzione verrà sospesa (messa in *stack*) per dare seguito alla funzione ISR. Al ritorno della funzione ISR l'esecuzione normale verrà ripresa nello stato originale (ripreso dallo *stack*). Le funzioni di libreria di Arduino per la configurazione della gestione interrupt sono:

attachInterrupt(...) aggancia una funzione ISR ad un segnale;

detachInterrupt(...) sgancia una funzione ISR;

noInterrupts() disabilita gli interrupt, si usa per marcare sezioni (cosiddette "sezioni critiche") di codice di uno *sketch* che non devono essere "disturbate" da "distrazioni" (dover gestire un interrupt), anche un ISR potrebbe disabilitare gli interrupt all'ingresso per non essere invocata di nuovo in caso di altro evento scatenante o, in generale, per non essere interrotta a sua volta;

interrupts() abilita gli interrupt, fine della "sezione critica".

Analizziamo la prima funzione, tutto sommato la più importante, la sua forma base, anche se di **uso non raccomandato**:

```
attachInterrupt(interrupt, ISR, mode)
```

ci serve solo per capire **quali informazioni** servono per configurare una risposta a interrupt.

Le informazioni sono:

[12] Si differenzia da funzioni "normali" solo per il vincolo (non imposto né verificato dal compilatore) di dover essere molto breve e veloce perché durante la sua esecuzione potrebbe bloccare tutte le funzioni ISR.

interrupt quale interrupt si intende agganciare (si veda sotto, la forma raccomandata)
ISR puntatore alla funzione di risposta
mode "modo", uno fra:
- **LOW** scatta quando il segnale è "basso" ($0V$)
- **CHANGE** scatta quando il segnale cambia di stato ($alto- > basso$ o viceversa)
- **RISING** scatta quando il segnale "va alto" ($basso- > alto$)
- **FALLING** scatta quando il segnale "va basso" ($alto- > basso$)
- **HIGH** [13] scatta quando il segnale è "alto" ($5V$ o analogo livello, ad es. $3V3$)

Dobbiamo cioè comunicare: quale *interrupt* vogliamo agganciare, quale azione vogliamo scatenare e quale condizione del segnale farà scattare l'azione.

La forma **raccomandata** dalla documentazione Arduino è la seguente:

```
attachInterrupt(digitalPinToInterrupt(pin), ISR, mode
    ↪ )
```

che differisce dalla precedente solo per il primo argomento, invece di specificare direttamente l'*interrupt*, dato che sulle varie versioni di Arduino la corrispondenza piedino-*interrupt* è variabile, si utilizza una funzione di libreria che restituisce il numero dell'*interrupt* corrispondente al piedino che si desidera "ascoltare".

9.4.12 *Shield* (con esempio)

Come già accennato, Arduino ha fatto nascere un ecosistema di prodotti "compatibili" (a vari livelli), molti dei quali sono *pluggable*, sono cioè agganciabili direttamente, a pressione, senza saldature, sulla zoccolatura/piedinatura di un Arduino (figura 9.2). Questi prodotti, che prendono il nome di *shield* (scudi, infatti vengono impilati sopra all'Arduino, proteggendolo in qualche modo), sono schede già pronte per funzionalità ormai

[13] Quest'ultimo supportato solo Arduino Due.

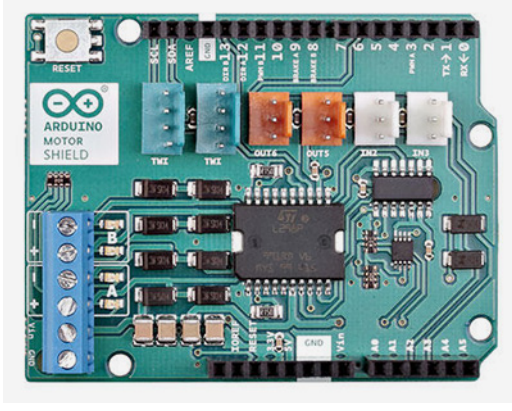


Figura 9.2: *Shield* motori (fonte: Arduino)

Figura 9.3: "*Shield*" rete

standardizzate come ad esempio motori in corrente continua e passo/passo, relè, rete ethernet, rete cellulare, WiFi, bluetooth, ecc. Alcuni altri, pur chiamati colloquialmente ancora *shield*, non sono direttamente impilabili (figura 9.3), ma vanno connessi mediante cavetti *patch*. L'enorme vantaggio di questi *shield* è che lo sviluppatore deve preoccuparsi solo di verificare la adeguatezza per il proprio progetto e poi caricare la libreria (che viene quasi sempre fornita col prodotto) di alto livello.

Vediamo, attraverso alcuni frammenti di codice, come integrare la funzionalità di uno *shield* Ethernet (figura 9.3) nel proprio *sketch*:

```
/* Including ENC28J60 libraries */
#include <EtherCard.h>
#include <IPAddress.h>

/* LAN-unique MAC address for ENC28J60 controller */
static byte mymac[]={0x70,0x69,0x69,0x2D,0x30,0x31};

/* TCP/IP send/receive buffer */
byte Ethernet::buffer[500];

...

void setup() {
    ...

    // inizializzazione
    if (ether.begin(sizeof Ethernet::buffer, mymac)
        ↪ ==0)
        Serial.println("Failed to access Ethernet");

    // richiesta indirizzo IP (via DHCP)
    if (!ether.dhcpSetup())
        Serial.println("DHCP failed");

    ether.printIp("IP:  ", ether.myip);
    ether.printIp("GW:  ", ether.gwip);
    ether.printIp("DNS: ", ether.dnsip);
    ...
}
...
```

Come si può notare è sufficiente includere qualche libreria per avere accesso a nuove funzioni che permettono la gestione

ad alto livello della connettività in rete.

Capitolo 10

Rete e protocolli

Nel contesto informatico/telecomunicazioni, con il termine rete si tende ad indicare un insieme di nodi distinti (non necessariamente paritari) ed un insieme di collegamenti tra i vari nodi.

Nel mondo IoT ("**Internet** Of Things", appunto), la rete gioca un ruolo fondamentale.

Presentiamo quindi rapidamente i due modelli (ISO/OSI e TCP/IP) utilizzati per definire uno *stack* (pila) di rete per poi descrivere i principali protocolli[1] usati nell'IoT.

Nella figura 10.1 è possibile apprezzare due delle "pile" di riferimento nel mondo della comunicazione tra elaboratori: a sinistra c'è la pila **ISO/OSI**[2], che rappresenta il modello ideale e di riferimento nella progettazione di reti definita nello standard ISO-7498 del 1984. A destra invece è rappresentato il modello omologo di una implementazione diventata lo standard *de fac-*

[1] Un "protocollo" è uno schema di comunicazione che specifica le modalità (tempi e formati dei messaggi) con cui due soggetti si scambiano dati/informazioni. Se i due soggetti che vogliono "parlarsi" usano lo stesso protocollo possono comprendersi. Viceversa, nel migliore dei casi non si capiscono per nulla, nel peggiore il ricevente interpreta un messaggio diverso da quello inteso dal mittente.

[2] Open System Interconnection

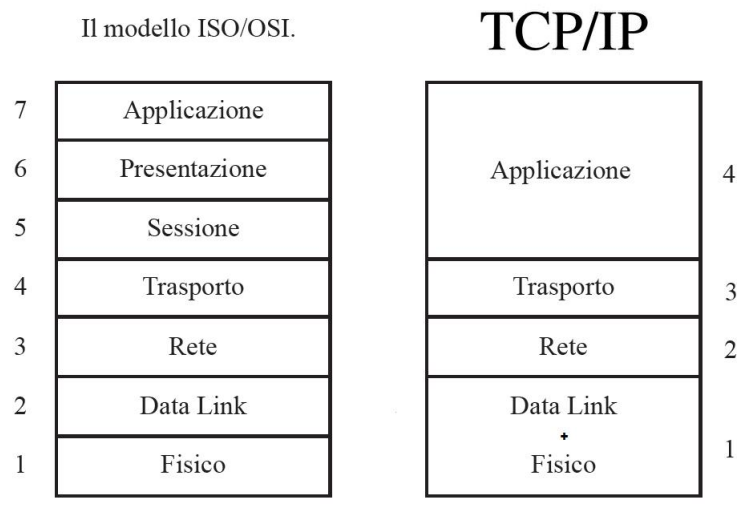


Figura 10.1: Comparazione della pila ISO/OSI e TCP/IP

to[3] nella comunicazione tra computer, il **TCP/IP**[4]. Come si può intuire dalla figura, non esiste una corrispondenza "uno ad uno" tra i livelli di una pila ed un altra.

I livelli della pila ISO/OSI sono i seguenti:

1. **Fisico**: gestione della trasmissione fisica dell'informazione, a livello di segnali, tensioni, frequenze e pin dei connettori;
2. **Collegamento**: livello che prepara i dati (sotto forma di pacchetti) alla trasmissione fisica, incapsulando l'informazione in *frame*, oltre che gestire il controllo di flusso e di errore;
3. **Rete**: un livello di astrazione che rende indipendenti i livelli inferiori da quelli superiori, e permette di "indirizzare" elaboratori che utilizzano livelli fisici e collegamento

[3] Termine latino atto ad indicare un elemento che è nella pratica in vigore, pur non avendo un riconoscimento ufficiale

[4] *Transmission Control Protocol/Internet Protocol*

diversi; questo livello è responsabile anche dell'instradamento (*routing*) dei pacchetti;
4. **Trasporto**: livello che si occupa di stabilire, mantenere e terminare connessioni tra elaboratori, con suppletivi controlli di errore e congestione;
5. **Sessione**: livello che si occupa di stabilire, mantenere e terminare connessioni tra le varie applicazioni;
6. **Presentazione**: livello che trasforma i dati dell'applicazione in modo standard, gestendone la protezione e la compressione;
7. **Applicazione**: il livello in cui i dati vengono generati.

Volendo (e)semplificare, si può immaginare un utente che, tramite una certa applicazione, deve mandare un messaggio ad un altro utente che utilizza la medesima applicazione su un altro elaboratore. Partendo dal livello 7, i dati inseriti tramite la tastiera, vengono passati al livello 6 dove sono compressi e crittografati. Successivamente il livello 5 fa partire un tentativo di connessione verso l'omologo livello 5 dell'elaboratore bersaglio. Per fare ciò devono essere identificati gli elaboratori (*hosts*) a livello 4 e deve essere indirizzata la comunicazione a livello 3. Nel livello 2 le informazioni vengono trasformate in una forma trasmissibile dal livello 1, che a sua volta le trasforma in segnali fisici. In ognuno di questi passaggi, all'informazione iniziale, vengono aggiunte ulteriori informazioni, analogamente ad un pacco che venga incartato più volte. Nella pratica comune, incartando più volte il medesimo pacco, è facile constatare come le sue dimensioni aumentino ad ogni passaggio. Come nell'esempio del pacchetto, il ricevente dovrà scartare i singoli incartamenti uno ad uno per poter raggiungere l'informazione contenuta del pacchetto originale. Ogni livello del ricevente scarta il proprio incartamento fino ad arrivare a presentare l'informazione al livello 7 del ricevente. Apparentemente tutti questi "incartamenti" potrebbero sembrare ridondanti, e potrebbe sfuggire l'utilità di avere tutti questi strati. Nei fatti, questa apparente complessità permette di usare standard definiti per ogni singola funzione, ed ogni *layer* ha bisogno di poter lavorare su informazioni che non siano ambigue; ogni strato riconosce le informazioni di cui necessita, e ciò permette agli sviluppatori di

hardware e driver di non doversi preoccupare del tipo di informazione che transita attraverso il dispositivo di rete e quale sia la sua collocazione nella rete stessa e contestualmente consente agli sviluppatori delle applicazioni di poter ignorare (con gli opportuni limiti) i dettagli del canale fisico e di come il sistema operativo lo gestisca.

Un comportamento simile è riscontrabile nel TCP/IP, dove i livelli 5, 6 e 7 sono fusi in un unico livello "Applicazione" (al cui interno possiamo trovare protocolli quali **HTTP**[5], **FTP**[6], **SMTP**[7] ed altri). A livello trasmissione si utilizzano prevalentemente i protocolli **TCP** ed **UDP**[8]: la differenza tra questi due protocolli è che TCP è un protocollo *Connection Oriented*[9] mentre UDP è un protocollo *Connectionless*[10]. Tali differenze li rendono adatti ad ambiti applicativi differenti. Sotto il livello trasmissione si trova il livello rete, che nel TCP/IP è proprio IP. Questo formalizza gli indirizzi delle interfacce di rete e gestisce i vari algoritmi di instradamento. Infine, vi sono i livelli 2 ed 1 fusi insieme in un unico *Network Access Layer*.

Va aggiunto che tra i modelli ISO/OSI e TCP/IP non esiste una perfetta sovrapposizione tra i livelli; ciò è dovuto in buona parta al fatto che in realtà TCP/IP è assai antecedente (1973) rispetto alla formalizzazione ISO/OSI. Per maggiori informazioni ed approfondimenti si rimanda il lettore a [66].

10.1 Topologie di rete

Nel mondo delle telecomunicazioni vi è stata un'incessante evoluzione che è allo sviluppo di svariate tipologie di rete, distinguibili per vari fattori:

[5] HyperText Trasfer Protocol
[6] File Transfer Protocol
[7] Simple Mail Transfer Protocol
[8] User Datagram Protocol
[9] In questo contesto indica un protocollo che si preoccupa di mantenere attiva la connessione, utilizza comandi di controllo e cerca di garantire la trasmissione dei pacchetti o la notifica della loro mancata trasmissione
[10] Indica un protocollo in cui non è garantito l'arrivo dei pacchetti, e che ha una struttura più "leggera" senza controllo connessione

Estensione Geografica: sulla base della superficie coperta (es. BAN[11], PAN[12], LAN[13], MAN[14], WAN[15], GAN[16])

Canale Trasmissivo: il mezzo utilizzato per la comunicazione (es. Wireless, Cavo UTP, Linea Telefonica, Fibra Ottica, Satellitari, Reti Elettriche)

Topologia: la modalità in cui sono connessi i vari nodi (es. Lineare, Anello, Albero, Stella, Bus, Maglia)

Nei sistemi embedded, oltre alle classiche LAN, sono assai frequenti anche le BAN le PAN ed ovviamente in ambiente automobilistico le CAN (vedere sezione 5.2.6).

In base allo scopo del dispositivo, può variare anche la topologia di collegamento:

Lineare (*daisy-chain*): ogni nodo è collegato a due nodi adiacenti, tranne il primo e l'ultimo;

Anello (*ring*): ogni nodo è collegato a due nodi adiacenti (i.e., formano un cerchio);

Albero (*tree*): da ogni nodo possono partire catene lineari, partendo dalla radice (*root*) muovendosi sulle foglie (*leaf*);

Stella (*star*): tutti i nodi (*spokes*) sono collegati a un unico nodo centrale (*hub*);

Bus: tutti i nodi sono collegati a un unico canale condiviso di comunicazione;

Maglia (*mesh*): Tutti i nodi trasmettono dati ad altri (non tutti) nodi (grafo non necessariamente totalmente connesso).

Parlando di dispositivi IoT e di reti di sensori, le reti *mesh* stanno riscontrando un notevole successo.

10.2 Protocolli

A livello "applicativo" esistono molti protocolli di rete atti a scambiare dati nel contesto embedded/IoT, di solito improntati

[11] Body area network, ad esempio alcuni dei dispositivi "wearable" (indossabili) o dispositivi biomedicali

[12] Personal Area Network, quali ad esempio IrDA (seriale infrarosso), Bluetooth, ZigBee

[13] Local Area Network, ad esempio le ethernet

[14] Metropolitan Area Network, reti cittadine

[15] Wide Area Network, collegamenti tra zone distanti

[16] Global Area Network, collegamenti globali

alla semplicità di implementazione[17] e all'efficienza di trasmissione: la "capacità trasmissiva" non viene data per scontata, anzi, si presume una limitata ampiezza di banda anche perché l'uso tipico prevede l'invio di dati di sensori/attuatori.

10.2.1 MQTT

MQTT (Message Queuing Telemetry Transport, `http://mqtt.org`) è un protocollo di comunicazione "leggero"[18], appoggiato su TCP/IP (*connection oriented*) che si basa sul *pattern*[16] *publish-subscribe*[19]: messaggi (sperabilmente) informativi vengono inviati da una sorgente (un nodo "client" della rete MQTT) verso un *dispatching broker* che li ridistribuisce a tutti i nodi che si erano dichiarati interessati a quel tipo (*topic*) di messaggio.

Ogni messaggio MQTT è composto da:

- un *topic* (metaforicamente analogo al *Subject* di una mail) che servirà al *broker* per la distribuzione, è una stringa il cui contenuto è libero, ma viene specificata la forma sintattica da utilizzare per costruirlo (parole separate dal carattere "/");
- un *payload*, il corpo del messaggio, è semplicemente una stringa UTF-8, la semantica di tale stringa è affidata alle convenzioni stabilite tra chi manda e chi riceve, non è definita nel protocollo.

Il *broker* gestisce una lista di nodi client interessati ai vari *topic*, ogni messaggio che riceve viene inviato solo ai nodi che si erano "iscritti" a quel particolare *topic*. I nodi che si registrano possono indicare come "interesse" anche una *wildcard* (una stringa contenente caratteri speciali che rappresenta un insieme di stringhe effettive, si veda esempio nel seguito) invece di un *topic* per esteso.

E' uno standard OASIS dal 2014, il documento di specifica formale del protocollo consta di circa ottanta pagine e può es-

[17] Ricordiamo che la potenza di calcolo e la memoria in questo contesto non sono sempre disponibili in grande quantità.

[18] Tra virgolette poiché lo standard non fornisce una definizione di leggerezza legata al contesto implementativo.

[19] Noto anche come *observer-observable*, si veda [21].

sere consultato all'URL: `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf`.

Per capire meglio l'architettura di MQTT vediamo un esempio di *workflow* in una ipotetica rete d'appartamento composta dai seguenti nodi:

- *SensoreTempSala*: misura la temperatura della sala, invia messaggi della forma:
  `Appartamento/Sala/Temperatura <valore>`[20]
- *SensoreTempCucina*: misura la temperatura della cucina, invia messaggi della forma:
  `Appartamento/Cucina/Temperatura <valore>`
- *SensoreUmidSala*: misura l'umidità della sala, invia messaggi della forma:
  `Appartamento/Sala/Umidita <valore>`
- *SensoreUmidCucina*: misura l'umidità della cucina, invia messaggi della forma:
  `Appartamento/Cucina/Umidita <valore>`
- *VisualizzatoreDatiTemperatura*: visualizza in modalità grafica i dati di temperatura puntuali e storici, non invia messaggi ma si iscrive al *broker* segnalando il *topic*:
  `Appartamento/+/Temperatura`
- *VisualizzatoreDatiUmidità*: visualizza in modalità grafica i dati di umidità puntuali e storici, non invia messaggi ma si iscrive al *broker* segnalando il *topic*:
  `Appartamento/+/Umidita`
- *AllarmeCucina*: attiva un allarme se la temperatura della cucina supera un certo valore, si iscrive al *broker* segnalando il *topic*:
  `Appartamento/Cucina/Temperatura`
- *Broker*: attende richieste di iscrizione da parte dei nodi e distribuirà i messaggi in funzione degli "interessi"

Come schematizzato in figura 10.2 si noti che ogni nodo "parla" (cioè si connette) solo col *broker* che si occuperà di distribuire poi l'informazione in funzione delle richieste di *topic*.

Si noti anche l'uso del *wildcard* "+" che, assieme a "#", permette di specificare un insieme di *topic* usando una sola stringa: "+" corrisponde a qualunque parola SENZA "/" mentre "#" a

[20]Notazione: *topic*, spazio, messaggio.

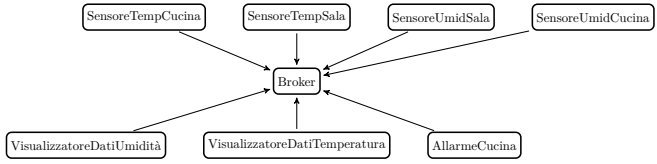


Figura 10.2: Casa MQTT

qualunque parola che non inizi per "/" ma che lo può contenere. Nella situazione sopra riportata la stringa
`Appartamento/+/Temperatura`
corrisponde a qualunque *topic* della forma
`Appartamento/<parola>/Temperatura`
cioè ad esempio:
`Appartamento/Sala/Temperatura`
`Appartamento/Cucina/Temperatura`
`Appartamento/Camera/Temperatura`
`Appartamento/Letto/Temperatura`
`Appartamento/Sgabuzzino/Temperatura`
`etc.`

Se usassimo invece un *topic* tipo
`Appartamento/#`
esso corrisponderebbe a qualunque *topic* della forma
`Appartamento/<parola>/<altraparola>/<ancora>/.../...`
cioè qualunque *topic* a partire da "Appartamento" in poi.

A regime possiamo immaginare una "strisciata" di messaggi che arrivano al *broker*, ad esempio:

```
...
Appartamento/Sala/Temperatura 21
Appartamento/Sala/Umidita 40
Appartamento/Cucina/Temperatura 22
Appartamento/Cucina/Umidita 43
Appartamento/Sala/Temperatura 19
Appartamento/Cucina/Temperatura 20
Appartamento/Cucina/Umidita 43
Appartamento/Sala/Temperatura 21
Appartamento/Sala/Umidita 40
Appartamento/Sala/Temperatura 18
Appartamento/Cucina/Umidita 47
Appartamento/Cucina/Temperatura 35
```

```
Appartamento/Sala/Temperatura 21
Appartamento/Sala/Umidita 40
...
```

Il *broker* girerà ai vari nodi solo i seguenti messaggi:

- al **VisualizzatoreDatiTemperatura**:

```
...
Appartamento/Sala/Temperatura 21
Appartamento/Cucina/Temperatura 22
Appartamento/Sala/Temperatura 19
Appartamento/Cucina/Temperatura 20
Appartamento/Sala/Temperatura 21
Appartamento/Sala/Temperatura 18
Appartamento/Cucina/Temperatura 35
Appartamento/Sala/Temperatura 21
...
```

- al **VisualizzatoreDatiUmidità**:

```
...
Appartamento/Sala/Umidita 40
Appartamento/Cucina/Umidita 43
Appartamento/Cucina/Umidita 43
Appartamento/Sala/Umidita 40
Appartamento/Cucina/Umidita 47
Appartamento/Sala/Umidita 40
...
```

- al **AllarmeCucina**:

```
...
Appartamento/Cucina/Temperatura 22
Appartamento/Cucina/Temperatura 20
Appartamento/Cucina/Temperatura 35
...
```

Nelle varie implementazioni (sono disponibili in moltissimi linguaggi e sistemi operativi e su piattaforme sia embedded che "normali") le funzioni principali sono proprio le due:

- `Subscribe(String subTopic, Function callback)` permette (una volta connessi ad un *broker*) la dichiarazione di interesse verso uno o più *topic*, quando il *broker* riceve un messaggio il cui *topic* corrisponde con *subTopic* lo invia al client che lo riceve e scatena l'invocazione della

funzione *callback* che va implementata a cura del programmatore del nodo "subscriber" (i visualizzatori o l'allarme del nostro esempio);

- `Publish(String topic, String message)`
  permette (una volta connessi ad un *broker*) l'invio di un messaggio associato ad un *topic*, il *broker* analizzerà il *topic* e "girerà" il messaggio ai nodi il cui *subTopic* corrisponde.

10.2.2 LoRa e LoRaWAN

LoRa™ (Long Range) e LoRaWAN™ (Long Range Wide Area Network) sono due standard di rete *wireless* molto noti e usati nel contesto IoT (Internet of Things). Sono definiti da un comitato di aziende, la LoRa Alliance (`https://lora-alliance.org`).

Il primo standard[21] si riferisce al livello fisico ("PHY", quello più basso della "stratigrafia" ISO-OSI[66] e vedere sopra) cioè definisce le modalità di trasmissione (modulazione) e le frequenze utilizzate: 169 MHz, 433 MHz, 868 MHz e 915 MHz a seconda delle necessità e delle legislazioni dei vari paesi. La modulazione utilizzata è la cosiddetta "*chirp spread spectrum*" che sfrutta un segnale a frequenza variabile (il cosiddetto *sweep*) per codificare i bit da trasmettere.

Il secondo invece descrive lo strato "MAC" (*Media Access Control*) cioè il protocollo vero e proprio di trasmissione, i.e., indirizzamento, crittografia, formato dei dati, tipo di *acknowledgement*, ecc.

Sono standard *wireless* che sostituiscono (o affiancano, in applicazioni ibride come nel caso dello shield mostrato in figura 10.3) gli standard WiFi IEEE802.11x per supportare la comunicazione tra sensori e *sensor gateway* nei contesti in cui servono bassi consumi e lunghe "portate" (distanze raggiungibili dai segnali), le caratteristiche si possono riassumere in:

- bidirezionalità della comunicazione
- buona resistenza alle interferenze

[21] Purtroppo proprietario, anche se è stato analizzato [45] in modo *black box*.

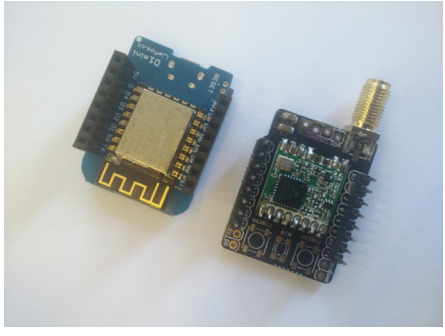


Figura 10.3: Shield LoRa (a dx) per MCU Wemos/ESP8266

- consumi ridotti
- comunicazioni a lungo raggio (anche dell'ordine dei Km)
- sicurezza (mediante crittazione) delle comunicazioni
- funzionamento anche *indoor*
  (soprattutto misto *out/indoor*)
- bassi costi di adozione

La specifica[61] pubblicata dalla LoRa Alliance descrive le reti LoRaWAN come organizzate in topologia a "stella di stelle" (figura 10.4) in cui esistono nodi *gateway* che fanno da accentratori e instradatori di messaggi fra nodi terminali (estremi dei raggi di una stella). Inoltre i *gateway* fungono opzionalmente da ponte verso reti tradizionali (e.g., Internet) essendo tipicamente connessi tramite TCP/IP. Le comunicazioni sono bidirezionali anche se il pattern predominante è quello in cui i *device* terminali (ad esempio sensori) inviano dati "verso l'alto", cioè verso i gateway e i vari server di raccolta dati.

Le comunicazioni tra i nodi e i *gateway* vengono effettuate su varie frequenze/canali e a velocità molto variabili. La scelta della velocità di trasmissione viene fatta (nei programmi che gestiscono i nodi) in modo da ottimizzare portata, durata dell'invio e possibili interferenze fra nodi vicini. Le velocità di trasmissione disponibili spaziano dai 0.3 kbps[22] fino ai 50 kbps.

[22]Kilobit per secondo.

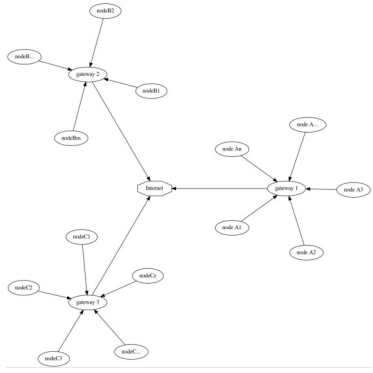


Figura 10.4: Topologia LoRa

Anche la potenza di trasmissione viene adattata al contesto da un minimo di 1 mW fino ad arrivare al Watt.

Ogni nodo può trasmettere su uno qualsiasi dei canali disponibili[23] per la propria configurazione, ma deve sceglierlo a caso (per minimizzare il rischio di collisioni fra due trasmissioni contemporanee) e deve rispettare le finestre temporali di trasmissione specifiche per il profilo scelto[24]. Inoltre, per risparmiare potenza, sui nodi (tipicamente a batteria) la ricezione è "schedulata", cioè il modulo ricevente viene attivato (alimentato) solo quando si prevede di poter ricevere messaggi.

Infatti, per quanto riguarda le modalità e le tempistiche di trasmissione/ricezione, i nodi possono essere catalogati secondo tre tipi/classi:

- **Classe A**: nodi bidirezionali "semplici", ogni invio prevede anche due finestre temporali di ricezione, quindi il nodo può ricevere informazioni solo a valle di aver inviato qualche messaggio, questa è la configurazione a minor consumo di energia;

[23] L'effettiva scelta dipende anche dalla legislazione vigente nel paese di installazione del device.

[24] E' un sistema analogo alle trasmissioni radiofoniche professionali (ad esempio quelle marine) dove si devono rispettare regole di "silenzio radio" (i primi tre minuti a partire dall'ora intera e dalla mezz'ora) e di durata (usare il canale per periodi consecutivi inferiori ai 30 secondi).

- **Classe B**: nodi bidirezionali con ricezione "schedulata", come per la classe A con in più la possibilità di definire finestre temporali di ricezione slegate dagli invii. Ovviamente serve un meccanismo di sincronizzazione degli orologi di bordo;
- **Classe C**: nodi bidirezionali con ricezione continua, il ricevitore risulta sempre alimentato per cui la ricezione di eventuali messaggi è completamente asincrona. Ovviamente il costo energetico in questo caso è massimo.

Appendice A

Ambiente di Test

Quale che sia la dimensione del progetto o le caratteristiche hardware utilizzato per la realizzazione, è in generale considerata una buona pratica disporre di ambienti di sviluppo per codice e configurazioni (il più delle volte sono *Workstation* con particolari **IDE**[1]) e di un opportuno ambiente di test, atto ad imitare o talvolta emulare il comportamento del sistema desiderato.

A.1 Linux Test Environment

Durante lo sviluppo di un progetto *Embedded Linux* non è possibile lavorare direttamente su quella che è l'immagine finale di sistema. I numerosi tentativi dovuti alla realizzazione di una configurazione di kernel ottimale, o di verifica funzionamento del *rootfs*, rendono poco pratico il lavorare direttamente su un supporto fisico. Per quanto sia possibile ad oggi eseguire un discreto lavoro con il *debug in place* (debug sul posto), nelle fasi iniziali può risultare essere assi più conveniente provare il software utilizzando dispositivi esterni a supporto, con degli opportuni accorgimenti all'interno nella configurazione del sistema. Per questi motivi risulta decisamente utile creare un

[1] *Integrated Development Environment*, tr. Ambiente di sviluppo integrato

ambiente di test, differente da quello che sarà il risultato finale "di produzione", ma comunque in grado di eguagliarne per quanto possibile il comportamento.

Non si può dire che esistano delle linee guida generali per tutti i dispositivi embedded che, come visto nella sezione dedicata all'hardware, sono suscettibili a numerose variazioni dal punto di vista della dotazione. Scendendo nello specifico delle evaluation board per SoC di fascia alta, la situazione tende ad essere più standardizzata; nella maggior parte dei casi (TI, ST, Freescale, Nvidia, RockChip, Mediatek per citarne alcuni) questi dispositivi dispongono di un collegamento di rete ethernet. Contrariamente a quanto avviene nel comune mondo desktop, in realtà, non è possibile definirla propriamente come un'unica scheda di rete, in quanto i livelli di questo collegamento possono (e sovente lo sono) essere disposti in "luoghi" differenti; il livello Mac viene ospitato all'interno del SoC, mentre il livello fisico (in gergo il "fisico" o phy) è collocato esternamente in prossimita (o internamente) del connettore (tipicamente un RJ45). Come già citato in precedenza nella sezione relativa all'hardware, nelle fasi finali di progettazione del prodotto il prototipo definitivo potrebbe non disporre più del connettore e del livello fisico della ethernet. Potrebbe infatti essere considerato inutile allo scopo del dispositivo, o addirittura un potenziale pericolo per la sicurezza del sistema. In questo senso, l'aggiunta di una ethernet nelle evaluation board, ad esclusione di casi eccezionali (es. prototipazione hardware di rete), è da intendersi proprio a scopo di sviluppo e testing. Grazie alle funzioni avanzate dei *bootloader* (vedere sezione 6.4) è infatti possibile caricare tutti gli elementi fondamentali del sistema attraverso la rete.

Come già ripetuto in numerose occasioni, il mondo dei sistemi embedded è assai variegato e specializzato: tale condizione non permette di fornire una guida universale alla creazione di un ambiente di test, ma solo di fornire un esempio che sia quanto più possibile vicino ai casi più comuni.

In questo caso considereremo un ambiente di test che sfrutta un comune PC (host) con sistema operativo GNU/Linux ed una evaluation board (target) dotata di bootloader utilizzabile (può succedere, in particolare in prodotti già ultimati che, nonostan-

te il bootloader sia presente e funzionante, non sia accessibile da parte dell'utente). Il sistema host può essere lo stesso sistema utilizzato come ambiente di sviluppo, anche se ciò non è strettamente necessario.

L'ambiente di test deve disporre di:

- Un PC Linux (possibilmente il sistema Host utilizzato come ambiente di sviluppo)
  Un server NFS[23] per la condivisione del rootfs
  Un server TFTP[24] per il caricamento delle immagini di Kernel e device-tree
- Collegamento ethernet tra Host e board (meglio se attraverso uno switch)
- Una porta seriale/cavo seriale.

Il termine porta seriale, utilizzato in questo contesto per chiarezza espositiva, in realtà potrebbe non essere il più indicato. Pur non rappresentando la totalità dei casi, nella maggioranza dei casi è possibile che la porta seriale non sia la classica RS-232 a 9 pin, ma una seriale TTL incompleta (termine utilizzato per indicare che solo alcuni fili sono realmente connessi) a voltaggio variabile (solitamente 3.3V o 5V). L'uso diretto di una seriale classica (con livelli molto più alti) potrebbe rivelarsi inutile e dannoso per la board stessa. Nelle board moderne e di fascia alta, la funzionalità seriale è offerta tramite porte USB OTG collegate ad un apposito convertitore (usb2serial) già impostato al corretto voltaggio.

Nel nostro caso è stato utilizzato il convertitore USB2Serial TTL presentato nella sezione Hardware. Quanto al PC Linux, si fa riferimento a quanto già segnalato nella fase di configurazione dell'ambiente di sviluppo ma con due aggiunte: i server 'nfs' e 'tftp'.

A.1.1 NFS

Come già visto nella sezione 7.3, NFS consente di esportare filesystem o parti di esse attraverso la rete. A differenza di molti altri server di simile funzionalità, questo è stato integrato all'interno del kernel Linux, limitando molto gli aspetti d'installazione a livello userspace. Nella configurazione Workstation di Fe-

dora 22 non è necessario installare nulla. Gli unici accorgimenti sono quelli in fase di configurazione.

Nel file "/etc/exports"

```
/home/build/rootfs/bbb *(rw,async,no_root_squash)
```

Il primo path indica il luogo dove è presente in forma decompressa il rootfs che si vuole utilizzare. Il valore "*" sta ad indicare le restrizioni di accesso a livello IP (potrebbe essere sostituito dall'indirizzo di una rete o di un singolo host). I parametri tra parentesi rappresentano le opzioni di mount del rootfs. Non resta che abilitare ed avviare il servizio, attraverso le apposite utility di systemd

```
# exportfs -va
# systemctl enable nfs-config.service
# systemctl enable nfs-idmapd.service
# systemctl enable nfs-mountd.service
# systemctl enable nfs-server.service
# systemctl start nfs-config.service
# systemctl start nfs-idmapd.service
# systemctl start nfs-mountd.service
# systemctl start nfs-server.service
```

Alcuni accorgimenti in ambito sicurezza potrebbero essere necessari; in sistemi dotati di firewall infatti la richiesta NFS della board potrebbe essere rifiutata, in particolare per problemi relativi alle RPC (remote procedure call) del server NFS che rispondono su porta arbitraria (a meno di configurazione) alle richieste. In alcuni casi potrebbe essere necessario (anche se poco raccomandabile) disabilitare il firewall di rete.

A.1.2 TFTP

Come già visto nella sezione 7.3, TFTP consente di caricare file attraverso un protocollo FTP semplificato (ideale per ambienti semplici come i bootloader). Per l'installazione è necessario eseguire:

```
# dnf install tftp-server
```

Un eempio del file di configurazione `/etc/xinetd.d/tftp`:

```
service tftp
{
        socket_type             = dgram
        protocol                = udp
        wait                    = yes
        user                    = root
        server                  = /usr/sbin/in.tftpd
        server_args             = -s /home/build/
            ↪ tftpboot
        disable                 = no
        per_source              = 11
        cps                     = 100 2
        flags                   = IPv4
}
```

Il percorso di "tftpboot" è una cartella non presente nel sistema, che deve essere creata e configurata manualmente. All'interno di questa cartella dovranno essere copiate le immagini di kernel e device-tree.

Anche in questo caso sono richiesti alcuni accorgimenti sulla sicurezza: oltre ad aprire sul firewall (qualora non sia abilitata) la porta 69 UDP, è necessaria una modifica nel comportamento di sicurezza del sistema host. Molti sistemi recenti utilizzano un meccanismo noto come SELinux, abilitato come scelta predefinita. Tale meccanismo non è molto compatibile con l'accesso di tftp, rendendolo di fatto inutilizzabile. Non sapendo quale sia il contesto di lavoro del sistema host, potrebbe risultare pratico impedire a SELinux di interferire con il lavoro di tftp-server con:

```
# setenforce 0
```

A.1.3 Utilizzo di NFS/TFTP in U-Boot

Contrariamente a quanto visto in precedenza, è necessaria una differente configurazione delle variabili di ambiente di U-boot (mostrate nella sezione 7.2.4) per poter utilizzare il boot da rete. Data l'estrema variabilità del sistema, e considerando il fatto di dover utilizzare il bootloader già presente all'interno della board, è poco proponibile l'utilizzo di un file `uEnv.txt`: in tal

senso, è più opportuno passare volta per volta questi parametri tramite la console seriale messa a disposizione di uboot.

```
setenv ipaddr 192.168.127.7
setenv serverip 192.168.127.1
setenv gatewayip 192.168.127.1
setenv netmask 255.255.255.0
setenv ipstring 'ip=192.168.127.7:192.168.127.1:
    ↪ 192.168.127.1:255.255.255.0:testsystem:eth0:::'
setenv netargs 'setenv bootargs console=console=tty00
    ↪ ,115200n8 root=/dev/nfs rootwait nfsroot
    ↪ =192.168.127.1:/home/build/rootfs/bbb,nolock,v3
    ↪  rw $ipstring'
setenv netcmd 'run netargs; tftp 0x80200000 uImage;
    ↪ tftp 0x80F80000 am335x-boneblack.dtb; bootm  0
    ↪ x80200000 - 0x80F80000'
```

Da notare che in questa configurazione è stato impostato un IP statico, anche se di frequente si preferisce utilizzare un server dhcp sul PC host in grado di fornire tutte queste informazioni.

Appendice B

Esempi pratici

Questo capitolo riporta alcuni di esempi per capire cosa è possibile realizzare con un sistema embedded a base GNU/Linux o Arduino.

B.1 Abilitare un LED tramite GPIO su Linux

Un esempio tipico di applicazione di un sistema embedded Linux è l'accensione è spegnimento di un LED tramite GPIO (comparabile al "Hello World" del primo programma scritto in C). Sia i LED (sez. 3.5) che i GPIO (sez. 5.2.8) sono stati già introdotti nei precedenti capitoli, per questo motivo, in questa sezione verranno presenti solamente le procedure operative per realizzare questo esempio. La board che utilizzeremo è la Creator CI20. I numeri di riferimento utilizzati per PIN e GPIO sono corretti esclusivamente per questa board, dato che ogni board utilizza una differente numerazione. Inoltre talvolta, in base al cosiddetto *device-tree* (sez. 6.2.2) la medesima board può avere una configurazione diversa, con una differente mappatura dei GPIO stessi che potrebbero essere utilizzati per altre periferiche.

Per iniziare è necessario collegare correttamente il LED: si collega l'anodo del LED al pin del GPIO selezionato, e catodo

alla massa (GND) della board. Per semplificare le fasi di collegamento e scollegamento, si suggerisce di evitare saldature ed utilizzare una breadboard. A salvaguardia del LED utilizzato, è spesso raccomandabile limitare la corrente aggiungendo una resistenza tra il pin del GPIO ed il LED (cfr. sezione 3.1.2).

Una volta collegato elettricamente il LED è necessario abilitare il GPIO, configurarlo, ed impostarne l'uscita. Nei sistemi Linux, i GPIO sono accessibili tramite l'interfacciamento nella cartella **/sys/class/gpio/**. Le modalità di funzionamento a basso livello dell'interfacciamento GPIO tramite il cosiddetto *sysfs* sono ampiamente descritte nella documentazione del kernel Linux[43, 41], ma è giusto ricordare anche in questa sede che nei sistemi Linux, vige la regola del "Everything is a file" (tes. "Tutto è un file"), che sta ad indicare come anche l'hardware venga reso disponibile sotto forma di file "speciali".

Ai fini di questo esempio è sufficiente considerare che tutti i GPIO disponibili possono essere controllati attraverso scritture su file presenti in questa cartella, al cui interno si trovano:

export: attraverso questo file è possibile fare il cosiddetto *export* di un determinato GPIO identificato attraverso un numero intero, rendendo visibile nella cartella */sys/class/gpio*. Senza fare l'export, non è possibile operare sul GPIO;

unexport: questo file è utilizzato per l'operazione inversa ad export; tale operazione rimuove i riferimenti ad un determinato GPIO, rimuovendone la relativa cartella;

gpioX: X è un numero intero che rappresenta il numero del GPIO. Una volta eseguita l'operazione di export su un GPIO "X" appare la cartella `gpioX`, dal quale è possibile controllarlo;

gpiochipX: X è un numero intero che rappresenta il controller dei GPIO. In molte *evaluation board* sono presenti più controller GPIO separati.

La procedura per abilitare il GPIO (nell'esempio si considererà il numero 124) parte dalla necessaria operazione di export:

```
$ cd /sys/class/gpio/
$ echo 124 > export
```

Con il comando `echo`[1] si va a richiedere al kernel di esportare il GPIO 124. Eseguita questa operazione, all'interno della cartella corrente apparirà la cartella `gpio124` in cui si potrà entrare con:

```
$ cd gpio124
```

Una volta entrati nella cartella sarà possibile notare due file:

direction: Il contenuto del file di testo può assumere due soli valori; **in** (*input*, GPIO configurato per leggere un valore dal *pin*) o **out** (*output*, GPIO configurato per far uscire un valore espresso nel file *value*);

value: può assumere valori 0 ed 1, ed in base alla configurazione di *direction* può essere letto (in) o scritto (out).

Dato che il GPIO in questione è configurato per funzionare in input e con valore 0, per accendere il LED è necessario modificare tale impostazione con:

```
$ echo out > direction
$ echo 1 > value
```

A questo punto, se il LED è configurato correttamente, si dovrebbe accendere. Per concludere, una volta ultimate le prove, è possibile (e talvolta raccomandabile) fare le operazioni in senso inverso per ritornare alla situazione iniziale:

```
$ echo 0 > value
$ cd ..
$ echo 124 > unexport
```

Con il primo comando di riporta il valore del GPIO a zero, con conseguente spegnimento del LED, mentre con il terzo si rimuove la cartella dello specifico GPIO dalla cartella */sys/class/gpio*.

[1] Comando da shell che è utilizzato per scrivere un output a schermo. Associato ad una redirezione (simbolo ">") è possibile redirigere l'output dallo schermo all'interno di un file.

B.2 Scrittura su I2C

A seguire un esempio di scrittura su bus I2C su una board ARM, con sistema GNU/Linux[2]. Per le modalità di funzionamento di I2C è possibile fare riferimento alla sezione 5.2.3.

```
int i2c_write(char *i2cdev, uint16_t slave_address,
    ↪ uint8_t *buffer, uint32_t count){
    int ret, fd;
    uint32_t nbBytesSent;
    fd= open(i2cdev, O_RDWR);
    if (fd < 0)  {
        fprintf(stderr, "i2c: Cannot write to
            ↪ unitialized bus.\n");
        return -1;
    }
    if (buffer == NULL) {
        fprintf(stderr, "i2c: Cannot write using
            ↪ invalid buffer.\n");
        return -1;
    }
    if (count == 0)
        return 0;
    if ((ret = i2c_select_slave(fd, slave_address)) <
        ↪  0)
        return ret;
    nbBytesSent = 0;
    while (nbBytesSent < count) {
        ret = write(fd, &buffer[nbBytesSent], count -
            ↪  nbBytesSent);
        if (ret < 0) {
            fprintf(stderr, "i2c: Failed to write.\n"
                ↪ );
            return -1;
        }
        nbBytesSent += ret;
    }
    close(fd);
    return nbBytesSent;
}
```

La funzione in oggetto accetta in ingresso i seguenti parametri:

[2]tratto da em_com: `https://github.com/axjslack/em_com`, rilasciato in GNU GPLv2

char *i2cdev: stringa sotto forma di puntatore a caratteri contenente l'astrazione Linux del dispositivo I2C (es. */dev/i2c-0*);

uint16_t slave_address: indirizzo esadecimale (rappresentato come numero intero senza segno) dello *slave*;

uint8_t *buffer: un array di valori interi privi di segno presentato come indirizzo;

uint32_t count: numero di dati da trasmettere[3].

In particolare è interessante notare come anche in questo caso valga la filosofia dei sistemi Unix/Linux (*"Everything is a file"*[4]) e di come il codice sfrutti i normali strumenti utilizzati per la gestione dei file ordinari (*open*[5], *close*[6], *write*[7]). Da sottolineare inoltre che la trasmissione dei dati avviene passando alla funzione write un byte (8bit) alla volta.

B.3 Una "termo-ventola" con Arduino

Cambiamo piattaforma presentando un caso classico del contesto Arduino: azionare un meccanismo in funzione di condizioni esterne lette attraverso un sensore.

Pensiamo ad una ventola la cui accensione viene controllata in funzione della temperatura dell'ambiente: si accende se la temperatura raggiunge una certa soglia di accensione (*activation trigger temp*) e si spegne se la temperatura scende sotto una soglia (diversa dalla precedente) di spegnimento (*deactivation trigger temp*). La "finestra" tra attivazione e spegnimento serve a evitare le situazioni "limite", se non ci fosse e si usasse una sola soglia potrebbero accadere casi in cui, con temperatura ambiente oscillante intorno alla soglia, la ventola continuerebbe

[3] Si potrebbe fare a meno di questo dato, a meno che non si voglia trasmettere un sottoinsieme dell'array di valori *buffer*

[4] Repetita iuvant.

[5] Funzione di libreria: restituisce un *file descriptor* dopo aver aperto un determinato file.

[6] Funzione di libreria: chiude i riferimenti al file, disassociando il *file descriptor*.

[7] Funzione di libreria: scrittura su file tramite *file descriptor*.

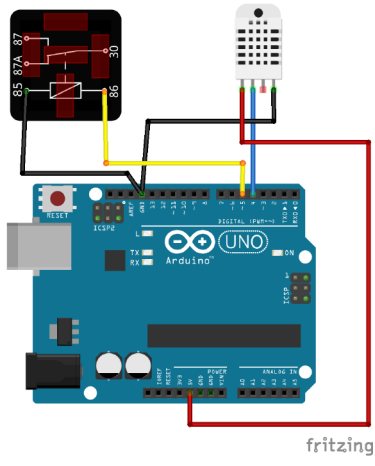


Figura B.1: Schema dei collegamenti

ad accendersi e spegnersi seguendo le minime variazioni della temperatura[8].

Per realizzare questo piccolo esempio abbiamo bisogno di:

- un Arduino UNO
- un sensore di temperatura (ad es. un DHT11)
- un relè
- una ventola
- una fonte di alimentazione

In figura B.1 viene presentato lo schema di massima del circuito (sono omessi per semplicità l'alimentatore connesso all'USB dell'Arduino, l'alimentazione della ventola e la ventola stessa, che andrà connessa al relè): il sensore DHT è collegato al piedino 4 (che quindi viene configurato come INPUT) mentre il relè è connesso al piedino 5 (che quindi viene configurato come OUTPUT).

Il software è relativamente semplice, iniziamo dalle definizioni delle costanti e variabili:

```
// libreria gestione DHT
```

[8]Si introduce cioè un certo grado di "isteresi" nel sistema.

```
#include "DHT.h"
#define DHTTYPE DHT11
#define DHTPIN 4
DHT dht(DHTPIN, DHTTYPE);

// costanti "comode"
#define ON      HIGH
#define OFF     LOW

// soglia e isteresi
int tempSoglia=26;
int finestraIsteresi=2;

// valori sensore
float humidity,temperature,fahreneit,hif,hic;

#define RELAY 5
```

Per poi passare all'inizializzazione, funzione `setup()`:

```
void setup() {
    Serial.begin(115200);
    Serial.println("Booting...");

    pinMode(RELAY, OUTPUT);
    pinMode(DHTPIN, INPUT);

    dht.begin(); // init sensore

    Serial.println("Boot complete!");
}
```

E concludere con il `loop()` che implementa il controllo del relè (e quindi della ventola):

```
void loop() {
    humidity = dht.readHumidity();
    temperature = dht.readTemperature();
    fahreneit = dht.readTemperature(true);

    // Check if any reads failed and exit early (to
        ↪ try again).
    if (isnan(humidity) || isnan(temperature) ||
        ↪ isnan(fahreneit)){
        Serial.println("Failed to read from DHT
            ↪ sensor!");
        return;
    }
```

```
    // Compute heat index in Fahrenheit (the default)
    hif = dht.computeHeatIndex(fahreneit, humidity);
    // Compute heat index in Celsius (isFahreheit =
        ↪ false)
    hic = dht.computeHeatIndex(temperature, humidity,
        ↪  false);

    // accende se sopra soglia
    if(temperature >= tempSoglia) digitalWrite(RELAY,
        ↪ ON);

    // spegne se sotto soglia - isteresi
    if(temperature <= (tempSoglia-finestraIsteresi))
      digitalWrite(RELAY,OFF);
}
```

Bibliografia

[1] AA.VV. Enciclopedia Treccani (online), 2017. `http://www.treccani.it/enciclopedia/sensore`.

[2] AA.VV. Enciclopedia Treccani (online), 2018. `http://www.treccani.it/enciclopedia/stabilita`.

[3] Amazon. *FreeRTOS Reference Manual v.10.0.* Amazon Web Services, 2016.

[4] Arduino. Storia di Arduino, 2015. `http://playground.arduino.cc/Italiano/StoriaDiArduino`.

[5] Arduino. SPI reference, 2017. `https://www.arduino.cc/en/Reference/SPI`.

[6] ArduPilot. Sensor driver I2C, 2017. `http://ardupilot.org/dev/docs/code-overview-sensor-drivers.html`.

[7] Atmel. Atmega8, 2015. `http://www.atmel.com/devices/atmega8.aspx`.

[8] Atmel. SAM3X8e, 2015. http://www.atmel.com/devices/sam3x8e.aspx.

[9] U.A. Bakshi and V.U. Bakshi. *Control System Engineering.* Technical Publications, 2008.

[10] Richard Barry. *Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide.* FreeRTOS, 2013.

[11] R.M. Brown, P. Lawrence, and J.A. Whitson. *How to Read Electronic Circuit Diagrams*. Tab Hobby Electronics Series. TAB Books, 1987.

[12] Buildroot. Buildroot, 2015. `http://buildroot.uclibc.org/about.html`.

[13] M. Castells. *The Information Age*. Wiley-Blackwell, 1999.

[14] V. Choudhary and K. Iniewski. *MEMS: Fundamental Technology and Applications*. Devices, Circuits, and Systems. CRC Press, 2017.

[15] Giuseppe Colombo. *Manuale dell'ingegnere*. Hoepli, 1985.

[16] James O. Coplien, Douglas C. Schmidt, and John M. Vlissides. *Pattern languages of program design*, volume 58. Addison-Wesley Reading, MA, 1995.

[17] Thomas M. Cover and Joy A. Thomas. *Elements of information theory*. John Wiley & Sons, 2012.

[18] Edsger W. Dijkstra. Letters to the editor: GOTO statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.

[19] Ferrer. *Tavole di fisica*. Giunti Bemporad Marzocco, 1968.

[20] International Organization for Standardization. Iso iso11898-1:2015, 2015. `https://www.iso.org/standard/63648.html`.

[21] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[22] Herman H. Goldstine. *The computer from Pascal to Von Neumann*. Princeton University Press, 1980.

[23] Network Working Group. Nfs: Network File System protocol specification RFC1094, 1989. `https://tools.ietf.org/html/rfc1094`.

[24] Network Working Group. The tftp protocol (revision 2) rfc1350, 1992. https://tools.ietf.org/html/rfc1350.

[25] Computer history museum. Federico Faggin's short bio, 2009. http://www.computerhistory.org/fellowawards/hall/federico-faggin/.

[26] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. Automata theory, languages, and computation. *International Edition*, 24, 2006.

[27] Patrick Hurley. *A concise introduction to logic*. Nelson Education, 2014.

[28] IEEE. IEEE standard for reduced-pin and enhanced-functionality test access port and boundary-scan architecture the official IEEE 1149.7 standard, 1990. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5412866.

[29] IEEE. Ieee 802.3 ethernet working group, 2017. http://www.ieee802.org/3/.

[30] IEEE. POSIX working group, 2017. http://standards.ieee.org/develop/wg/POSIX.html.

[31] Open Source Initiative. The open source definition, 2007. https://opensource.org/osd.

[32] Texas Instruments. AM3358, 2011. http://www.ti.com/lit/ds/symlink/am3358.pdf.

[33] Texas Instruments. Am335x power consumption, 2015. http://processors.wiki.ti.com/index.php/AM335x_Power_Consumption_Summary.

[34] Texas Instruments. Am335x thermal consideration, 2015. http://processors.wiki.ti.com/index.php/AM335x_Thermal_Considerations.

[35] Intel. *ISA Bus specification and application notes*. Intel Corporation, September 1989.

[36] International Electrotechnical Commission. IEC 61131-3, 2013. https://webstore.iec.ch/publication/4552.

[37] International Organization for Standardization. ISO 13485:2003, 2003. http://www.iso.org/iso/catalogue_detail?csnumber=36786.

[38] Paul Israel. *Edison: a life of invention*. John Wiley, 1998.

[39] Jill Jonnes. *Empires of light: Edison, Tesla, Westinghouse, and the race to electrify the world*. Random House Trade Paperbacks, 2004.

[40] William Joy. An introduction to the C shell. 1986.

[41] kernel.org. Sysfs documentation, 2015. https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt.

[42] kernel.org. GPIO Linux kernel documentation, 2017. https://www.kernel.org/doc/Documentation/gpio/gpio.txt.

[43] kernel.org. GPIO sysfs Linux kernel documentation, 2017. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux-stable.git/tree/Documentation/gpio/sysfs.txt.

[44] B.W. Kernighan and D.M. Ritchie. *C Programming Language*. CreateSpace Independent Publishing Platform, 2017.

[45] Matthew Knight and Balint Seeber. Decoding lora: Realizing a modern lpwan with sdr. In *The technical proceedings for the 6th GNU Radio Conference*, 2016.

[46] Suhas Kumar. Fundamental limits to Moore's law. *arXiv preprint arXiv:1511.05956*, 2015.

[47] ST Microelectronics. 32F429IDiscovery short description, 2015. http://www.st.com/web/catalog/tools/FM116/SC959/SS1532/PF259090.

[48] Forrest M. Mims. *Getting started in electronics*. Master Publishing, 2000.

[49] Salvatore Monaco. *Enciclopedia Italiana*, chapter CONTROLLI AUTOMATICI, pages App. III, I, p. 430; IV, I, p. 523. Istituto della Enciclopedia Italiana, 1991.

[50] Leonardo Mureddu. La radio a galena, 2017. `http://www.museoradiotv.rai.it/dl/portali/site/articolo/ContentItem-70366813-4216-40c1-80d8-3e61e273f081.html`.

[51] Nerio Neri. *Radiotecnica per radioamatori*. Edizioni C&C, 2015.

[52] Carmine Noviello. *Mastering STM32*. Leanpub, 2017.

[53] James O'Neill. *Prodigal genius: the life of Nikola Tesla*. Book Tree, 2007.

[54] Parallax. *Javelin Stamp Manual*, 2005. `http://www.parallax.com/sites/default/files/downloads/JS1-IC-Javelin-Stamp-Manual-v1.1.pdf`.

[55] D.L. Perry. *VHDL: Programming by Example*. McGraw-Hill Education, 2002.

[56] Charles Petzold. *The annotated Turing: a guided tour through Alan Turing's historic paper on computability and the Turing machine*. Wiley Publishing, 2008.

[57] C. Piana, R. di Cosmo, and S. Aliprandi. *Open source, software libero e altre libertà: Unintroduzione alle libertà digitali*. Copyleft Italia. Ledizioni, 2018.

[58] Yocto Project. Yocto project, 2011. `https://www.yoctoproject.org/about`.

[59] Thomas W Scharle et al. Axiomatization of propositional calculus with Sheffer functors. *Notre Dame Journal of Formal Logic*, 6(3):209–217, 1965.

[60] NXP Semiconductor. I2S specification, 1996.

[61] N. Sornin (Semtech), M. Luis (Semtech), T. Eirich (IBM), T. Kramp (IBM), and O.Hersent (Actility). *LoRa specification*. LoRa Alliance, February 2016.

[62] R Stallman. The gnu operating system and the free software movement. open sources: Voices from the open source revolution. c. dibona, s. ockman and m. stone. calif, 1999.

[63] Guy Steele. *Common LISP: the language*. Elsevier, 1990.

[64] Andrew S. Tanenbaum. *Structured Computer Organization, 5th ed.* Pearson Prentice All, 2006.

[65] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 4th edition, 2014.

[66] A.S. Tanenbaum and D.J. Wetherall. *Computer Networks*. Pearson Education, 2012.

[67] Jeffrey J. P. Tsai, S Jennhwa Yang, and Yao-Hsiung Chang. Timing constraint Petri nets and their application to schedulability analysis of real-time system specifications. *IEEE transactions on Software Engineering*, 21(1):32–49, 1995.

[68] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 1937.

[69] U-boot. U-boot image format, 2015. `http://www.denx.de/wiki/view/DULG/UBootImages`.

[70] Jonathan W Valvano. *Embedded microcomputer systems: real time interfacing*. Cengage Learning, 2011.

[71] D. Waitzman. A standard for the transmission of IP datagrams on avian carriers. `http://tools.ietf.org/html/rfc1149`, March 1990.

[72] M Mitchell Waldrop. The chips are down for Moore's law. *Nature*, 530(7589):144–147, 2016.