ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

SISTEMI EMBEDDED E INTERNET OF THINGS

prof. Alessandro Ricci

A.A. 2020/2021

Emanuele Lamagna

INTRODUZIONE AI SISTEMI EMBEDDED

I sistemi embedded sono sistemi di elaborazione special purpose (ossia che svolgono una specifica funzione) incorporati in sistemi o dispositivi elettronici di diverse dimensioni. Il compito tipicamente richiede di interagire col mondo fisico mediante sensori e attuatori. Sono costituiti da una parte hardware e, eventualmente, una parte software. Sono diffusi nell'elettronica di consumo, nelle home appliances, nell'automazione aziendale, negli strumenti per il business, nelle automobili ecc. Un sistema embedded tipicamente esegue uno specifico programma ripetutamente: è quindi progettato per eseguire un'applicazione specifica, e questo permette a design time di minimizzare le risorse da utilizzare e massimizzare la robustezza: Hanno user interface dedicate.
Tutti i sistemi di elaborazione hanno risorse limitate, ma quelle dei sistemi embedded in particolare. Vengono usate metriche di progettazione: misura di una caratteristica dell'implementazione / realizzazione, con costo (inclusi i NRE, non-recurring-engineering, ossia costi one time), dimensioni, performance, consumo di energie. La progettazione è orientata all'efficienza. L'affidabilità dev'essere garantita, per cui si richiedono dependability, reliability, avaliability (alta probabilità di corretto e continuo funzionamento), safety and security (alta probabilità che non rechi danni agli utenti e ottemperi norme di sicurezza e privacy), reattività e real-time (spesso sono usati in contesti dove devono reagire prontamente agli stimoli dell'ambiente fisico).

I Cyber-Physical Systems (CPS) sono sistemi che integrano computazione con processi fisici. Gli aspetti specifici rispetto ai sistemi di pura elaborazione delle informazioni sono:

- gestione del tempo
- concorrenza
- reattività
- eventi asincroni

Le 3 parti del CPS sono:

- parte fisica (physical)
- parte computazionale/embedded (cyber)
- parte di rete (cyber)

L'industria 4.0 è un trend corrente in ambito di automazione nell'industria manifatturiera, che include GPS, IOT e cloud computing. Chiamato anche "smart factory", poiché ci sono fabbriche digitalizzate e interconnesse:

- i CPS monitorano i processi fisici
- sfruttando IOT, i CPS comunicano tra di loro e con operatori umani in real-time
- sfruttando cloud e Internet dei servizi, vengono resi disponibili servizi sia all'interno dell'organizzazione, sia all'esterno

L'accezione "physical computing" è usata per indicare sempre sistemi computazionali che interagiscono col mondo fisico, tipicamente orientati in questo caso alla costruzione ed esplorazione di framework per interazione uomo-macchina. I sistemi "wearable computing" sono sistemi embedded indossabili, e tecnologie abilitanti per sistemi hands-free o di realtà aumentata e mondi aumentati.
L'architettura di un sistema embedded si compone di hardware, SO (opzionale) e software applicativo (opzionale).

L’architettura

Per i sistemi embedded esistono sostanzialmente 3 tipi di tecnologie di processori:

- processori general-purpose, che hanno un’architettura e un insieme di istruzioni predefinito e lo specifico comportamento è definito dal software in esecuzione
- processori single-purpose, con circuiti digitali progettati per implementare la specifica funzionalità o programma (ASIC)
- application-specific processors (ASiP), una via intermedia, ossia processori programmabili, ottimizzati per una specifica classe di applicazioni aventi caratteristiche comuni (microcontrollori, SoC, DSP), e anche in questo caso la logica è implementata via software.

Esistono diversi tipi di implementazioni dei single-purpose:

- full-custom/VLSI, dive si progetta l’intero processore in modo custom
- semi-custom si parte da un insieme di livelli costruiti, implementandovi sopra il processore
- PLD, tutti i livelli sono già presenti, quindi si può acquistare un IC e programmarlo
- PLA, array programmabili di porte AND e OR
- PAL, solo un tipo di array
- FPGA, circuiti integrati con funzionalità programmabili via software, che permettono connettività/programmabilità più a livello generale rispetto a PLA e PAL.

Microcontrollori

I microcontrollori sono dispositivi elettronici nati come evoluzione alternativa ai microprocessori integrando su singolo chip un sistema di componenti che permette di avere la massima autosufficienza funzionale per applicazioni embedded. Possiedono processore, memoria permanente, memoria volatile, canali (pin) di I/O, gestore interrupt ed eventuali altri blocchi specializzati. Sono generalmente dotati di CPU CISC con architettura di Von Neumann, ma di recente ne sono apparsi con architettura RISC. Il mercato è molto frammentato, con molti produttori e molte architetture, nessuna dominante. I microcontrollori single-board incorporano in un’unica scheda il microcontrollore e tutta la circuiteria necessaria per eseguire i compiti di controllo. Sono presenti microprocessore, I/O, generatore di clock, RAM, memoria per contenere il programma, ecc. La scheda, in questo modo, può essere immediatamente utilizzabile dallo sviluppatore di applicazioni. Un esempio è Arduino: il modello Uno ha CPU a 8 bit a 16 MHz, memoria istruzioni di 32 KB e memoria dati di 2 KB volatile + 1 KB non volatile.

SoC e single-board CPU

SoC significa system-on-a-chip, e in questo caso è un chip stesso che incorpora un sistema completo, che include CPI, memoria, controllori ecc. Sono tipicamente usati per creare single-board CPU. Un esempio è Broadcom, usato nella famiglia Raspberry.
ESP8266 è un dispositivo pensato per IOT. Ha un processore RISC a 32 bit a 80 MHz, 64 KB di RAM per le istruzioni, 96 KB di RAM per i dati, 16 pin GPIO e il protocollo Wi-Fi.
La scelta di MCU e SoC dipende dai requisiti del progetto ed è sempre un trade-off. Al giorno d’oggi vengono sempre più usati, poiché portano benefici a tutto il processo di progettazione e sviluppo (facilitano lo sviluppo di famiglie di prodotti che forniscono diverse funzionalità, e inoltre l’uso di processori con set predefinito di istruzioni spesso conduce ad un’implementazione più rapida).

Sensori e attuatori

Un sistema embedded interagisce con l'ambiente in cui è situato mediante sensori e attuatori. I sensori sono dispositivi trasduttori che permettono di misurare un certo fenomeno fisico, fornendo una rappresentazione in scala o intervallo. Possono essere:

- analogici: la grandezza elettrica prodotta in uscita, tensione o corrente, varia con continuità in risposta alla grandezza misurata, e di solito nel microcontrollore è incluso un convertitore analogico-digitale ADC
- digitali: con due soli valori, oppure un insieme discreto di valori

Gli attuatori sono dispositivi che producono un qualche effetto misurabile sull'ambiente, a partire da una specifica condizione o evento chiamato trigger.

Bus e comunicazione

L'interazione fra MCU/microprocessore e sensori/attuatori avviene mediante canali di comunicazione (bus) con specifici protocolli. Nel caso dei sistemi embedded i protocolli più usati implementano una trasmissione seriale (parole multi-bit inviate come serie di bit sequenziali). Esempi sono I^2C, SPI, JTAG.
Uart (Universal Asynchronous Receiver Transmitter) è il più datato dei protocolli seriali, supportato da qualsiasi MCU. Consente di convertire flussi di bit di dati da un formato parallelo a un formato seriale asincrono o viceversa. Di per sé Uart non genera o riceve direttamente i segnali da convertire: questo generalmente viene fatto da dispositivi di interfacciamento separati come RS-232, Bluetooth, USB..
USART (Synchronous/Asynchronous) estende il protocollo con la trasmissione di un segnale di clock per la sincronizzazione.
I^2C (Inter-Integrated Circuit) è un protocollo/bus seriale sincrono, a due fili (uno per dati e uno per clock), è semplice da usare e più espandibile.
SPI (Serial Peripheral Interface) è un protocollo/bus seriale sincrono, basato su master e slave. È più veloce di I^2C, ma più difficile da usare.
JTAG è il protocollo standard per il test funzionale di dispositivi, da usare in accoppiata con strumenti come il debugger.
CAN-bus (Controller Area Network) è un protocollo standard seriale per bus usati nell'ambito automotive e in contesti industriali. È pensato per collegare varie unità di controllo e progettato per funzionare in ambienti disturbati dalla presenza di onde elettromagnetiche. È basato sullo scambio di messaggi in multicast.
Ci sono tecnologie e standard che permettono la comunicazione wired e wireless con altri sistemi (embedded e non). Quelli wireless, in particolare, sono Bluetooth, Wi-Fi, ZigBee, ecc.

Progettazione di sistemi embedded

Il processo iterativo è modelling, design e analisi:

- il modelling è un processo finalizzato ad ottenere un'approfondita comprensione o conoscenza relativamente al sistema da costruire. Tale conoscenza è rappresentata da modelli, che sono il risultato di questo processo. Rappresentano cosa il sistema deve fare. La modellazione può riguardare la parte fisica o la parte logica, oppure un ibrido. Nel design si sceglie la tecnologia e l'architettura hardware, e poi si sceglie l'architettura hardware, e poi si sceglie l'architettura software più appropriata.

- il design è un processo finalizzato alla creazione degli artefatti tecnologici che rappresentano il sistema: rappresentano come il sistema fa quello che deve fare. La modellazione piò avere un ruolo importante anche in questa fase, fornendo una descrizione astratta di come funziona il sistema, che prescinde dalla specifica implementazione. Il design può essere top-down o bottom-up, attraverso livelli diversi:
    - i requisiti, che consistono nella raccolta della descrizione informale da parte del customer circa il sistema e cosa si supponga che faccia (requisiti funzionali e non funzionali) e nell'uso di requirement forms
    - le specifiche, ossia una descrizione più precisa e rigorosa dei requisiti, che dovrebbe essere comprensibile per una verifica del soddisfacimento dei requisiti da parte del cliente (si utilizza il linguaggio UML)
    - la progettazione dell'architettura descrive come il sistema implementa le funzioni descritte nelle specifiche e si divide in architettura (piano della struttura complessiva del sistema che descrive quali componenti servono e come essi interagiscono) e diagrammi a blocchi (hardware e software)
    - progettazione dei componenti, sia della parte software che hardware, alcuni componenti possono essere ready-made (quindi già disponibili), quali CPU, GPS receivers, db topografici ecc.
    - integrazione di sistema, ossia dei componenti seguendo l'architettura
- l'analisi è un processo finalizzato ad ottenere un'approfondita comprensione e conoscenza del comportamento del sistema. Specifica perché un sistema fa quello che fa

In sintesi, sviluppo e programmazione di sistemi embedded sono manuali o model-driven. Avvengono su computer host, con linguaggi/piattaforme di alto livello (sui microcontrollori sono frequenti C/C++, su quelli con SO sono più frequenti linguaggi di alto livello.
Modellazione e progettazione sono molto importanti, e bisogna mantenere il livello di astrazione individuato nella modellazione (ci sono vari vantaggi, tra cui riusabilità, comprensibilità e manutenibilità).

SISTEMI EMBEDDED BASATI SU MICROCONTROLLORE

Architettura: Arduino Uno

Arduino Uno ha un microcontrollore MCU ATMega328P, della famiglia AVR ATMega, a 8 bit. È un RISC con 32 registri, ha una frequenza di 16 MHz (16 milioni di cicli di clock al secondo). Per le istruzioni ha una memoria FLASH da 32 KB, mentre per i dati una SRAM da 2 KB e una EEPROM da 1 KB. Ci sono 14 pin digitali di I/O, di cui 6 possono essere usati come uscite PWM, e 6 input analogici. La memoria è a 16 bit. I pin digitali da 0 a 13 operano ad una tensione di 5V e possono fornire o assorbire fino a 40mA si corrente. Gli ingressi analogici da A0 ad A5 operano sempre a 5V e hanno una risoluzione a 10 bit: valori di tensione da 0 a 5V vengono convertiti mediante un convertitore analogico-digitale (ADC) in valori a 10 bit, da 0 a 1023 (ossia 2^{10}-1). Col pin 21 (AREF) si può cambiare il valore di riferimento per l'ADC. Tipicamente la programmazione di un microcontrollore avviene utilizzando un sistema esterno (es. un PC), dove i programmi vengono editati, compilati e viene creato l'eseguibile in codice binario. L'eseguibile viene quindi trasferito sul microcontrollore mediante opportuni dispositivi HW, oppure direttamente mediante collegamento (seriale) col PC host. Lo stack SW su microcontrollori non include alcun sistema operativo (il codice eseguibile viene trasferito in memoria e direttamente eseguito dal processore).
Il trasferimento mediante USB viene effettuato ad opera di un bootloader, un piccolo programma nella memoria FLASH di circa 0.5 KB, precaricato via ICSP (In-Circuit Serial Programmer), ossia un'interfaccia che si può usare al posto del PC e collegamento USB: va abbinata ad un apposito dispositivo "programmer" interno.
L'ambiente di programmazione è composto da:

- Wiring: è un framework open source di C/C++ disponibile per diversi tipi di microcontrollore
- Arduino IDE: è un semplice tool di editing, compilazione e trasferimento dell'eseguibile sul microcontrollore.

CPU e unità di memoria

Il funzionamento di una CPU è definito a livello di modello della macchina di Von Neumann (ciclo di esecuzione fetch-decode-execute). L'instruction-execution cycle di un computer in quanto macchina di Von Neumann consiste nel:

- caricamento dalla memoria (fetch) dell'istruzione da eseguire, depositata in un apposito registro (instruction register)
- decodifica dell'istruzione (decode), che può comportare il caricamento di altri operandi dalla memoria
- esecuzione dell'istruzione (execute), che può portare all'aggiornamento dei registri e alla scrittura di dati in memoria.

L'architettura Harvard è una variante dell'architettura di Von Neumann in cui istruzioni e dati sono memorizzati in memorie fisicamente separate, con bus distinti. È l'architettura tipicamente usata nei microcontrollori e DSP (processori dedicati all'elaborazione di segnali digitali). le due memorie utilizzate possono avere caratteristiche molto diverse, ad esempio:

- codice in memoria FLASH: si ha un accesso veloce in lettura, ma lento in scrittura
- dati in SRAM: si ha un accesso veloce sia in lettura che in scrittura, ma consuma di più.

Codice e dati possono essere scritti parallelamente, dato che usano bus distinti. Quindi le istruzioni sono sulla FLASH (non volatile) da 32 KB, di cui 0.5 occupati dal bootloader, mentre i dati sono in

parte in SRAM (volatile, da 2 KB, ci sono le variabili) e in parte in EEPROM (non volatile, da 1KB, ci sono i dati persistenti). Quindi possiamo intuire che la memoria utilizzabile per dati/variabili è esigua, il che ci porta a minimizzare l'uso di stringhe e strutture dati. Lo stack è sulla SRAM, quindi anche il nesting di chiamate (es. chiamate ricorsive) dev'essere usato con attenzione.
Confronto fra memorie:

- FLASH è non volatile, veloce in lettura e lenta in scrittura, ha bassi costi e consumi
- SRAM è volatile, veloce in scrittura e lettura, consuma di più della FLASH
- EEPROM è non volatile, accesso meno performante, può essere letta e scritta con una libreria di Arduino

La CPU esegue le istruzioni del programma codificate in linguaggio macchina. È caratterizzata da un proprio Instruction Set Architecture (ISA): esso è composto dall'insieme di possibili istruzioni riconosciute dalla CPU e dall'insieme di registri, che possono essere generali/programmabili (GPR) o dedicati/special purpose (SFR) come il Program Counter (PC) dove risiede l'indirizzo della prossima istruzione da eseguire o il Program Status Word (PS) che ha informazioni sullo stato del processore.
Il microprocessore presente nei microcontrollori ATMega328P è un AVR8, con architettura RISC a 16 bit dati, 8 bit codice a 16 MHz. Ci sono 32 registri general purpose a 8 bit (R0-R31): tutte le operazioni aritmetiche e logiche avvengono usando questi registri, solo le operazioni di load e store accedono alla memoria RAM. C'è invece un insieme ristretto di registri dedicati, che includono:

- PC (Program Counter, a 16 o 22 bit)
- SP (Stack Pointer, a 8 o 16 bit)
- SREG (a 8 bit)

Sia i registri general purpose sia quelli relativi all'I/O sono mappati in memoria, accessibili nei primi indirizzi dello spazio di indirizzamento.
Uno sguardo alle performance: la frequenza del clock supportato dal microprocessore è di 16 MHz, il periodo è quindi 62.5 nanosecondi. La maggior parte delle istruzioni del processore vengono eseguite in 1 o 2 cicli di clock, ovvero tra i 60 e i 125 ns. Queste informazioni permettono di prevedere in modo piuttosto accurato il tempo di esecuzione di programmi o di porzioni di essi, in particolare del caso peggiore (in programmi non deterministici): è un aspetto importante per lo sviluppo di sistemi real-time.
L'architettura di controllo più semplice che si può adottare nella programmazione di microcontrollori è il super-loop. È molto diffusa poiché non richiede supporti HW specifici (es. interruzioni). Le sue caratteristiche sono inizializzazione (setup in Wiring) e ciclo infinito (loop in Wiring) che esegue ripetutamente un certo task.
I pro del super-loop sono:

- semplicità (nessun sistema operativo)
- portabilità
- affidabilità e sicurezza
- efficienza

I contro sono:

- temporizzazioni non accurate
- fragilità e scarsa flessibilità
- versione base consuma molto (funzionamento full-power)

Ogni volta che Arduino viene resettato, entra in esecuzione il bootloader per qualche secondo. Se esso riceve un apposito comando dall'IDE, allora cerca di caricare il programma che gli invia l'IDE, se non riceve nessun programma o comando, allora parte eseguendo l'ultimo sketch caricato in memoria. Anche qui abbiamo heap e stack che crescono in direzioni opposte e possono collidere in caso di memory overflow.

Possono essere rilevati dei problemi:

- polling: è l'approccio utilizzato per rilevare la pressione del pulsante, ossia con una lettura periodica del valore. Il problema è che se il tasto viene premuto e rilasciato prima che possa essere letto, l'evento di pressione non viene rilevato.
- corse critiche: nel loop è necessario eseguire l'assegnamento della variabile in sezione critica, per evitare corse critiche dovute all'eventuale accesso concorrente da parte dell'interrupt handler. La realizzazione di sezioni critiche su microcontrollori avviene mediante la disabilitazione delle interruzioni, e in particolare in Wiring si utilizzano le funzioni noInterrupts() e interrupts(). Con le interruzioni disabilitate il microcontrollore non reagisce e quindi c'è una perdita di eventi, ecco perché il periodo in cui vengono disabilitate dev'essere il più breve possibile.
- bouncing: quando i pulsanti vengono premuti, internamente, a livello meccanico, possono "rimbalzare" prima di stabilizzarsi (entro qualche millisecondo) al valore aperto o chiuso. Questi rimbalzi possono quindi generare un treno di interruzioni (poiché il valore passa da LOW a HIGH più volte) e, se per esempio stiamo utilizzando un contatore, verrà incrementato più volte ad una sola pressione. Le soluzioni sono hardware (con pulsanti senza bouncing) o software (ignorando eventuali impulsi che arrivano nell'arco di un certo numero di millisecondi dopo aver rilevato il primo).

GPIO e convertitori AD

I microcontrollori utilizzati in ambito embedded contengono usualmente un certo insieme di pin che possono essere usati per gestire input e output. I pin tipicamente sono general-purpose, nel senso che possono essere programmati per fungere da input o output seconda delle necessità. In input il valore può essere letto via software, in output può essere pilotato via software. I pin possono essere:

- digitali: assumono solo due valori (HIGH o LOW, 1 o 0)
- analogici: possono assumere un qualsiasi valore all'interno di un certo range

In Arduino ci sono 14 pin digitali (configurabili sia come input che come output) e 6 pin analogici (solo di input). Alcuni pin hanno più funzioni (multiplexed), come i pin 0 e 1 che sono usati come porta seriale.
I parametri elettrici di funzionamento per i pin sono:

- tensione: in Volt, da applicare (nel caso di input) o prodotta in uscita (nel caso di output), in Arduino sono 5V, in Raspberry Pi ed ESP sono 3.3V
- corrente: in Ampere, che può ricevere (input) o che può fornire (output), in Arduino sono 40 mA, in ESP sono 12 mA

Internamente i pin possono essere equipaggiati di circuiti pull-up, per fissare il valore di tensione (a +VCC) anche quando il circuito attaccato al pin è aperto ed evitare malfunzionamenti dovuti al floating del segnali (si usano resistori di decine di KOhm.

Le porte sono ciò che permette al microcontrollore di interagire con i pin, e quindi coi dispositivi esterni, e come per i pin anch'esse sono di input o output o entrambi. Sono costituite da uno o più registri special purpose (SRF) connessi ai pin che trasportano i segnali digitali o analogici inviati o ricevuti dai dispositivi esterni: il registro mantiene lo stato dei vari pin; ad ogni porta è tipicamente associato anche un registro special purpose i cui bit determinano la configurazione/stato dei corrispondenti pin (se input, output, ecc.). Per esempio, l'ATMega328P ha 23 linee di I/O raggruppate in 3 porte da 8 bit l'una denominate B, C e D. La porta D contiene i pin da 0 a 7, la porta B i pin da 8 a 13, la porta C quelli analogici. Ogni pin è identificato da una sigla come PD0, PC1 o PB2: ad esempio PD0 identifica il pin 0 della porta D. Ogni porta è gestita da 3 registri:

- DDRx
- PORTx
- PINDx

dove x è il nome della porta. Ad esempio nel caso della porta D:

- DDRD: registro di lettura/scrittura, contiene la direzione dei pin ad esso collegati (bit a 0 INPUT, bit a 1 OUTPUT)
- PORTD: registro di lettura/scrittura, contiene lo stato dei pin, che cambia a seconda della selezione dei pin:
    - se il pin è impostato come INPUT, un bit a 1 attiva la resistenza di PULL-UP, mentre un bit a 0 la disattiva
    - se il pin è impostato come OUTPUT, un bit a 1 indica uno stato HIGH sul pin, mentre un bit a 0 indica la presenza dello stato LOW
- PIND: registro di sola lettura, contiene, col pin impostato come INPUT, la lettura del segnale collegato al pin: 1 per HIGH, 0 per LOW

Quindi per impostare un pin come OUTPUT con un livello HIGH basta impostare la direzione col registro DDRx e il suo stato col registro PORTx. In ATMega328, come in molti microcontrollori e anche microprocessori general-purpose, i registri di I/O sono mappati in memoria, per cui si accedono/manipolano in modo uniforme mediante scritture e letture di byte/word ad indirizzi di memoria ben definiti. Nell'ATMega328P sono accessibili a partire dall'indirizzo 0x20, ovvero subito dopo i tre registri general-purpose.
Su Wiring sono fornite API di base per queste operazioni:

- pinMode()
- digitalWrite()
- digitalRead()

Queste API accedono ai registri delle porte visti in precedenza.
La procedura delay() è disponibile in Wiring ed esegue un busy waiting per la quantità di millisecondi specificata. La procedura fa uso di un'ulteriore procedura millis(), che restituisce il numero corrente di millisecondi trascorsi dall'accensione del sistema (accede a variabili del timer). Il busy waiting rende di fatto non reattivo il loop di controllo e altri possibili, e quindi in generale è da evitare.
Oltre ad essere general-purpose, alcuni pin hanno un'ulteriore funzione specifica:

- pin 0 e 1: interfaccia seriale TTL
- pin 2 e 3: interruzioni
- pin 3, 5, 6, 9, 10, 11: PWM
- pin 10, 11, 12, 13: comunicazione SPI
- pin 13: builtin led
- pin A4 e A5: I^2C

Il Pulse-With-Modulation (PWM) è una tecnica di pilotaggio di dispositivi di output che permette di emulare in uscita un segnale analogico a partire dalla generazione di segnali digitali detti PWM. il segnale analogico di un certo valore V su un pin è "emulato" mediante un segnale periodico che si ottiene emulando il duty cycle di un'onda quadra, ovvero un segnale che passa ripetutamente da 0 a 1 (per duty cycle si intende la percentuale di tempo che il segnale è a 1 rispetto a quella in cui è a 0). Sul pin si ottiene in tal modo un segnale equivalente analogico il cui valore (tensione) risultante medio dipende dal duty cycle: ad esempio avendo un duty cycle pari al 50% otteniamo una tensione media pari alla metà del valore massimo (2.5V), con duty cycle pari all'80% otteniamo 4V. Non funziona per pilotare qualsiasi dispositivo analogico di output: funziona bene ad esempio per pilotare led con intensità variabile, motori in continua (per variare la velocità).

Su Arduino i pin 3, 5, 6, 9, 10 e 11 possono essere utilizzati per output PWM a 8 bit tramite la funzione analogWrite().
Un segnale digitale su un pin può assumere nel tempo solo due valori (HIGH o LOW), mentre un segnale analogico su un pin può invece assumere un valore continuo all'interno di un certo range (es. 0-5V). Quindi per poter elaborare un segnale analogico da un sistema di elaborazione (che lavora solo con valori digitali), questo dev'essere convertito. La conversione avviene mediante un componente detto convertitore ADC (Analog-to-Digital), che mappa il valore continuo in un valore discreto in un certo range (es. 0...1023). Il numero di bit utilizzati per codificare il valore discreto rappresenta la risoluzione del convertitore (es. 10 bit) e ne determina la precisione. Arduino UNO può gestire fino a 6 segnali di input analogici (pin A0-A5). Per ogni input è presente un ADC con risoluzione a 10 bit (ogni segnale analogico viene convertito in un valore a 10 bit, quindi da 0 a 1023, poiché 2^{10} = 1024 valori). La funzione analogRead(), internamente, legge i registri della porta C. È fornita anche un'ulteriore funzione, map(), che permette di mappare un valore in un certo range specificato.

Interruzioni e timer

Il meccanismo delle interruzioni permette al microcontrollore (o meglio, al programma in esecuzione) di reagire ad eventi, evitando di fare polling: possono riguardare i dispositivi di I/O e il timer. In generale le CPU mettono a disposizione uno o più pin (chiamati IRQ, Interrupted Request) dove ricevere i segnali di interruzione. Quando riceve una richiesta di interruzione, sospende l'esecuzione della sequenza di interruzioni, salva sullo stack l'indirizzo della prossima istruzione che avrebbe eseguito e quindi trasferisce il controllo all'interrupt routine corrispondente, chiamata interrupt handler o interrupt service routine (ISR). L'indirizzo della ISR è memorizzato nella tabella o vettore delle interruzioni. Nei microcontrollori gli IRQ sono tipicamente connessi ad uno o più GPIO.
In ATMega328P due pin (il 2 e il 3) possono essere configurati per generare interruzioni esterne. In generale può essere generata un'interruzione su un valore LOW, su un RISING o FALLING edge o su un cambiamento di valore. In Wiring è possibile implementare una ISR a livello di user program, mediante la routing attachInterrupt(intNum, ISR, mode), dove intNum è l'intero che identifica l'interruzione, ISR è il puntatore all'interrupt handler (la funzione da eseguire) e mode è la situazione per la quale viene generata l'interruzione (CHANGE/FALLING/RISING rispettivamente ad ogni cambiamento, 1->0, 0->1, oppure LOW/HIGH quando è 0 o 1). È disponibile la funzione digitalPinToInterrupt(numPin) che recupera il numero dell'interruzione associata al pin specificato.
Quando disabilitiamo le interruzioni il sistema non è più reattivo ad eventi esterni, il che pone alcuni vincoli sul design degli interrupt handler:

- devono essere eseguiti in tempi brevi
- non possono bloccarsi o eseguire loop infiniti
- si deve avere una bassa interrupt latency, ossia il sistema deve impiegare poco tempo a reagire ad un'interruzione

FAQ SULLE INTERRUZIONI:

- Può un processore essere interrotto durante l'esecuzione di un'istruzione? No, tipicamente le istruzioni in LM vengono eseguite atomicamente, a meno di non considerare quelle istruzioni che permettono di operare trasferimenti molteplici di valori in memoria.
- Se due interruzioni avvengono contemporaneamente, quale viene servita? Ogni processore assegna delle priorità alle interruzioni, per cui viene eseguita l'interruzione più prioritaria.

- Può una richiesta di interruzione interrompere un'altra interruzione? Nella maggior parte dei processori si, supportando la "interrupt nesting". Questo si può evitare disabilitando le interruzioni all'inizio della routine e riabilitandole alla fine.
- Cosa succede ad una segnalazione di interruzione se avviene quando le interruzioni sono disabilitate? Tipicamente il processore tiene traccia di tali richieste e le esegue non appena vengono riabilitate.
- Cosa succede se non riabilito le interruzioni? Malfunzionamento del sistema, crash.
- Cosa succede se disabilitiamo le interruzioni più volte di seguito o le riabilitiamo più volte? Nulla, sono idempotenti.
- Quando un processore parte (boot) le interruzioni sono abilitate o disabilitate? Disabilitate.

Su AVR/Arduino l'interrupt handler viene eseguito con le interruzioni disabilitate. Non tutte le primitive di libreria funzionano se chiamate all'interno di un interrupt handler: in particolare non funzionano quelle il cui funzionamento si basa direttamente o indirettamente sull'uso di interruzioni, come ad esempio delay() o millis(). Invece delayMicroseconds() funziona, poiché non si basa sull'uso di timer, ma sull'esecuzione di istruzioni che hanno una specifica durata in clock cycle.

Il timer svolge un ruolo fondamentale per la realizzazione di comportamenti time-oriented, in cui il tempo diventa aspetto importante dell'elaborazione e dei compiti del microcontrollore:

- misurare intervalli di tempo
- segnali PWM
- realizzazione di timeout e allarmi
- realizzazione di forme di multitasking basato su preemptive scheduling

Ad alto livello può essere rappresentato con un contatore che viene incrementato a livello HW ad una certa frequenza. A livello programmatico, è tipicamente possibile configurare la frequenza, accedere e leggere il valore corrente del contatore, agganciare interruzioni per implementare computazioni event-driven (time triggered).
ATMega328 ha 3 timer interni: timer0, timer1 e timer 2. Ognuno di essi ha un contatore, che è incrementato ad ogni tick del clock del timer. Timer0 e timer2 hanno contatori a 8 bit, mentre timer1 a 16 bit. Il timer viene sfruttato per implementare varie funzioni, come millis() e analogWrite().
In ATMega328 esistono più modalità di gestione delle interruzioni relative ai timer: una di queste si chiama CTC (Clear Timer on Compare Match). In modalità CTC le interruzioni del timer sono generate quando il contatore raggiunge un certo specifico valore (memorizzato in un registro chiamato "compare match register". Quando il contatore del timer raggiunge questo valore, resetta il conteggio al clock successivo e quindi riparte a contare. Scegliendo il valore da mettere nel registro compare-match si specifica la frequenza con cui devono avvenire le interruzioni (sapendo che il contatore del timer è aggiornato a 16 MHz). I contatori sono incrementati a 16 MHz, quindi ogni tick avviene ogni 63 ns. I timer 0 e 2 sono a 8 bit, quindi impiegano 16.123 ns ad andare in overflow (ripartire da 0), mentre il timer 1 impiega 4.1 ms. Questa frequenza può essere modulata specificando un valore detto prescaler, che funge da divisore della frequenza originaria con cui viene incrementato il timer (ovvero 16 MHz):

timer speed (Hz) = (Arduino clock speed (16 MHz)) / prescaler

Quindi specificando un prescaler di 1 -> incrementa il contatore a 16 MHz, un valore di 8 lo incrementa a 2 MHz, ecc. I valori di prescaler possono essere 1, 8, 64, 256 e 1024.
Considerando il prescaler, allora la frequenza con cui vengono generate le interruzioni è data dalla frequenza di incremento diviso il valore specificato nel compare match register:

desired interrupt frq (Hz) = 16.000.000 Hz / (prescaler(compare match register + 1))*

C'è il +1 perché il valore 0 rappresenta il primo valore significativo del registro. Esprimendo l'equazione rispetto al compare match register si ha:

*CMR = (16.000.000 Hz / (prescaler*desired interrupt frq)) - 1*

Dobbiamo controllare che questo valore sia inferiore a 256 se usiamo timer 0 o 2, inferiore a 65.526 se usiamo timer 1. Se non lo è dobbiamo aumentare il prescaler. Ad esempio, supponiamo di volere un'interruzione al secondo, ovvero alla frequenza di 1 Hz. Allora :

*CMR = (16.000.000 / (prescaler*1)) - 1*

Se usiamo un prescaler da 1024 otteniamo:

*CMR = (16.000.000 / (1024*1)) - 1 = 15.624*

Siccome il valore è maggiore di 256 ma inferiore a 65536, possiamo usare solo il timer 1.

I registri dei timer che memorizzano la configurazione sono due: TCCRxA e TCCRxB (dove x è il numero del timer 1..3). Sono a 8 bit, e i bit più importanti sono i 3 bit meno significativi di TCCRxB denominati CSx2, CSx1 e CSx0, che determinano la configurazione del clock del timer. La modalità CTC si abilita attivando uno specifico bit in TCCRxB. Il compare match register è etichettato con OCRxA.

In Arduino possiamo usare la macro ISR per dichiarare il corpo dell'handler di interruzione relative ad interrupt predefiniti: dentro di essa le variabili che andiamo a modificare devono essere volatile. Esistono varie librerie in Arduino per gestire il timer a più alto livello, come ad esempio TimerOne. Nell'implementazione della libreria che gestisce i GPIO in Wiring su Arduino, i timer vengono utilizzati per realizzare le uscite PWM. In particolare Timer0 è usato per i pin 5 e 6, Timer1 per i pin 9 e 10 e Timer2 per i pin 11 e 3.

Il watch dog timer (WDT) contiene un timer in grado di eseguire un conteggio fino ad un certo valore, dopo il quale genera un segnale di output con cui resetta il circuito. Nel funzionamento normale il WDT riceve (periodicamente) un segnale prima che arrivi alla soglia, con cui resetta il conteggio. Se non riceve il segnale in tempo, allora significa che il microprocessore è entrato in una situazione critica (es. blocco) e resetta il circuito. È un componente molto diffuso nei sistemi embedded.

I timer e le interruzioni sono sfruttati per realizzare molte delle funzionalità messe a disposizione da Wiring (es. uscite PWM). Possono esserci quindi dei malfunzionamenti se si utilizzano in maniera concordante delle funzionalità che sfruttano, ad esempio, il medesimo timer. Per evitare questi problemi è necessario prendere visione nel dettaglio della documentazione relativa alle varie API usate.

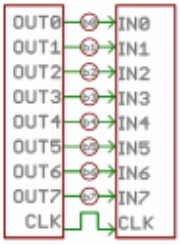
Una funzionalità molto importante supportata in genere dai microcontrollori è la possibilità di funzionare in modalità che comportano livelli diversi di risparmio energetico (sleep mode), con l'istruzione SLEEP. In ATMega328P ci sono 5 modalità di sleep (con risparmio crescente):

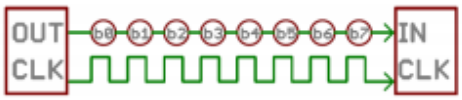
- Idle Mode
- ADC Noise Reduction Mode
- Power-save Mode
- Standby Mode
- Power-down Mode

Bus e comunicazione seriale

I protocolli di scambio di informazioni mediante singoli pin possono diventare molto complessi a seconda dei dispositivi di I/O con cui interfacciarsi. Allo scopo sono stati introdotti nel tempo varie interfacce e protocolli che possono essere classificate in seriali e paralleli. Le interfacce parallele permettono di trasferire più bit allo stesso tempo: richiedono bus a 8, 16 o più fili, mentre le interfacce seriali inviano i dati in uno stream sequenziale bit per bit, utilizzando un solo filo o comunque un numero esiguo di fili.

I bus possono essere:





- asincroni: non usano clock per la sincronizzazione, usano essenzialmente due linee (trasmissione e ricezione), alcuni esempi sono USB, RS-232 e TTL
- sincroni: per la sincronizzazione fra le parti che comunicano usano una linea con un segnale di clock, che si affianca alla linea dove vengono trasmessi i dati. Questa permette di avere trasferimenti più rapidi, al prezzo di una maggiore complessità. Esempi di bus sincroni sono I^2C e SPI.

Nelle interfacce seriali asincrone, dal momento che non c'è il supporto di un segnale di clock esterno a sincronizzare la comunicazione, dev'essere stabilito un opportuno protocollo per assicurare che il trasferimento dei dati avvenga in modo robusto e senza errori. Non c'è un solo modo, il protocollo è variamente configurabile. Questione cruciale è che le parti che comunicano adottino lo stesso protocollo. Vari aspetti da stabilire sono:

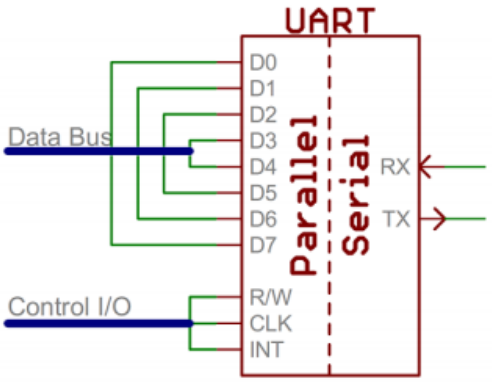
- baud rate: specifica quanto velocemente i dati sono inviati sulla linea seriale, ed è usualmente espresso in bits-per-second (bps). Un tipico baud rate per la comunicazione seriale è 9600. Più è alto, più velocemente i dati sono inviati/ricevuti. C'è un limite alla velocità, ossia tipicamente 115.200, legata alla capacità dei microcontrollori. Con valori più elevati è possibile andare incontro a errori di trasmissione.
- data frame: ogni blocco dati (tipicamente un byte) è inviato in forma di pacchetto (packet o frame) di bit. È quindi detto data frame. Questi frame sono creati aggiungendo un bit di sincronizzazione e bit di parità ai dati inviati. Il data chunk rappresenta le vere e proprie informazioni trasmesse. In ogni pacchetto si riescono ad inviare dai 5 ai 9 bit. Un valore tipico è 8 (data la lunghezza dei byte). Occorre specificare e accordarsi sull'endianess dei dati, ovvero l'ordine con cui sono inviati i bit (dal più significativo al meno, e viceversa), ma se non specificato sono inviati a partire da quelli meno significativi.
- bit di sincronizzazione: includono bit di start e stop, per segnare inizio e fine pacchetto (c'è sempre solo 1 bit di start, i bit di stop possono essere anche 2).
- bit di parità: opzionali, sono usati a volte per avere una forma low-level di rilevazione di errori

Un esempio concreto: il 9600 8N1, ossia 9600 baud, 8 data bits, no parity, 1 stop bit, è uno dei più usati protocolli seriali. Consideriamo l'esempio in cui vogliamo trasmettere i caratteri ASCII 'O' e 'K', relativamente 01001111 e 01001011 in notazione binaria. I dati sono trasferiti a partire dal bit meno significativo:

Siccome stiamo trasferendo a 9600 bps, il tempo speso per mantenere ognuno di questi bit ad un valore alto o basso è 1/(9600 bps), ovvero 104 µs per bit. Per ogni byte inviato ci sono 10 bit inviati: uno start bit, 8 data bits e uno stop bit. Quindi a 9600 bps stiamo inviando 960 (9600/10(byte al secondo.

A livello HW un bus seriale consiste di due soli fili: uno per inviare bit e uno per riceverli. Per questo motivo, i dispositivi seriali necessitano di 2 pin (un ricevitore, RX, e un trasmettitore, TX). Il nome RX e TX è da intendersi rispetto al dispositivo stesso. Quindi il pin RX di un dispositivo dovrebbe essere collegato a quello TX dell'altro dispositivo, e viceversa (il trasmettitore deve parlare con il ricevitore e viceversa). Un componente fondamentale del sistema seriale è l'UART (Universal Asynchronous

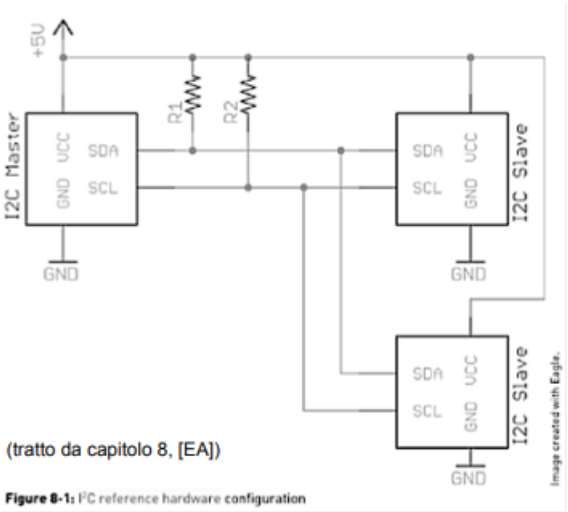
Receiver/Transmitter), ovvero il blocco circuitale che ha il compito di convertire i dati verso e dall'interfaccia seriale. Agisce da intermediario fra l'interfaccia parallela e quella seriale: da una parte di un UART c'è un bus a 8 o più linee dati (più qualche control pin), dall'altro ci sono due linee seriali, RX e TX. Gli UART esistono come circuiti integrati stand-alone, tuttavia sono più comunemente inclusi dentro ai microcontrollori. Ad esempio Arduino UNO, che si basa su ATMega328, ha un singolo UART, mentre Arduino Mega, che si basa su ATMega2560, ne ha 4.



Un'interfaccia seriale dove entrambi i dispositivi possono inviare e ricevere dati può essere full-duplex (se entrambi i dispositivi possono inviare e ricevere simultaneamente) o half-duplex (se a turno inviano e ricevono). Alcuni bus seriali possono avere la necessità di un solo collegamento tra un dispositivo che invia e quello che riceve (single wire).
Arduino UNO ha una porta seriale hardware TTL, multiplexed sui GPIO 0 e 1. Il pin 0 (RX) è usato per ricevere, il pin 1 (TX) per trasmettere. Questi pin sono connessi ai pin corrispondenti dell'ATMega328 USB-to-TTL-serial-chip, che converte seriale TTL in USB e viceversa. Questi pin sono quindi usati per più di una funzione (come seriale e come GPIO), e bisogna quindi fare attenzione alle interferenze: se si usa per una funzione, non può essere usata per l'altra e viceversa.
La libreria Serial dà completo controllo e gestione della porta seriale, ed è utilizzabile per comunicare (inviare e ricevere) messaggi al sistema collegato via USB.

Un'interfaccia seriale sincrona accoppia sempre la sua linea per trasmettere i dati con un segnale di clock, per sincronizzare. Questo permette di avere trasferimenti più rapidi, al prezzo di una maggiore complessità. Esempi di protocolli sincroni sono I^2C e SPI.

I^2C è un bus/protocollo standard che permette una comunicazione veloce, robusta, a 2 vie, utilizzando un numero minimo di pin. Fu introdotto da Philips negli anni 80, e diventò uno standard negli anni 90. È chiamato "two-wire" (TW) perché utilizza due linee per la comunicazione (clock, data). Utilizza uno spazio di indirizzamento a 7 o 10 bit, il bus speed arriva a 100 kb/s in standard e 10 kb/s in low-speed mode. Versioni recenti hanno introdotto fast mode a 400 kbit/s, fast mode plus a 1 Mbit/s e High-Speed mode a 3.4 Mbit/s.



(tratto da capitolo 8, [EA])

Figure 8-1: I²C reference hardware configuration

L'architettura di I^2C è di tipo master-slave: un I^2C bus è controllato da un master device, tipicamente il microcontrollore, e contiene uno o più slave device che ricevono informazioni dal master. Più device condividono le stesse due linee di comunicazione: una clock signal, che serve a sincronizzare la comunicazione, e una linea bidirezionale per inviare e ricevere dati dal master agli slave.
La comunicazione viene sempre fatta partire dal master, e gli slave possono solo rispondere. Tutti i comandi inviati dal master sono ricevuti da tutti i dispositivi sul bus, tuttavia ogni slave ha un proprio ID unico di 7 bit che viene specificato dal master

quando inizia la comunicazione, e solo lo slave target reagisce al messaggio inviato dal master. Lo schema di comunicazione tipico è:

1. master invia start bit
2. master invia l'id a 7 bit del dispositivo con cui vuole comunicare
3. master invia read (0) o write (1) bit, a seconda che voglia scrivere dati nei registri del dispositivo o leggerli
4. slave risponde con un ack (un ACK bit, che è attivo quando vale 0)
5. in write mode, il master invia un byte di informazioni alla volta e lo slave risponde con degli ACK. In read mode il master riceve 1 byte alla volte e invia un ACK dopo ogni byte
6. quando la comunicazione è finita, il master invia uno stop bit

Arduino/ATMega328P supporta nativamente I^2C con 2 pin analogici, A4 e A5, che sono multiplexed fra il convertitore analogico/digitale (ADC) e l'interfaccia HW per I^2C. La libreria di riferimento è Wire. Quando si inizializza la libreria, i pin verranno usati per I^2C e non potranno più essere usati per comunicazioni analogiche.

Lo SPI bus implementa un protocollo seriale full-duplex che permette la comunicazione simultanea bidirezionale fra un master e uno o più slave. Come I^2C è un protocollo seriale, basato su schema master-slave, ma diversamente da esso usa linee diverse per trasmettere e ricevere dati, e utilizza una linea per selezionare lo slave. Non segue un vero e proprio standard formale, per cui ci possono essere lievi differenze implementative e relative all'insieme dei comandi supportati. I dispositivi SPI sono anch'essi sincroni, come I^2C: i dati sono inviati in sync con un segnale di clock condiviso (SCLK).
L'architettura di SPI è quindi master-slave: un SPI bus è controllato da un master device (tipicamente il microcontrollore), e contiene uno o più slave device che ricevono informazioni dal master. Per comunicare ci sono 3 linee:

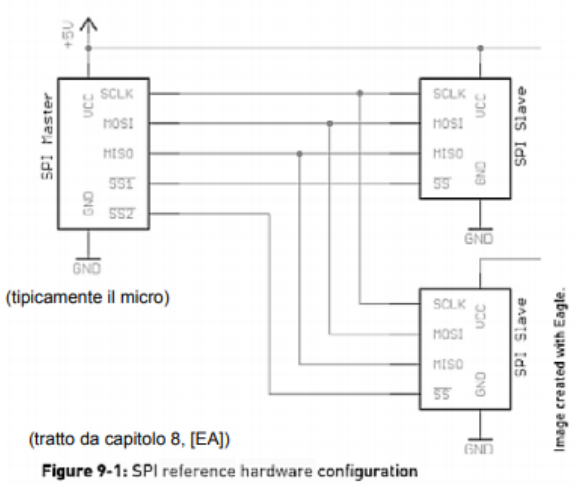
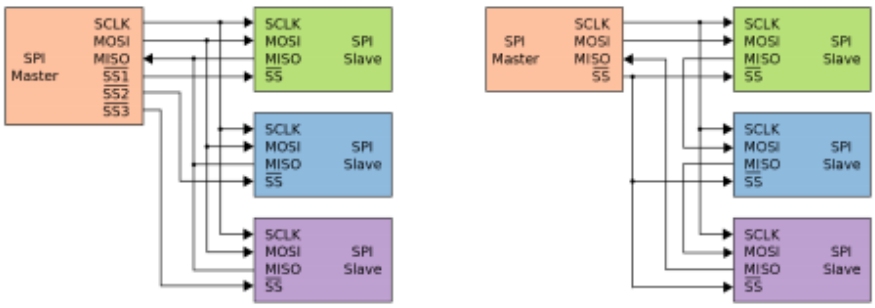


Figure 9-1: SPI reference hardware configuration

- uno shared clock signal (SCK), per sincronizzare la comunicazione
- una linea Master Out Slave In (MOSI), che serve per inviare i dati dal master allo slave
- una linea Master Out Slave Out (MISO), che serve per inviare i dati dallo slave al master
- una linea Slave Select (SS), che serve per selezionare lo slave

Nel caso in cui serva collegare più slave allo stesso master ci sono due possibilità architetturali:

- utilizzare più linee SS (immagine in basso a sinistra)
- utilizzare un collegamento daisy chain, in cui il master comunica con tutti gli slave simultaneamente (immagine in basso a destra)



Arduino/ATMega328P supporta nativamente la comunicazione via SPI, con l'uso multiplexed di 4 GPIO: 10 (SS), 11 (MOSI), 12 (MISO), 13 (SCK).
In Wiring è disponibile la libreria SPI che permette di interagire agilmente con dispositivi SPI. Per iniziare una sessione di uso della porta SPI è disponibile il metodo SPI.beginTransaction(settings), in cui si specifica la configurazione "settings" mediante la funzione SPISettings. La fine della sessione avviene col metodo SPI.endTransaction(). La configurazione da specificare in beginTransaction include i seguenti parametri:

- massima velocità che il dispositivo può usare
- se i dati sono in formato MSB (Most Significant Bit) o LSB (Least Significant Bit)
- modalità relativa al sampling dei dati

Per inviare e ricevere dati è disponibile il metodo transfer(), che permette farlo anche simultaneamente. È possibile trasferire anche più byte, specificando un buffer.
la sequenza di azioni da fare per interagire con un dispositivo slave è:

- settare lo slave select pin a LOW, con digitalWrite sul pin SS
- chiamare SPI.transfer() per trasferire i dati
- settare il pin SS a HIGH

In definitiva, quali sono le differenze? Molti dispositivi sono disponibili sia su I^2C che su SPI. I^2C richiede solo due linee e la modalità di indirizzamento slave è più agile, invece SPI può operare a velocità più elevate, in generale è più semplice da usare e non sono richieste resistenze di pull-up.

Cosa sono le resistenze pull-down e pull-up? La resistenza di pull-down, ad esempio nel caso del bottone, è inserita per fare in modo che quando il bottone non sia premuto, il valore del pin di input sia pilotato al valore LOW (si chiama pull-down in questo caso perché porta il segnale al valore LOW, essendo collegato a GND). Quelle di pull-up invece sono collegate a VCC, "tirando" la tensione a valore HIGH. Le resistenze di pull-up e pull-down hanno tipicamente valore molto alto (es: 10 KΩ) quando il pulsante è premuto, vogliamo che la corrente che fluisce nel ramo con la resistenza sia minima.

SENSORI E ATTUATORI

Introduzione

Un sistema embedded interagisce con l'ambiente in cui è situato mediante sensori e attuatori.
I sensori sono dispositivi trasduttori che permettono di misurare un certo fenomeno fisico (come la temperatura, le radiazioni, l'umidità, ecc.), o rilevare e quantificare una concentrazione chimica (es. il fumo). Forniscono una rappresentazione misurabile di un fenomeno su una certa specifica scala o intervallo (es. una tensione in Volt). Possono essere analogici o digitali. Nel caso di analogici, nel microcontrollore è tipicamente incluso un convertitore analogico-digitale (ADC).
Gli attuatori sono dispositivi che producono un qualche effetto misurabile sull'ambiente, e anche questi sono analogici o digitali.
Le grandezze fisiche oggetto di misura da parte dei trasduttori possono essere continue (ad esempio la temperatura di un ambiente o la velocità di rotazione di un motore) o discrete (ad esempio il verso di rotazione del motore, il numero di pezzi lavorati al minuto). Le informazioni associate alle grandezze fisiche sono dette segnali: le grandezze continue sono descritte da segnali analogici, quelle discrete sono descritte da segnali logici nel caso si abbiano due valori ammissibili, codificati se il numero di valori ammissibili è superiore a due.
Il principio di funzionamento di un sensore è basato su una legge fisica nota che regola la relazione grandezza fisica da misurare e una grandezza elettrica di uscita. Un esempio è la termoresistenza (sensore di temperatura): la resistività cambia al variare della temperatura. per poter acquisire il segnale occorre fornire una corrente (o una tensione) e misurare poi la tensione (corrente) generata.
I sensori sono classificati quindi in:

- analogici: forniscono un segnale elettrico continuo a risoluzione finita
- digitali: forniscono un'informazione di tipo numerico con risoluzione finita
    - logici: hanno un'uscita di tipo booleano
    - codificati: hanno un'uscita numerica codificata in una stringa di bit

Un segnale analogico prodotto da un sensore analogico pe poter essere elaborato dal calcolatore dev'essere campionato mediante un sistema di conversione A/D nel caso di sensori analogici (dà origine ad un segnale digitale). Il campionamento del segnale è l'acquisizione di campioni del segnale analogico ad istanti discreti di tempo. La quantizzazione è un'operazione di approssimazione del valore campionato al più vicino valore digitale.

La misura di una grandezza fisica è il confronto di due quantità appartenenti a una grandezza della stessa specie (omogenee): stabilisce in che rapporto una quantità incognita stia rispetto ad un'altra, che opera come riferimento, esprime il risultato della misura. L'operazione che porta a questo risultato è detta misurazione. Il Sistema Internazionale (SI) è il più diffuso sistema di unità di misura.
Misure ripetute di uno stesso parametro non forniscono lo stesso valore, e le cause possono essere molteplici (legate per esempio allo strumento o all'ambiente). Per esprimere questa dispersione di valori si introduce il concetto di incertezza di misura:

- esprime la dispersione dei valori, reale o potenziale
- può essere interpretata come il dubbio riguardante il risultato
- una grandezza fisica può essere determinata soltanto a un livello finito di incertezza (errore)

Il risultato della misura è costituito sempre da un numero e da un'incertezza, generalmente preceduta dal simbolo ± e da un'unità di misura.
Gli errori di misura possono essere:

- sistematici: se, fissate le condizioni sperimentali, in grandezza e segno l'errore ha la stessa influenza sul risultato della misura
- accidentali: errori la cui influenza sulla misura può cambiare in grandezza e segno se si ripete la precedente misurazione, a causa per esempio delle condizioni ambientali
- grossolani: riguardano l'operatore o guasti dello strumento

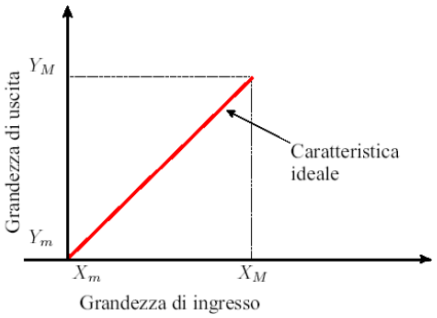
Le caratteristiche metrologiche di un sensore sono:

- caratteristiche statiche: si riferiscono a condizioni di funzionamento in cui viene variata molto lentamente la variabile di ingresso del sensore, registrando la corrispondente variabile di uscita. Abbiamo quindi accuratezza, precisione, linearità, sensibilità, risoluzione, ripetibilità, riproducibilità, stabilità.
- caratteristiche dinamiche: se la variabile di ingresso varia velocemente, l'uscita può evidenziare un'attenuazione rispetto alla caratteristica statica ed un ritardo.

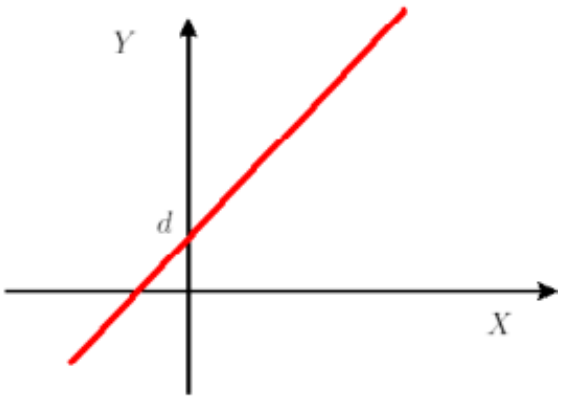
La caratteristica statica di un trasduttore è definita da una funzione tipo Y=f(X), dove X è il segnale di ingresso è Y quello di uscita dal trasduttore. La caratteristica è definita su un intervallo finito chiamato campo di ingresso avente estremi Xm e XM e ha valori sul campo di uscita (output range o span) con estremi Ym e YM. Si definiscono:

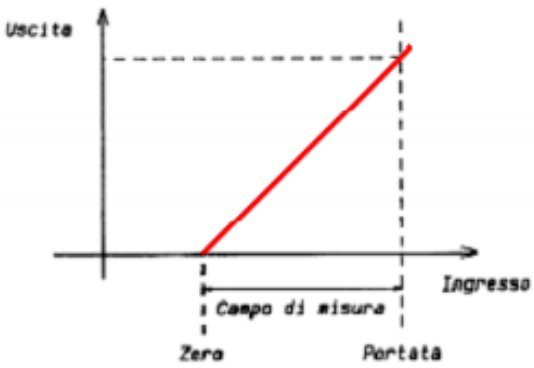
- range d'ingresso: Xs = XM - Xm
- range d'uscita: Ys = YM - Ym

La caratteristica statica di un sensore deve avere idealmente un andamento lineare: la costante di proporzionalità fra valori di ingresso e di uscita viene chiamata guadagno (K) del trasduttore. I trasduttori commerciali hanno una caratteristica statica reale che si differenzia da quella ideale a causa di inevitabili imperfezioni costruttive. La qualità di un sensore si misura in base a quanto la caratteristica reale si scosta da quella ideale.



La linearità di un trasduttore può essere definita in diversi modi, non del tutto equivalenti. Per i trasduttori lineari la relazione tra grandezza fisica misurata e il segnale di uscita è descrivibile attraverso una semplice relazione matematica: Y = KX, dove K è il guadagno. L'errore di linearità è la massima deviazione dell'uscita del trasduttore rispetto alla caratteristica lineare che approssima al meglio la caratteristica reale. La caratteristica lineare viene normalmente ottenuta col metodo dei minimi quadrati. L'errore di offset (in basso a sinistra) è il valore "d" che assume l'uscita del trasduttore quando la grandezza da misurare è nulla: Y=f(X) = KX + d. L'errore di soglia (in basso a destra) corrisponde al più basso livello di segnale rilevabile dal sensore: esso non sempre coincide con lo zero della grandezza da misurare.





L'errore di guadagno è la differenza tra il guadagno della caratteristica ideale del trasduttore (K) e il guadagno della retta (K1) che approssima meglio la caratteristica reale del trasduttore. L'errore di guadagno è solitamente espresso in percentuale: eG= (|K1-K|) / K * 100.
L'operazione di campionamento non produce in via teorica un degrado dell'informazione associata al segnale, se si rispettano le condizioni del teorema di campionamento. L'operazione di quantizzazione comporta inevitabilmente l'introduzione di un errore sul segnale acquisito.
L'accuratezza di un sensore è la misura di quanto il valore letto dal sensore si discosti dal valore corretto: spesso si fa riferimento, più che all'accuratezza, all'equivalente incertezza della misura. Come si calcola? È il massimo scostamento tra misura fornita dal sensore e il valore vero del segnale misurato.
La precisione descrive quanto un sensore sia soggetto o meno ad errori accidentali: se eseguiamo un numero elevato di letture con un sensore che ha precisione elevata, il range di tali valori sarà piccolo. Alta precisione, tuttavia, non implica elevata accuratezza. È legata alla ripetibilità o riproducibilità: esprime la riproducibilità di una misura, ossia esprime l'attitudine di un sensore a fornire valori della grandezza in uscita poco diversi tra loro, a parità di segnale di ingresso (stesso valore vero) e nelle stesse condizioni di lavoro. Si calcola considerando la deviazione dei valori letti rispetto al valor medio.
La calibrazione di un sensore è l'aggiustamento dei suoi parametri per farne corrispondere l'uscita a valori rilevati accuratamente con un altro strumento. La taratura, invece, è la misurazione della grandezza di uscita per valori noti della grandezza di ingresso al trasduttore stesso.
Ogni sensore ha la sua dinamica:

- la risposta di un sensore non è istantanea
- è spesso trascurabile rispetto alle dinamiche del processo

Quando in ingresso al trasduttore applichiamo una sollecitazione a gradino (cioè un gradino della grandezza da misurare) l'uscita (risposta) varierà fino a raggiungere, dopo un certo tempo, un nuovo valore. Si definisce:

- tempo di salita: il tempo impiegato per passare dal 10% al 90% del valore finale
- tempo di risposta: il tempo impiegato per raggiungere una percentuale prefissata del valore finale

Tipologie di sensori

I sensori di prossimità sono sensori in grado di rilevare la presenza di oggetti nelle immediate vicinanze del "lato sensibile" del sensore stesso, senza che vi sia un effettivo contatto fisico: la distanza entro cui questi sensori rilevano oggetti è definita portata nominale (o campo sensibile). Alcuni modelli dispongono di un sistema di regolazione per poter calibrare la distanza di rilevazione. L'assenza di meccanismi d'attuazione meccanica, o di un contatto fisico tra sensore e oggetto, fa si che questi sensori presentino un'affidabilità elevata. Normalmente i sensori di prossimità rilevano solamente la presenza o l'assenza di un oggetto all'interno della loro portata nominale, e conseguentemente il segnale elettrico di uscita sarà di tipo on/off in quanto deve rappresentare solo gli stati assenza/presenza. I sensori di prossimità possono essere realizzati basandosi su diversi tipi di tecnologie: sensori induttivi, capacitivi, magnetici, sensori ad ultrasuoni, sensori ottici.
I sensori ad ultrasuoni funzionano sul principio del Sonar: emettono impulsi sonori ultrasonici, e rilevano un'eventuale eco di ritorno generata dalla presenza di un oggetto all'interno della portata nominale. Gli svantaggi sono il costo e la bassa velocità di commutazione, i vantaggi sono:

- portate nominali elevate (fino a 10 m)
- immunità a disturbi elettromagnetici
- possono rilevare oggetti di qualsiasi materiale (eccetto materiali fonoassorbenti)

- possono rilevare oggetti senza che questi siano stati preventivamente preparati

Una certa attenzione va però posta nella dimensione e nell'orientamento della superficie dell'oggetto che si rivolge al sensore: infatti una superficie troppo piccola o orientata malamente (non ortogonale alla direzione di lettura del sensore) può non assicurare la generazione di un'eco rilevabile.
Il sensore HC-SR04 invia un impulso sul pin Trig e misura quanto tempo impiega ad arrivare l'echo sul pin Echo: dato tale tempo è possibile ricavare la distanza. La routine pulseIn di Wiring monitora il pin di input dove deve ricevere l'impulso, tenendo traccia del tempo intercorso. I tempi possono essere molto piccoli, dell'ordine delle decine di microsecondi, per cui è importante adottare un approccio molto efficiente:

- accesso diretto ai registri associati ai pin
- accesso diretto al registro che tiene traccia dei tick del processore

I sensori di prossimità ottici si basano sulla rilevazione della riflessione di un fascio luminoso da parte dell'oggetto rilevato (sono chiamati anche fotoelettrici). Normalmente viene usato un fascio di raggi infrarossi, in quanto questa radiazione difficilmente si confonde con i disturbi generati da fonti luminose ambientali. Nella modalità d'uso più semplice, il fascio viene riflesso dalla superficie stessa dell'oggetto rilevato, per lo stesso fenomeno per cui la luce visibile può essere riflessa e percepita dai nostri occhi. Il problema è che la quantità di radiazione riflessa dipende dalla composizione e dall'orientamento della superficie; pertanto il campo sensibile di questi sensori di prossimità dipende sostanzialmente dalla natura della superficie dell'oggetto da rilevare, tipicamente da 10 a 100 cm.

Tra i sensori di movimento abbiamo:

- PIR: sensore infrarosso passivo, è un sensore elettronico che misura i raggi infrarossi (IR) irradiati dagli oggetti nel suo campo di vista e possono essere usati come rilevatori di movimento. Un PIR non rileva autonomamente un movimento, tuttavia rileva brusche variazioni di temperatura che modificano lo stato che il PIR aveva memorizzato come "normale". Quando qualcosa o qualcuno passa di fronte a uno sfondo, ad esempio un muro, precedentemente "fotografato" dal PIR come stato normale, la temperatura in quel punto si innalza bruscamente, passando dalla temperatura della stanza a quella del corpo: questo rapido cambiamento attiva il rilevamento, e lo spostamento di oggetti di temperatura identica, com'è prevedibile, non innesca alcuna rilevazione.
- Trasduttori di posizione angolare: ne esistono di due tipi:
    - uno rileva la posizione senza avere un vincolo meccanico, cioè sfrutta la forza di gravità (es. inclinometro, detto anche convertitore d'angolo)
    - uno rileva la posizione sfruttando un vincolo meccanico (encoder). L'encoder è un dispositivo elettromeccanico che converte la posizione angolare del suo asse rotante in brevi impulsi elettrici che necessitano di essere elaborati da un circuito di analisi del segnale sotto forma di segnali numerici digitali. Ce ne sono di vari tipi (tachimetrici, relativi, assoluti).

  Ci sono varie tecnologie (capacitivi/induttivi, magnetici, potenziometrici, ottici).

Tra i sensori di accelerazione abbiamo:

- accelerometro: sensore che misura l'accelerazione a cui è soggetto l'oggetto a cui è connesso il sensore, misurandone in realtà la forza specifica (per unità di massa). Ce ne sono di vari tipi, che si basano su principi di funzionamento diversi.

- giroscopio: dispositivo fisico rotante che mantiene il suo asse di rotazione orientato in una direzione fissa, per effetto della legge di conservazione del momento angolare. È usato per scopi vari, per esempio può fungere da bussola, indicando la direzione verso il nord.

Gli accelerometri e i giroscopi elettronici si basano su tecnologia MEMS (Micro Electro-Mechanical Systems). I MEMS sono costituiti da un insieme di dispositivi di varia natura (meccanici, elettrici ed elettronici) integrati in forma altamente miniaturizzata (ordine dei micrometri) su uno stesso substrato di materiale semiconduttore (ad esempio il silicio). Coniugano le proprietà elettriche degli integrati a semiconduttore con proprietà opto-meccaniche: abbinano funzioni elettroniche, di gestione dei fluidi, ottiche, biologiche, chimiche e meccaniche in uno spazio, integrando la tecnologia dei sensori e degli attuatori e le più diverse funzioni di gestione dei processi. Sono fondamentali per lo sviluppo dei sistemi embedded moderni.

I sensori di contatto sono:

- pulsanti tattili e microswitch
- potenziometri: permette di variare in modo meccanico la resistenza del componente, e di conseguenza la tensione ai morsetti. È un dispositivo elettrico equivalente ad un partitore di tensione resistivo variabile.
- potenziometro lineare: la tensione di uscita è legata linearmente a quella di ingresso attraverso un rapporto
- sensori capacitivi: si basano sul principio della rilevazione della capacità elettrica di un condensatore: il loro lato sensibile ne costituisce un armatura, l'eventuale presenza nelle immediate vicinanze di un oggetto conduttore realizza l'altra armatura del condensatore. Così la presenza di un oggetto crea una capacità che i circuiti interni rilevano, comandando la commutazione del segnale di uscita.

I sensori di pressione permettono di misurare la forza esercitata su una determinata superficie.

I sensori ottici (o sensori di immagini) sono dispositivi che convertono un'immagine ottica in un segnale elettrico: i componenti vengono utilizzati soprattutto nelle fotocamere digitali, nelle telecamere, nelle videocamere e in altri dispositivi che trattano elettronicamente immagini. Ne esistono molti tipi, in base a caratteristiche come metodo di rilevamento dei colori, tecnologia, sensibilità alla luce, risoluzione, ecc. Una classificazione di base è fra quelli che riprendono a colori e quelli che riprendono in bianco e nero. Il principio di funzionamento è il seguente:

1. l'immagine viene focalizzata su una griglia composta da una miriade di piccoli sensori puntiformi
2. i singoli sensori convertono la luminosità rilevata
3. il sensore che riprende in bianco e nero è quello più semplice ed è anche il primo nato, successivamente si è arrivati a rilevare anche il colore

I foto-rivelatori sono dispositivi in grado di rivelare la radiazione elettromagnetica, fornendo in uscita un segnale avente un'intensità di corrente o una differenza di potenziale proporzionale all'intensità della radiazione rilevata. Esistono tipi diversi di foto-rivelatore, realizzati in base a diversi effetti di interazione tra la radiazione e la materie. Possono differire per la porzione di spettro elettromagnetico che sono in grado di rilevare, e per l'intensità luminosa minima che riescono a misurare (alcuni sono in grado di rivelare i singoli fotoni). Questi dispositivi sono indicati anche con il termine fotocellula. Le varie tecnologie sono:

- fotoresistenza: basato sulle cariche fotogenerate (cioè sugli elettroni eccitati dalla luce incidente dalla banda di valenza alla banda di conduzione) in un semiconduttore

- fotodiodo: basato sulle cariche fotogenerate in una giunzione p-n
- Change Coupled Device (CCD): circuiti integrati basati su cariche fotogenerate in un semiconduttore

Riguardo ai sensori legati ad elettricità e magnetismo abbiamo:
- rilevatori di tensione e corrente
- magnetometro
- bussola

Riguardo al tempo atmosferico abbiamo:
- sensore di temperatura
- sensore di umidità
- sensore di pressione barometrica

Il Sistema di Posizionamento Globale (GPS) è un sistema di posizionamento e navigazione satellitare civile che, attraverso una rete satellitare dedicata di satelliti artificiali in orbita, fornisce ad un terminale mobile o ricevitore GPS informazioni sulle sue coordinate geografiche ed orario, in ogni condizione metereologica, ovunque sulla Terra o nelle sue immediate vicinanze ove vi sia un contatto privo di ostacoli con almeno quattro satelliti del sistema. Il sistema è gestito dal governo degli USA, ma liberamente accessibile da chiunque sia dotato si ricevitore GPS. Il grado di accuratezza è dell'ordine dei metri, in dipendenza dalle condizioni meteorologiche, dalla disponibilità e dalla posizione dei satelliti rispetto al ricevitore, dalla qualità e dal tipo di ricevitore, dagli effetti di radiopropagazione del segnale radio in ionosfera e troposfera (es. riflessione) e dagli effetti della relatività. Il principio di funzionamento è il seguente:
- si basa su un metodo di posizionamento sferico (trilaterazione), che parte dalla misura del tempo impiegato da un segnale radio a percorrere la distanza satellite-ricevitore
- la localizzazione avviene tramite la trasmissione di un segnale radio da parte di ciascun satellite e l'elaborazione dei segnali ricevuti da parte del ricevitore

I sensori per identificazione sono:
- RFID: tecnologia che permette di avere dei piccoli tag in grado di contenere un certo quantitativo di dati e di essere rilevati, letti e scritti a distanza, wireless da opportuni lettori
- NFC: concettualmente analoghi ai RFID, la scoperta e interazione avviene per contatto fra lettore e tag
- iBeacon: dispositivi che sfruttano la tecnologia BLE (Bluetooth low energy) per trasmettere uno UUID nel raggio di una certa località e rilevabile mediante opportuni ricevitori Bluetooth 4.0

Attuatori e dispositivi di output

I trasduttori convertono una forma di energia in un'altra. I trasduttori di output (attuatori) permettono di realizzare sistemi embedded che eseguono e controllano azioni sull'ambiente fisico in cui sono immersi: l'azione più semplice che possiamo immaginare è accendere o spegnere un dispositivo. Alcuni esempi sono led, buzzer, motori, ma in generale ogni dispositivo che può essere acceso o spento (una ventola, una radio, un'automobile).
Per l'interfacciamento abbiamo due casi principali:
- è sufficiente la corrente/tensione in uscita ai GPIO per alimentare il trasduttore (es. coi led)

- in caso contrario, il dispositivo dev'essere alimentato da un circuito (e alimentazione) separato. In questo caso, mediante un GPIO si apre/chiude l'altro circuito mediante dispositivi come transistor e relè che fungono da interruttori.

Dal punto di vista elettronico, i dispositivi pilotabili sono classificabili in due categorie:

- carichi resistivi: possono essere assimilati ad un componente che al passaggio della corrente oppone una certa resistenza che implica una variazione della tensione
- carichi induttivi: dispositivi che operano inducendo corrente su un filo mediante la corrente su un altro filo o immergendo il filo in un campo magnetico (es. motori e solenoidi). Generano una tensione/corrente inversa: è necessario dotare il circuito di diodi di protezione per eliminare o ridurre l'effetto di questa corrente, che potrebbe altrimenti danneggiare il circuito.

Il LED (Light Emitting Diode) è un diodo ad emissione luminosa: è un dispositivo optoelettronico che sfrutta le proprietà ottiche di alcuni materiali semiconduttori di produrre fotoni attraverso un fenomeno di emissione spontanea. Sono interessanti per le loro caratteristiche di elevata efficienza luminosa A.U./A e di affidabilità.

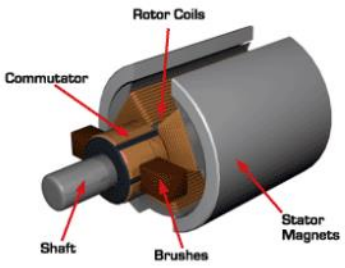
Un display LCD è un display basato sulle proprietà ottiche di particolari sostanze denominate cristalli liquidi: tale liquido è intrappolato tra due superfici vetrose provviste di numerosi contatti elettrici con i quali poter applicare un campo elettrico al liquido contenuto; ogni contatto elettrico comanda una piccola porzione del pannello identificabile come un pixel (o subpixel per gli schermi a colori). Si ha un basso consumo di potenza elettrica, il che li rende indicati per i sistemi embedded.

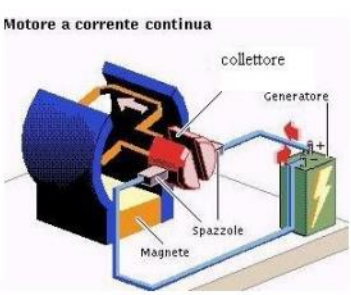
I motori elettrici sono macchine elettriche in cui la potenza di ingresso è di tipo elettrico e quella di uscita è di tipo meccanico, assumendo la funzione di attuatore. Il motore elettrico, così come l'alternatore, è composto da statore e rotore. Questi componenti generano un campo magnetico, in alcuni casi grazie all'uso di magneti, che determinano quindi il movimento del rotore. Fra i motori elettrici più usati nell'embedded abbiamo:

- motori in corrente continua (CC): sono particolarmente indicati nei progetti in cui occorre
  - la rotazione continua
  - 360° di movimento continuo
  - elevata velocità o coppia (motoriduttori)
  - controllare la velocità di rotazione

  Esempi di utilizzo sono
  - ruote di un robot
  - slider per telecamere
  - avvolgi-tenda o tapparella

  Lo schema di funzionamento è il seguente:
  1. la corrente elettrica continua passa in un avvolgimento di spire che si trova nel rotore, composto da fili di rame, creando un campo elettromagnetico al passaggio di corrente.





2. questo campo elettromagnetico è immerso in un altro campo magnetico creato dallo statore, il quale è caratterizzato dalla presenza di una o più coppie polari (calamite, elettrocalamite, ecc.).
3. il rotore per induzione elettromagnetica inizia a girare, in quanto il campo magnetico del rotore tende ad allinearsi a quello dello statore analogamente a quanto avviene per l'ago della bussola che si allinea col campo magnetico terrestre.
4. durante la rotazione il sistema costituito dalle spazzole e dal collettore commuta l'alimentazione elettrica degli avvolgimenti del rotore in modo che il campo magnetico dello statore e quello del rotore non raggiungano mai l'allineamento perfetto, in tal modo si ottiene la continuità della rotazione.

Ci sono due tipi di motori CC:

- brushless (senza spazzole, ossia i passo-passo)
- brushed: con spazzole, sono meno costosi, ma durano di meno. La corrente è trasferita alla bobina (coil) mediante spazzole. Quando la corrente passa nella bobina del rotore, genera un campo magnetico che è attratto o respinto dai magneti presenti nello statore. Mediante le spazzole, viene ad essere invertita la polarità ogni mezzo giro, creando un momento angolare

Sfruttando il medesimo principio, è possibile usare il motore come generatore di corrente (in direzione opposta): ruotando il rotore a mano è possibile generare corrente per, ad esempio, illuminare un led. Il controllo sulla velocità avviene cambiando la tensione ai morsetti: in questo modo cambiamo la velocità del motore. Invertendo la tensione è possibile invertire il senso di rotazione (allo scopo a supporto si usa un circuito integrato chiamato ponte H).

In generale un motore DC può richiedere più corrente di quella fornita dalla parte di alimentazione di Arduino, e inoltre può generare picchi di corrente in direzione opposta, in virtù del suo principio di funzionamento. Per cui quando si usano motori DC è opportuno isolarli dal punto di vista dell'alimentazione da Arduino, per pilotarli senza problemi.

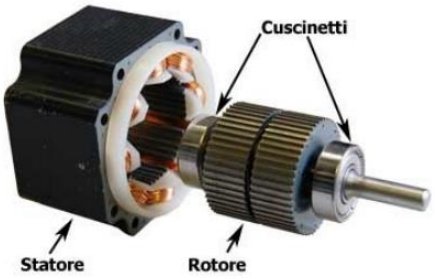
I transistor in generale sono usati per molti scopi, fra cui quello di interruttore (accendendolo e spegnendolo rapidamente).

È possibile invertire il senso di marcia nel motore invertendo la polarità della tensione applicata ai morsetti. Allo scopo nei circuiti si usa tipicamente un componente chiamato ponte H: esso integra 4 switch (realizzati da transistor) e tutta la circuiteria di protezione. Ha 4 stati operazionali:

- open: tutti gli switch aperti e il motore non gira
- forward: sono aperti due switch su diagonale opposta, la corrente fluisce e il motore si muove in una direzione
- braking: tutto il movimento residuo causato dal momento angolare viene cessato e il motore si ferma
- backward: sono aperti gli altri due switch su diagonale opposta, la corrente fluisce nel motore nella direzione opposta e il motore si muove nella direzione opposta.

Se entrambi gli switch a sinistra o a destra vengono chiusi, si crea un corto fra la batteria e terra: la batteria si riscalda velocemente e potrebbe in alcuni casi prendere fuoco o esplodere; il ponte H può danneggiarsi, così come tutte le altre parti del circuito; è il caso in cui un'errata programmazione può creare corti e danni. Alcuni chip hanno un circuito di protezione, ma bisogna fare comunque molta attenzione. La pratica di programmazione da seguire è di disabilitare il chip prima di cambiare lo stato di uno qualsiasi degli switch.

- motori passo-passo: spesso chiamato anche stepper, è un motore elettrico in CC senza spazzole (brushless) che può suddividere la propria rotazione in un grande numero di step. La posizione del motore può essere controllata accuratamente. Trovano il loro miglior impiego nei progetti in cui si ha bisogno di controllare in modo accurato la posizione dell'albero. Alcuni esempi sono stampanti 3D, macchine a controllo numerico, robotica per controllo bracci.



- servo-motori: come i passo-passo permettono di pilotare in modo preciso l'angolo in cui deve posizionarsi il rotore. A differenza dei motori passo-passo, però, la specifica posizione è assoluta, non relativa alla posizione corrente (questo semplifica la programmazione). Lo svantaggio rispetto ai passo-passo è l'escursione limitata (0-180°).
  Come funzionano? I servo si controllano specificando precisamente l'angolo al quale devono posizionare l'albero (tipicamente da -90° a +90°). A livello di dispositivo sono caratterizzati da tre fili (2 per l'alimentazione e 1 come segnale di controllo digitale). il controllo avviene inviando uno stream di impulsi al segnale di controllo ad una specifica frequenza (tipicamente 50 impulsi al secondo, 50Hz). La lunghezza dell'impulso determina l'angolo al quale vogliamo portare il servo. Gli impulsi tipicamente vanno da 1 ms (-90°) a 2 ms (+90°): più è lunga la durata dell'impulso, più grande è la durata dell'angolo specificato. La scala è lineare, per cui l'impulso di durata intermedia (1.5 ms) indica il centro (0°). Tuttavia la lunghezza reale degli impulsi può variare da servo a servo.

  In Wiring è disponibile la libreria Servo, che comprende funzioni tra cui attach(pin) per specificare il pin a cui è collegato il motore e write(angle) per specificare l'angolo, da 0 a 180.

Il relay (relè) permette di aprire e chiudere un secondo circuito (in cui ad esempio è necessario usare tensioni e correnti elevate) mediante l'azione di un elettromagnete, pilotato dal primo circuito. Internamente un relay contiene due circuiti elettricamente disaccoppiati, in cui uno (dove circolano correnti basse e ci sono tensioni ridotte) può pilotare/commutare l'altro (in cui circolano correnti anche elevate) e ci sono tensioni significative. Le bobine usate internamente nell'elettromagnete possono generare picchi di tensione quando sono disattivate che possono danneggiare transistori e microcontrollori collegati, per cui è necessario utilizzare opportuni diodi in parallelo al relay, come nel caso dei motori.

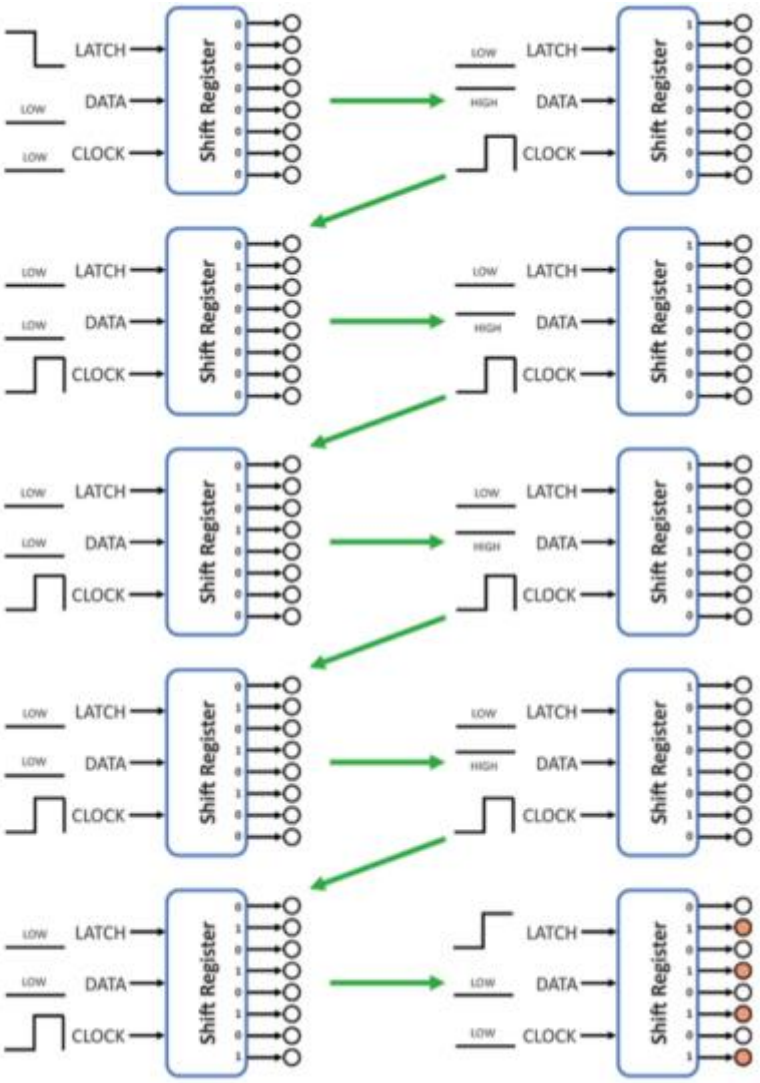
E quando i pin non bastano? È un problema ricorrente. Come soluzione si possono usare schede microcontrollori con numero di pin adeguato (es. Arduino Mega con 54 pin I/O) o usare dispositivi come registri a scorrimento. Gli shift register sono dispositivi utili per convertire input sequenziale/parallelo in output parallelo/sequenziale. La conversione (seriale -> parallelo) è fatta mediante registri SIPO (Serial In Parallel Out):

- dispositivi che accettano in ingresso uno stream sequenziale di bit e fornisce in uscita l'output di tali bit come porta di I/O parallela
- il numero di bit in uscita può variare - esempio: a 8, 16, 32 ..
- possono essere concatenati facilmente nello stesso circuito

I segnali in input sono:

- CLOCK: segnale che sincronizza l'acquisizione dei dati in input
- DATA: pin dove insiste lo stream di bit in input
- LATCH: segnale che campiona l'output e lo rende disponibile (register clock pin)

Di seguito un esempio col caricamento del valore 10101010:



SISTEMI EMBEDDED E MODELLAZIONE OO

Nella progettazione e sviluppo di sistemi software embedded ci sono due approcci da integrare:

- approccio top-down: si parte dal dominio e dalla sua modellazione con paradigmi di alto livello (Object Oriented)
- approccio bottom-up: si parte dalle caratteristiche a livello hardware e dal comportamento che il sistema deve avere a questo livello.

Verrà considerato l'approccio top-down.

I paradigmi di programmazione sono anche anzitutto paradigmi di modellazione, coi quali definisco in che modo concettualizzo, rappresento, progetto un sistema software e in che modo analizzo e formulo la soluzione di un problema in un certo dominio applicativo. Il modello è la rappresentazione degli aspetti salienti del sistema, astraendo da quelli che non sono significativi. Gli aspetti salienti concernono la struttura, il comportamento, le interazioni che il sistema deve avere al fine di ottenere le funzionalità volute. Esiste uno stretto rapporto tra modelling e programming. Analizzare il problema e definire soluzioni più astratte rispetto alla pura implementazione e ai relativi linguaggi è un aspetto centrale dell'ingegneria. I vantaggi sono:

- miglior comprensione del funzionamento del sistema e gestione più efficace delle relative complessità
- riusabilità e portabilità
- estendibilità
- possibilità di avere forme di analisi più rigorosa
- correttezza e verificabilità

Le dimensioni fondamentali della modellazione sono:

- struttura: come sono organizzate le varie parti
- comportamento: il comportamento computazionale di ogni singola parte
- interazione: come interagiscono le parti

I paradigmi di modellazione definiscono un insieme coerente di concetti e i principi con cui definire i modelli (ad esempio il paradigma ad oggetti). Sono spesso correlati ai paradigmi di programmazione (es. OOP).
I linguaggi di modellazione forniscono un modo rigoroso non ambiguo per rappresentare i modelli (ad esempio il linguaggio UML). Sono spesso correlati a paradigmi di modellazione specifici (ad esempio UML è fortemente ispirato al paradigma OO, in particolare per la parte strutturale e per i diagrammi delle interazioni).

Il paradigma ad oggetti è fra i paradigmi di riferimento nella modellazione e progettazione del software: si tiene ad un livello di astrazione efficace per catturare aspetti essenziali del dominio, in particolare aspetti strutturali. Proprietà importanti sono:

- modularità
- incapsulamento
- meccanismi per riuso ed estendibilità
- aspetti non direttamente catturati dal paradigma (concorrenza, interazioni asincrone, distribuzione)

Su microcontrollore si hanno due micro-parti di base:

- controllore: incapsula la logica di controllo per lo svolgimento del compito o dei compiti del sistema. Allo scopo usa/osserva/gestisce risorse (ad esempio sensori ed attuatori). È il modello tipicamente attivo
- elementi controllati: modellano le risorse gestite/utilizzate dal controllore per svolgere il compito (ad esempio dispositivi di I/O, sensori, attuatori). Incapsulano funzionalità utili al controllore. Sono il modello tipicamente passivo.

Gli aspetti di basso livello (ad esempio l'uso di primitive Wiring per pilotare i pin) sono "nascosti" nell'implementazione, non esposti a chi usa gli oggetti (cioè il controller): si ha in questo modo maggiore leggibilità, riusabilità e portabilità del codice.

Nel modello a loop ad ogni ciclo il controller legge gli input di cui necessita e sceglie le azioni da compiere: la scelta dipende dallo stato in cui si trova, e l'azione può cambiare anche lo stato interno. Si ha necessità di linguaggi e modelli più astratti e rigorosi che non lo pseudocodice: le macchine a stati finiti (FSM), sincrone o asincrone.
Nel modello a loop si hanno:

- efficienza: nonostante a livello logico il controllore sia in idle in attesa di percepire input, con questo approccio continua ad eseguire cicli, anche quando l'input non è cambiato
- reattività: la reattività del controllore dipende da quanto rapidamente viene completato un ciclo; se durante l'esecuzione del ciclo avvengono ripetute variazioni di stato nei sensori, tali variazioni non vengono rilevate

Il controller è concettualmente rappresentabile come oggetto secondo il modello Object-Oriented? Nei casi visti finora, led/luci/pulsanti sono rappresentabili come oggetti passivi: cambiano stato a fronte di azioni eseguite o dal mondo esterno (per esempio con un button) oppure dal controllore (ad esempio un led), e hanno specifiche interfacce che permettono al controller di interagirvi, o per eseguire azioni o per leggerne lo stato. Il controller è concettualmente un'entità attiva, dotata di flusso di controllo autonomo: incapsula a livello logico la strategia di controllo del sistema. Quale astrazione utilizzare per modellare questo tipo di entità?
Il controller è un'entità diversa dal punto di vista concettuale dagli oggetti: è attivo, non ha interfaccia, incapsula la strategia di controllo.
Utilizziamo la nozione di agente per identificare questi tipi di entità: sono entità attive, dotate di un flusso di controllo logico autonomo, progettate per svolgere uno o più compiti (task) che richiedono di elaborare informazioni che provengono in input dall'ambiente da sensori e di agire su attuatori in output, eventualmente comunicando con altri agenti.
Si ha una progettazione task-oriented.
Nel caso di controllori (e loop di controllo) complessi ed articolati, c'è la necessità di introdurre principi e tecniche di decomposizione e modularizzazione. Si effettua una decomposizione per task: è un approccio adottato anche negli approcci ad agenti (un agente è l'esecutore di uno o più task) e si ha integrazione con le macchine a stati (il comportamento di ogni task è descrivibile mediante una macchina a stati finiti).
I sistemi embedded di una certa complessità tipicamente sono distribuiti e costituiti da più sottosistemi embedded che interagiscono e cooperano (ad esempio lo Smart Home nel suo complesso, c'è una prospettiva IOT). Questo richiede paradigmi di modellazione che considerano interazione distribuita, comunicazione e cooperazione come aspetti di prima classe (ad esempio sistemi multi-agente, in cui la comunicazione fra agenti è basata su scambio asincrono di messaggi).

MODELLI BASATI SU MACCHINE A STATI FINITI

In generale, il comportamento dinamico di un sistema può essere modellato con modelli nel continuo o nel discreto: un esempio di modello nel continuo sono i sistemi di equazioni differenziali lineari. Le macchine a stati finiti sono il modello nel discreto più utilizzato per modellare sistemi embedded: la macchina a stati finiti è essa stessa un sistema discreto, ovvero opera in una sequenza di passi (discreti) e la sua dinamica è caratterizzata da sequenze di eventi (discreti); un evento avviene ad un determinato istante, non ha durata.
Molti sistemi embedded hanno una natura inerentemente discreta.

Si consideri un sistema che deve effettuare il conteggio delle auto in un parcheggio, per tenere traccia del numero di auto che ci sono in un qualsiasi istante. Si consideri un modello di sistema costituito di componenti:

- ArrivalDetector, sottosistema che rileva l'arrivo di una nuova auto
- DepartureDetector, sottosistema che rileva la dipartita di un auto
- Counter, sottosistema che tiene traccia del conteggio delle auto

In questo sistema ogni arrivo o dipartita delle auto sono modellati in modo naturale come eventi discreti.
Il sistema Counter reagisce in sequenza agli eventi di input che si presentano e produce un segnale di output: l'input è rappresentato da una coppia di segnali discreti (up/down) che in certi momenti hanno un evento (sono presenti) e in altri momenti non hanno eventi (assenti); l'output è un segnale discreto che, quando l'input è presente, ha un valore ed è un numero naturale, e negli altri momenti è assente. Quando un evento è presente alla porta up/down, Counter incrementa/decrementa il conteggio e produce il valore in uscita. In tutti gli altri casi (quando non c'è l'input) non produce output.
I segnali "u" e "d" di input (up e down) che rappresentano il succedersi di eventi discreti possono essere rappresentati come funzioni:

$$u: \mathbb{R} \longrightarrow \{ \text{absent, present} \}$$

ovvero ad ogni istante t, il segnale u(t) è valutato o in "absent", che significa che non ci sono eventi in quel momento, oppure "present", intendendo che c'è un evento in quel momento. Questi segnali vengono detti puri: non portano contenuto informativo, se non la sola informazione relativa alla presenza o assenza dell'evento rilevato dai sensori. Il segnale di uscita può essere rappresentato invece con una funzione:

$$c: \mathbb{R} \longrightarrow \{\text{absent}\} \cup \mathbb{N}$$

questo non è un segnale puro, avendo contenuto informativo, anche se come u e d, può essere presente o assente.
La dinamica dei sistemi discreti come questo può essere descritta da una sequenza di step (passi) chiamati reazioni, ognuno dei quali si presuppone sia istantaneo, ovvero abbia durata nulla. Le reazioni in un sistema discreto sono scatenate (triggered) dall'ambiente in cui opera il sistema: quando sono scatenate da eventi di input allora si dicono event-triggered. Nell'esempio del parcheggio, le reazioni da parte del Counter sono scatenate quando sono presenti uno o più eventi di input relativamente all'arrivo o dipartita di un auto. Quando entrambi gli input sono assenti, nessuna reazione avviene.
L'esecuzione di una reazione comporta la valutazione degli input e degli output:

- ad ogni segnale in input si associa una variabile a cui viene assegnato il valore della funzione che rappresenta il segnale in quell'istante

- stessa cosa vale per i segnali di uscita: per ogni segnale di uscita viene assegnato il valore della funzione in quell'istante
- i valori possono includere anche "absent", ad intendere che non è presente alcun segnale

Intuitivamente lo stato di un sistema è la condizione in cui si trova in un certo istante temporale. Formalmente, lo stato rappresenta tutto ciò che è successo nel passato che ha un effetto nel determinare la reazione del sistema agli input correnti e futuri. Nel caso di un sistema Counter nell'esempio, lo stato state(t) al tempo t è rappresentabile con un intero compreso fra 0 ed M, dove M è il numero massimo di posti. Definito l'insieme States = {0, 1, 2, .. M}, allora lo stato è modellabile come una funzione

state: $\mathbb{R} \longrightarrow$ States

Una macchina a stati è un modello di sistema a dinamica discreta, in cui ogni input del sistema viene mappato in un output a seconda del suo stato corrente, o meglio ad ogni reazione le valutazioni degli input vengono mappate in valutazioni degli output secondo lo stato corrente. Una macchina a stati finiti, FSM, è una macchina a stati in cui il numero degli stati è finito. Se il numero degli stati è ragionevolmente ridotto, si può rappresentare graficamente con diagrammi di stato.
Le transizioni fra stati sono ciò che rappresentano l'evoluzione dinamica discreta, ovvero il comportamento, di una macchina a stati. Una transizione coinvolge sempre due stati e può essere rappresentata da due elementi:

- guardia: è un predicato (ovvero una funzione booleana) sulle variabili di input e specifica le condizioni per cui la transizione è abilitata, ovvero può avvenire. Se in una reazione nessuna guardia è abilitata, la FSM rimane nello stato in cui si trova
- azione: specifica il valore che devono assumere le variabili di output come risultato della transizione. Se una variabile di output non viene specificata, si assume implicitamente che assuma il valore absent.

Una FSM non specifica di per sé quando una reazione deve avvenire, ovvero quando dev'essere operata la valutazione degli input che porta a far scattare le transizioni. Ci sono due possibilità:

- FSM asincrone (event-triggered): in questo caso la reazione avviene a fronte di un evento di input, quindi è l'ambiente in cui opera la macchina che stabilisce quando la valutazione e quindi le reazioni devono avvenire
- FSM sincrone (time-triggered): in questo caso le reazioni avvengono ad intervalli regolari di tempo (è definito un periodo e quindi una frequenza di funzionamento).

Nell'applicazione delle macchine a stati ai sistemi embedded, in generale l'I/O è modellato in termini di variabili a cui la macchina può accedere:

- variabili di input: modificate dall'ambiente in cui opera la macchina
- variabili di output: modificate dalla macchina stessa mediante azioni, come il controllo dell'ambiente
- oltre all'I/O, le variabili possono essere usate come modo più flessibile per caratterizzare lo stato a livello globale (accessibili da tutte le azioni e condizioni)

Una transizione avviene a fronte del verificarsi di determinate condizioni, come predicati/funzioni booleane sul valore delle variabili. Ad esempio citiamo il blinking e il button-led.
Aspetto importante della letteratura e della prassi dello sviluppo dei sistemi embedded classici è il mapping:

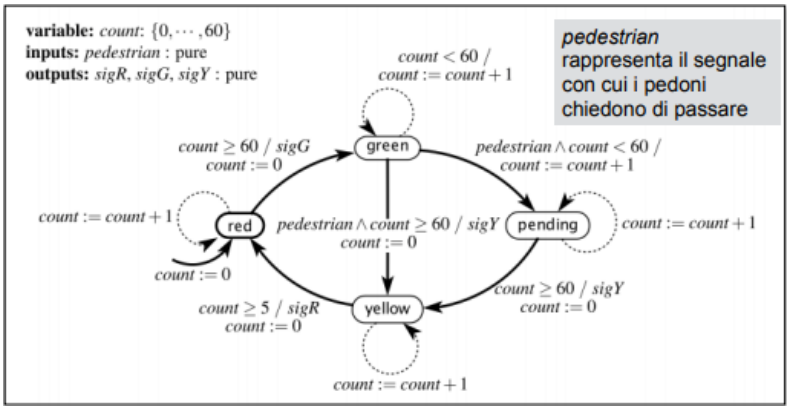
- si cattura e descrive il comportamento con una macchina a stati
- poi lo si converte in codice da eseguire sul microcontrollore
- la conversione è automatizzabile (model-driven engineering)

Nella conversione c'è una separazione fra la parte che rimane nel loop e la funzione che rappresenta lo step della macchina. In dettaglio, si ha:

- una rappresentazione esplicita degli stati
- una variabile che tiene traccia dello stato corrente
- step() della macchina nel main loop caratterizzato da un costrutto di selezione sullo stato corrente, con un ramo per ogni stato
- nell'implementazione delle macchine sincrone si fa in modo che la selezione avvenga solo ogni specificato periodo di tempo

Per il corretto funzionamento e codifica di una macchina a stati ogni azione deve sempre terminare, ossia non ci devono essere loop infiniti, e la valutazione di una condizione non deve cambiare lo stato delle variabili.

Non appena la FSM diviene articolata, si considera un'estensione in cui si usano delle variabili come parte dello stato, ovvero fornendo una rappresentazione intensionale dello stato. In questo caso la parte delle azioni include delle set actions, con cui si assegna il valore della variabile di stato. Prendiamo ad esempio una FSM che descrive un semaforo pedonale (a destra): si presuppone sia time triggered e che il periodo sia 1 secondo, quindi reagisce una volta al secondo.



Da notare che il numero reale degli stati di una EFSM include tutti quelli che derivano da ogni possibile configurazione ammissibile delle variabili di stato, ossia |States| = $n*p^m$, dove n è il numero degli stati discreti, m è il numero di variabili che descrivono lo stato e p sono i possibili valori che può assumere ogni variabile.

Le macchine a stati descritte in precedenza sono chiamate anche Machine di Mealy. Un approccio alternativo è dato dalle Macchine di Moore, in cui l'output è associato allo stato, ed è prodotto quando si transita nello stato stesso.

Molti comportamenti nei sistemi embedded sono time-oriented: non solo prevedono un ordinamento temporale delle azioni, ma includono la specifica di intervalli temporali nella definizione del comportamento (ad esempio il led blinking). Le macchine a stati sincrone sono macchine a stati estese con supporto nativo del trascorrere del tempo per agevolare la specifica di comportamenti time-oriented: in particolare in queste macchine lo step di una macchina a stati viene eseguito periodicamente, secondo un certo periodo. È come se la macchina a stati avesse un clock interno e le transizioni vengano eseguite solo al verificarsi di un tick del clock.

La descrizione del comportamento di una macchina a stati sincrona in linguaggi procedurali come il C su sistemi a microcontrollore tipicamente si basa sull'uso dell'interruzione di Timer programmabili, che sono inclusi nel microcontrollore o sulla scheda e permettono di generare un segnale di interruzione (del timer) periodico.

Un sistema può avere intervalli temporali di lunghezza diversa da gestire e il comportamento può essere rappresentato da opportune macchine a stati sincrone: come periodo si sceglie il MCD degli intervalli caratterizzanti il sistema. Ad esempio, nel caso del blinking, se il led è acceso per 500ms e spento per 750 ms si sceglie come periodo 250 ms.

Considerando macchine a stati sincrone, la lettura periodica di un sensore ad una data frequenza (ovvero con un certo periodo) viene definito campionamento (sampling): è caratterizzata da un certo periodo di campionamento; per frequenza di campionamento si intende il valore inverso del

periodo, ovvero il numero di volte al secondo in cui avviene (in Hz). La scelta del periodo è un aspetto molto importante della progettazione, poiché dev'essere sufficientemente piccolo da evitare di perdere eventi. Maggiore è la frequenza (e minore è il periodo), e:

- da un lato, maggiore è la reattività
- dall'altro, è maggiore l'utilizzo (spesso inutile) del microcontrollore e quindi il consumo di energia (e questo è un aspetto molto importante nei sistemi embedded)

Quindi, in generale, il periodo dovrebbe essere scelto in modo da essere il più grande possibile, per minimizzare il consumo, a partire dai valori che tuttavia garantiscono il corretto funzionamento del sistema. In sintesi, il valore del periodo dev'essere il più grande fra i valori sufficientemente piccoli da garantire il corretto funzionamento del sistema.

Il MEST (Minimum Event Separation Time) è l'intervallo più piccolo che, dato l'ambiente in cui opera il nostro sistema, può esserci fra due eventi di input. Secondo il teorema, in una macchina a stati sincrona, scegliendo un periodo inferiore al MEST si garantisce che tutti gli eventi saranno rilevati. Notiamo che il MEST di eventi di input come pulsanti non include solo gli eventi in sé di pressione del pulsante, ma l'intervallo di tempo fra tali eventi. Se le pressioni di un pulsante richiedono di essere separate da intervalli di 300 ms, allora sampling rate maggiori di 300 ms possono risultare nel non rilevamento dell'evento di rilascio.
Alcuni eventi di input possono dover scatenare nuovi eventi di output o informazioni in output. L'intervallo che trascorre tra l'evento di input e la generazione dell'output è detta latenza: ad esempio, nel sistema button-led, il tempo che intercorre dalla pressione del pulsante all'accensione del led è una latenza. L'obiettivo usuale è minimizzare le latenze: questo influisce sulla scelta del periodo, infatti più piccolo è il periodo e più si riducono le latenze. Ad esempio, per il button-led, 300 ms di latenza non sarebbe accettabile, poiché la latenza sarebbe percepibile come ritardo da una persona: in questo caso 50 ms invece è un valore accettabile.
I sensori in generale possono presentare imperfezioni (a livello hardware) che richiedono aggiustamenti (input conditioning, condizionamenti dell'input) dei valori letti da parte del microcontrollore. Un classico esempio è il bouncing dei pulsanti: il meccanismo interno a molti pulsanti, a fronte della pressione, può portare ad avere un fenomeno fisico simile al rimbalzo (meccanico), per cui il segnale corrispondente passa da un valore LOW ad un valore HIGH e viceversa più volte, prima di assestarsi sul valore HIGH. In questo caso, se la frequenza di campionamento del segnale è molto alta, allora il sistema può inferire non correttamente che il pulsante sia stato premuto più volte.
Per button debouncing si intende la funzionalità di ignorare i rimbalzi (bouncing) spuri del pulsante, in modo che una singola pressione sia rilevata. Il problema si può risolvere via hardware oppure a livello software, considerando una frequenza di campionamento inferiore alla frequenza del bouncing, ovvero un periodo superiore a quello del rimbalzo. Nel caso concreto del pulsante, i pulsanti rimbalzano non più di 10-20 ms, per cui un periodo di 50 ms va bene.
Generalizzando il caso del bouncing, l'input conditioning può avvenire mediante tecniche di filtraggio utili per ignorare la presenza di eventi di input spuri: i segnali spuri, detti glitches o spikes, possono esserci anche a causa delle condizioni ambientali. Una tecnica di filtraggio consiste nella riduzione del periodo di campionamento: questo può ridurre la probabilità di rilevare segnali spuri, tuttavia non la elimina, poiché un glitch può essere sempre rilevato se coincide esattamente con il momento in cui viene fatto il rilevamento. Una soluzione più generale in questo caso consiste nel richiedere che il segnale rilevato che rappresenta uno specifico evento, per essere considerato tale, abbia una certa durata minima: può corrispondere, ad esempio, a una serie di più campionamenti e considerare che il periodo di campionamento sia maggiore della durata di un glitch. Per esempio, per un periodo di campionamento di 50 ms di un pulsante e glitch che durano al più 10 ms, possiamo

richiedere che una serie di 2 campionamenti siano rilevati a HIGH per poter considerare il pulsante premuto. Questo tuttavia ha un impatto sul MEST, riducendo (a metà) l'intervallo che può intercorrere fra due eventi. Anche questa soluzione di per sé non elimina del tutto la possibilità di interpretare glitch come eventi: ad esempio 2 glitch che avvengono nel momento in cui c'è il campionamento sono interpretati come evento. Inoltre tutto ciò introduce latenze.

ARCHITETTURE BASATE SU TASK CONCORRENTI

La modellazione e progettazione di sistemi software embedded articolati può essere complessa: si ha la necessità di approcci opportuni per decomporre/modularizzare il comportamento e le funzionalità. Un approccio utilizzato nel contesto della programmazione concorrente è la decomposizione del comportamento complessivo in più task: ogni task rappresenta un compito ben definito, un'unità di lavoro da svolgere. Il comportamento di ogni task può essere descritto da una opportuna FSM, e il comportamento complessivo è l'insieme delle FSM.
La decomposizione in task è un principio di progettazione importante, che permette di rendere modulare il sistema:

- ogni modulo è rappresentato da un task
- un task può essere decomposto in sotto-task (ricorsivamente), o equivalentemente un task complesso può essere definito come composizione di sotto-task più semplici
- le FSM possono essere usate per rappresentare il comportamento associato ad un task "atomico", che in questo caso significa non ulteriormente decomponibile

Quali sono i vantaggi?

- modularità
- separazione dei concetti
- chiarezza
- manutenibilità
- estendibilità
- riusabilità

Ci sono dei problemi? Si, come prima considerazione i task sono concorrenti, quindi la loro esecuzione si sovrappone nel tempo, inoltre i task possono avere dipendenze che richiedono di essere gestite mediante opportune forme di coordinazione:

- cooperative: ad esempio la comunicazione e la sincronizzazione
- competitive: ad esempio la mutua esclusione e le sezioni critiche

Nei sistemi embedded basati su microcontrollore (e senza OS) un primo modo semplice di realizzare task concorrenti è come semplice estensione del caso a singolo task per cui, anziché avere un'unica procedura tick (o step) che viene richiamata ad ogni ciclo, in questo caso ci sono più funzioni tick, una per ogni task. Considerando poi una modellazione ad oggetti, modelleremo ogni task, in cui è definito il metodo step, specializzato in ogni classe concreta: quindi nel programma avremo una sola istanza di questa classe (pattern singleton).
Nel caso di FSM con periodi diversi è necessario:

- tenere traccia per ogni task anche del periodo specifico
- far funzionare la macchina a stati con il periodo pari al MCD
- tener traccia per ogni task del tempo trascorso e nel caso si sia raggiunto il valore del periodo, richiamare il tick

Il tipo di multitasking che possiamo adottare è quello cooperativo (altrimenti preemptive), con scheduling di tipo round-robin. I vantaggi sono che non c'è interleaving fra le istruzioni di task diversi (tick eseguita atomicamente rispetto a task diversi) e non ci possono essere corse critiche. Gli svantaggi sono che il comportamento errato di un task può compromettere l'esecuzione degli altri (ad esempio un loop infinito nel tick di un task manda in starvation gli altri task). La generalizzazione dell'approccio porta alla definizione di un vero e proprio scheduler, in cui si tiene traccia mediante un'opportuna struttura dati (ad esempio una coda) della lista dei task in esecuzione. Nel main loop

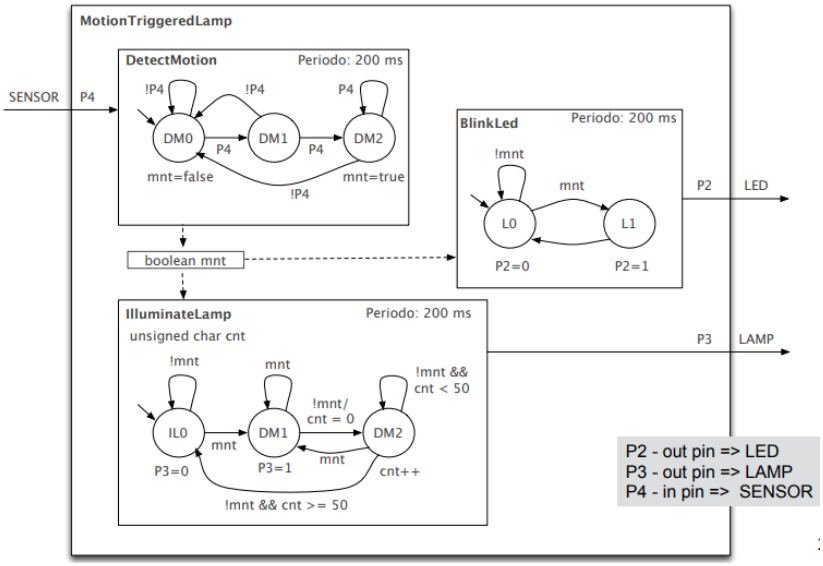
si visita ogni elemento in coda, richiedendone il tick nel caso in cui il tempo trascorso sia pari al periodo previsto dal task.
Non sempre è possibile suddividere task in sottotask totalmente indipendenti, anzi è più frequente il caso in cui i sottotask abbiano delle forme di dipendenza di vario tipo:

- temporale: ad esempio un task T3 può essere eseguito solo dopo che sono stati eseguiti i task T1 e T2
- produttore/consumatore: ad esempio un task T1 ha bisogno di un'informazione prodotta da un altro task T2
- relative a dati: ad esempio due task T1 e T2 necessitano di condividere dati (in lettura o in scrittura)

Il modello più semplice che si può utilizzare per rappresentare queste dipendenze in un sistema embedded è mediante variabili condivise.

Consideriamo l'esempio di un motion-triggered lamp system, un sistema che deve accendere una luce a fronte del rilevamento di movimenti in un certo ambiente:

- è un sistema dotato di un certo sensore di movimento collegato ad un certo pin (es. P4)
- il movimento dev'essere rilevato a fronte del fatto che P4 sia HIGH per due volte consecutive, considerando il periodo di campionamento di 200 ms
- quando il movimento è rilevato, il sistema deve illuminare una lampada (led) collegata al P3. e mantenere la lampada settata per 10 secondi dopo l'ultimo rilevamento di movimento
- il sistema dovrebbe inoltre far lampeggiare un led connesso a P2 con periodo di 200 ms per tutto il tempo in cui il movimento è rilevato



Questo compito complesso può essere facilmente espresso come composizione di tre task (come descritto dall'immagine a destra):

- DetectMotion
- IlluminateLamp
- BlinkLed

I vantaggi rispetto al caso monolitico sono:

- separazione dei concetti: ogni task è focalizzato su un compito ben preciso
- comprensibilità: è più semplice capire il comportamento dei singoli moduli, meno complessi
- debugging: si ha una localizzazione più rapida degli errori
- modificabilità, estensione, riuso: il raffinamento del comportamento di un modulo non richiedono di mettere mano agli altri moduli

La presenza di variabili condivise fra i task richiede di porre attenzione al problema del loro accesso concorrente. È noto che accessi concorrenti alla medesima variabile non danno problemi se sono in lettura, invece possono insorgere corse critiche nel caso di accessi concorrenti in scrittura o lettura/scrittura. il problema si può risolvere in generale presupponendo che le transizioni di ogni FSM siano eseguite atomicamente, come pure le eventuali azioni associate ad uno stato: in altre parole non dev'esserci interleaving delle azioni di cui si possono comporre transizioni o stati di FSM concorrenti (nello stesso sistema embedded).

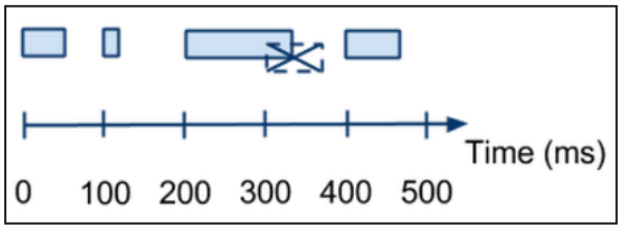
In un modello a task per FSM sincrone, i task comunicano tipicamente con l'uso di variabili globali condivise. In alternativa, come approccio avanzato è possibile adottare un modello a scambio di messaggi asincrono:

- ogni task ha la propria coda/buffer di messaggi
- la comunicazione fra task avviene inviando messaggi in modo asincrono
- approccio critico in termini di risorse di memoria richieste

Nel caso di interrupt-driven cooperative scheduler, similmente a quanto accade nei sistemi operativi, lo scheduler può essere invocato direttamente dall'interrupt handler del timer: ciò conduce ad una versione totalmente reattiva del sistema, dove nel loop non è necessario svolgere alcun compito. In questo caso è possibile pensare di sfruttare facilmente le modalità di funzionamento a basso consumo dei microcontrollori, per cui nel loop si pone il microcontrollore in stato di sleep, da cui esce all'arrivo di interruzioni. Questo porta a sistemi con un consumo di energia nettamente inferiore, quindi particolarmente significativo per sistemi embedded alimentati da batterie, che devono durare a lungo (ad esempio reti di sensori). Occorre fare attenzione in questo caso però al tempo impiegato dal micro per entrare e uscire da tale modalità, poiché potrebbe introdurre latenze che non sono compatibili con il periodo scelto e con i task che devono essere eseguiti.

Nelle FSM abbiamo fatto l'assunzione (come modello) che le azioni eseguite avessero un tempo di esecuzione nullo, o comunque inferiore al periodo della macchina. Per sistemi con molte azioni in uno stato o transizione, oppure con molti task che condividono lo stesso microcontrollore, tale assunzione può non essere sempre plausibile: è necessario considerare la durata delle azioni eseguite. In tal caso, lo sviluppatore di sistemi embedded deve porre attenzione alla reale durata delle azioni in relazione al periodo per fare in modo che non ci siano malfunzionamenti.

Chiamiamo eccezione di overrun la situazione per la quale il tempo di esecuzione delle azioni oltrepassa il periodo: si parla di timer overrun nel caso in cui consideriamo uno scheduler interrupt-driven in cui le interruzioni possono essere annidate, per cui una nuova interruzione può essere generata prima che l'interrupt handler di un triggering precedente abbia concluso la propria esecuzione. A destra un esempio di overrun. Il parametro di utilizzo di un microcontrollore è dato dalla percentuale di tempo in cui è utilizzato per eseguire dei task:



U = (tempo utilizzo per task / tempo totale) * 100%

Nel modello che possiamo usare per fare l'analisi dell'utilizzo del processore possiamo trascurare le istruzioni che fanno parte dello scheduler. Il Worst-Case-Execution-Time (WCET) è dato dal tempo di esecuzione nel caso peggiore in termini di numero di istruzioni eseguite ad ogni periodo della macchina sincrona. Nel caso di più stati, si considera lo stato/transizione con la sequenza di condizioni/azioni più lunga: va considerato nel calcolo anche il tempo per valutare le condizioni.
Una analisi che conduce ad ottenere un parametro di utilizzo U > 100% indica che si verificherà un eccezione di overrun. In tal caso, le azioni da intraprendere sono:

- utilizzare macchine a stati con periodi più lunghi
- ottimizzare la sequenza di condizioni/azioni più lunga
- spezzare sequenze lunghe fra più stati
- usare un microcontrollore più veloce
- eliminare funzionalità dal sistema

L'analisi dell'utilizzo del processore è ancora più frequente nel caso di sistemi a più task. Nel caso semplice di task con lo stesso periodo, allora il WCET si calcola come somma dei singoli WCET dei task.
Nel caso più generale, i task possono avere periodi diversi. In tal caso, il calcolo del WCET dev'essere fatto considerando degli iperperiodi, ossia periodi che siano il MCM fra i periodi considerati.
In sintesi, il calcolo di U nel caso di T1..Tn task su un microcontrollore M è dato dai seguenti passi:

- si determina il sec/instr rate R, ovvero quanti secondi servono per eseguire una istruzione del micro si analizza ogni task Ti, determinando il suo WCET, determinando prima il numero massimo di istruzioni per tick e moltiplicando tale numero per R
- si determina l'iper-periodo H come minimo comune multiple fra i periodi di tutti i task (T1.H, T2.H,…)
- il parametro di utilizzo è calcolato allora come:
  U = ((H/T1.period)*T1.WCET + (H/T2.period)*T2.WCET+...)/H*100
  - U > 100 => ci sarà overrun
  - U < 100 => singolo task: overrun non può accadere, task multipli: overrun può accadere

Si definisce jitter il ritardo che intercorre dal momento che un task è pronto per essere eseguito e il momento in cui viene effettivamente eseguito. Strategie di scheduling diverse possono portare a jitter diversi. In generale, dare priorità ai task con periodo più piccolo porta a minimizzare il jitter medio.
Si definisce deadline l'intervallo di tempo entro il quale un task dev'essere eseguito dopo essere diventato ready. Se un task non viene eseguito entro la sua deadline, si ha un'eccezione di missed-deadline che può portare a malfunzionamenti del sistema. Se dato un task non viene specificata la sua deadline, allora per default la deadline è data dal periodo, ovvero il tempo entro il quale il task torna ad essere ready. A seconda della strategia di scheduling adottata, si possono avere jitter diversi e soddisfare o meno le deadline.
Per priorità in generale si intende l'ordinamento con cui devono essere eseguiti i task: quando ci sono più task ready, allora lo scheduler sceglie quello prioritario. In generale gli scheduler possono essere suddivisi in scheduler a priorità:

- statica: la priorità viene assegnata ad ogni task prima che i task siano eseguiti e queste non cambiano durante l'esecuzione.
  Un approccio frequente in questo caso è di assegnare la priorità maggiore ai task con la deadline più piccola: l'intuizione in questo caso è che i task possono non rispettare la deadline se non eseguiti prima.
  Se tutti i task hanno le deadline pari ai periodi dei task, allora questo approccio risulta nell'assegnare la priorità maggiore ai task con il periodo più piccolo: questo scheduling è chiamato rate-monolitic scheduling (RMs), ed è chiamato così perché le priorità sono basate sul rate del task, con priorità assegnate in modo monotono crescente.
  Un altro approccio assegna la priorità al task con WCET più piccolo: è utile per ridurre il jitter, tuttavia richiede per essere applicato la conoscenza del WCET, che non sempre è noto.
  Nello scheduler visto, la priorità dei task era data implicitamente dall'ordine in cui erano inseriti nell'array dei task da eseguire.
- dinamica: questo tipo di scheduler determina le priorità dei task man mano che il programma viene eseguito, per cui le priorità assegnate possono cambiare nel tempo. L'approccio più utilizzato in questo caso è denominato earliest-deadline-first (EDF): l'intuizione è che i task ready che hanno la deadline più vicina siano quelli che possono violarla se non vengono eseguiti subito. La schedulazione a priorità dinamica tipicamente

riduce jitter e missed deadline, tuttavia richiede scheduler più complessi e costosi dal punto di vista delle computazioni. EDF è usato spesso anche quando i task sono eseguiti in reazione ad eventi, anziché time-oriented.

Uno scheduler può essere:

- preemptive: è possibile togliere il processore ad un task in esecuzione prima che abbia completato (sono più difficili da implementare)
- cooperativo: una volta selezionato, un task esegue fino al completamento

Il caso preemptive è tipicamente utilizzato quando i task sono organizzati in termini di processi, con ciclo infinito nel caso in cui siano periodici. L'organizzazione a stati vista con le FSM sincrone può essere realizzata con scheduling cooperativo: ad ogni tick della macchina vengono eseguite atomicamente le azioni associate all'eventuale transizione/nuovo stato.

Le FSM viste finora per rilevare cambiamenti sull'input eseguono campionamenti periodici (polling). Un approccio per ridurre l'utilizzo della CPU (sprecato in attese attive) consiste nello sfruttare supporti HW che permettono di rilevare cambiamenti su pin di input e quindi generare interruzioni, chiamando conseguentemente apposite interrupt handler (FSM event-triggered). Quindi il micro può essere fatto transitare in uno stato di inattività fin quando non viene rilevato l'evento che le fa transitare in uno stato attivo. A livello di modello, è possibile modellare questa situazione con l'introduzione di uno speciale stato "inactive", in cui la macchina è in idle. Alla ricezione di un evento relativo ad un pin in ingresso, la macchina esce dallo stato inactive e inizia ad eseguire il comportamento sincrono periodico, fino ad entrare nuovamente in uno stato inactive.
L'integrazione di approcci event-triggered può ridurre in modo significativo l'utilizzo del processore: nel caso triggered il comportamento periodico si ha solo quando la FSM è in uno stato attivo.
È possibile usare un modello alternativo per integrare modello a stati con interruzioni/eventi. Si abbandona il modello sincrono e si assume un modello asincrono, dove le transizioni di stato sono determinate da guardie eventi/condizioni (FSM event-driven). Tuttavia fra gli eventi si assumono quelli del timer, per cui il modello sincrono è modellato come macchina che reagisce ad eventi periodici "tick" a frequenza prefissata. In questo modo non è necessario inserire lo stato "inattivo" esplicitamente nel diagramma degli stati.

Finora sono stati considerati solo task periodici, ossia task che vengono mandati in esecuzione periodicamente dallo scheduler. Più in generale, può essere utile considerare l'esecuzione anche di task aperiodici, ossia che vengono mandati in esecuzione a tempi arbitrari. Le gestione di task aperiodici complica in modo significativo lo scheduler. Approcci semplificati sono:

- task aperiodici realizzati come task periodici con stato "idle"
- task con meta-stato attivo o non-attivo, selezionato dallo scheduler solo se si trova in uno stato attivo
- introdotti staticamente e attivati/disattivati da altri task

Architetture ad eventi

A basso livello le interruzioni di input rappresentano eventi che richiedono l'attenzione del processore. Permettono di realizzare strategie di gestione dei sensori che evitano l'uso di polling, ovvero evitano che sia il programma/processore a dover continuamente interrogare il sensore per conoscere il valore corrente ed in particolare se ci sono stati cambiamenti. Concettualmente è il sensore che notifica il cambiamento al programma. Possiamo sfruttare le interruzioni per realizzare architetture/pattern ad eventi di alto livello

- pattern observer: è composto da diversi elementi

    - sorgente o generatore di eventi
    - eventi generati
    - osservatori

  È implementato mediante interruzioni, utilizzate all'interno dei componenti generatori per fare in modo che all'occorrenza dell'interruzione (associata ad un certo evento) vengano chiamati gli ascoltatori registrati sul componente. Il codice dell'osservatore viene mandato in esecuzione direttamente dall'interrupt handler, ad interruzioni disabilitate, per cui non deve contenere task computazionalmente lunghi. Per evitare corse critiche è necessario rendere atomica l'esecuzione dei blocchi di codice eseguiti dal super loop in cui si accede a variabili modificate dagli osservatori, disabilitando e abilitando le interruzioni.
- FSM asincrone: a differenza delle macchine sincrone, la valutazione delle reazioni avviene a seguito di un evento, generato mediante interrupt. Non c'è la nozione di periodo, come nelle FSM sincrone. Possiamo eventualmente rappresentare eventi temporali come eventi generati da un timer. In questo caso c'è disaccoppiamento fra generazione degli eventi (gestito a interrupt) e reazione della FSM (eseguita da flusso di controllo del super loop): il codice della FSM asincrona non è così soggetto alle limitazioni relative agli interrupt handler, pur mantenendo la reattività del sistema; nel codice della FSM asincrona non è necessario disabilitare/riabilitare le interruzioni, dal momento che non si accede direttamente a variabili che possono essere modificate dagli interrupt handler.

SISTEMI EMBEDDED BASATI SU SOC E RTOS

I SOC (System-on-a-chip) sono chip che incorporano un sistema completo, e sono tipicamente usati per creare delle single-board CPU. Un SOC comprende:

- uno o più processori/core
- un modulo di memoria contenente uno o più blocchi di tipo ROM, RAM, EEPROM o memoria flash
- un generatore di clock
- periferiche come contatori, orologi e altro
- connettori per interfacce standard, come USB, Ethernet, USART, I^2C e SPI
- DAC e ADC
- regolatori di tensione e circuiti di gestione dell'alimentazione

L'ESP 8266 è un dispositivo pensato per l'IoT. Ha un processore L106 a 32 bit RISC funzionante a 80 MHz, una RAM per le istruzioni da 64 KB e una RAM per i dati da 96 KB, 16 pin GPIO con supporto SPI, I^2C, I^2S + UART, ADC a 10 bit, IEEE 802.11 b/g/n Wi-Fi.
I sistemi embedded basati su SoC hanno risorse sufficienti (in termini di capacità computazionale e memoria) da poter montare un vero e proprio sistema operativo. Si parla si Sistemi Operativi Embedded e Real-Time (RTOS), ossia sistemi operativi progettati per essere usati nei sistemi embedded.

Richiami di sistemi operativi

Il sistema operativo (OS o SO) è quella parte software di un sistema di elaborazione che controlla l'esecuzione dei programmi applicativi e funge da intermediario fra questi e la macchina fisica (hardware). Gli obiettivi principali del SO sono:

- eseguire programmi degli utenti e controllare la loro esecuzione, in particolare l'accesso alla macchina fisica (SO come extended/virtual machine, svolge una funzione di astrazione, controllo e protezione)
- rendere agevole ed efficace l'utilizzo delle risorse del computer (SO come resource manager, svolge una funzione di ottimizzazione)
- abilitare e coordinare le interazioni fra applicazioni, utenti e risorse (più applicazioni in esecuzione concorrente, più utenti che condividono e usano simultaneamente il sistema)

Il SO media l'interazione fra livello applicazione e livello hardware, controllando e coordinando l'accesso e l'uso del livello hardware richiesto dal livello applicazione, e rendendo i due livelli il più possibile indipendenti fra loro. Un ruolo fondamentale è svolto dalle API, che sono un'interfaccia di programmazione per i programmi e contengono un insieme di funzioni o primitive di base chiamate system calls.
Il SO maschera ai programmi applicativi la struttura reale della macchina fisica, facendo veder loro una macchina virtuale:

- una macchina più semplice e ad un livello di astrazione maggiore rispetto al livello fisico
- accessibile mediante le system calls che nell'insieme costituiscono l'interfaccia della macchina virtuale
- vista top-down

I benefici della macchina virtuale sono:

- agevolare la progettazione e programmazione delle applicazioni: il programmatore non deve conoscere necessariamente i dettagli specifici della macchina fisica per interagire con le risorse e può concentrarsi sulle funzionalità del programma
- aumentare il livello di portabilità dei programmi: il medesimo programma può essere messo in esecuzione su macchine virtuali con la medesima interfaccia che girano su macchine fisiche diverse; semplificazione del porting di un programma da un sistema operativo ad un altro

Il SO è gestore di risorse:

- condivise e utilizzate da più programmi, anche concorrentemente
- eventualmente appartenenti a più utenti che utilizzano la stessa macchina

Ha quindi funzionalità di:

- ottimizzazione degli accessi, con algoritmi di scheduling
- protezione e sicurezza, per evitare che un programma in esecuzione possa provocare malfunzionamenti al sistema e ad altri programmi in esecuzione, e garantire protezione e sicurezza dei dati degli utenti

La virtualizzazione delle risorse avviene tramite:

- memoria virtuale: si fa in modo che un programma in esecuzione veda la memoria a disposizione come uno spazio lineare virtualmente illimitato, e si fa un utilizzo trasparente della memoria di massa come estensione di quella principale
- file system virtuale: accedere e modificare file nel file system in modo uniforme a prescindere dal tipo specifico di file system e dall'effettiva locazione dei file stessi (locali o remoti)

L'esecuzione di programmi è gestita con:

- multiprogrammazione: capacità di caricare in memoria centrale più programmi che vengono eseguiti in modo da ottimizzare l'utilizzo della CPU. Se un programma in esecuzione è impegnato in un'operazione di I/O e non ha bisogno della CPU, la CPU viene allocata ad un altro programma in esecuzione. Permette una prima forma di multitasking, ovvero di esecuzione concorrente di più programmi.
- multitasking: estensione logica della multiprogrammazione funzionale all'esecuzione di programmi che richiedono interazione con l'utente. È la capacità di eseguire più programmi concorrentemente (a prescindere dalle operazioni di I/O), con condivisione della CPU fra i programmi in esecuzione secondo determinate strategie di schedulazione.

La modalità di funzionamento è interrupt-driven: entra in esecuzione in seguito ad eventi che sono associati all'occorrenza di:

- interruzioni hardware: sono segnali inviati da dispositivi, e sono asincrone
- trap software: sono richieste da parte dei programmi di eseguire servizi (system calls), oppure eccezioni generate da errori nei programmi, e sono sincrone

Sfrutta due modalità operative distinte della CPU:

- user-mode: esecuzione dei programmi utente
- kernel-mode: esecuzione delle parti di programma del sistema operativo

Riassumendo, le attività principali del SO sono:

- gestione dei processi
- gestione della memoria
- gestione della persistenza delle informazioni
    - gestione del file system
    - gestione dei dischi
    - caching

- gestione dell'I/O
- gestione della rete
- gestione della protezione e sicurezza
- gestione utenti

Sistemi operativi embedded e real-time

I Sistemi Operativi Embedded e Real-Time (RTOS) sono sistemi operativi progettati per essere usati nei sistemi embedded. Le loro caratteristiche in generale sono:

- compattezza
- gestione estremamente efficiente delle risorse
- affidabilità

Rispetto ai sistemi operativi desktop spesso sono progettati per mandare in esecuzione una sola applicazione per volta, che tuttavia è tipicamente multi-task/multi-thread.
I sistemi real-time:

- devono essere in grado di reagire/rispondere ad eventi e input entro limiti di tempo prestabiliti
- devono eseguire task/computazioni entro determinati vincoli temporali
- hanno la nozione di deadline

Le sottoclassi sono:

- hard real-time, in cui le deadline devono essere sempre rispettate
- soft real-time, in cui le deadline devono essere rispettate in condizioni normali, ma sono ammessi casi in cui non vengono rispettate

Il determinismo è un aspetto importante di molti sistemi real-time, in particolari quelli hard real-time. È una proprietà per cui dev'essere predicibile:

- il tempo impiegato per svolgere un certo task
- il tempo massimo richiesto per eseguire una certa azione o acquisire un certo valore in input o da un sensore o a rispondere ad una certa interruzione
- il numero di cicli richiesti per eseguire una certa operazione dev'essere sempre lo stesso

In questi sistemi l'esecuzione può essere interrotta (interruzioni), tuttavia l'overhead relativo (latenza, tempo di elaborazione dell'interruzione) dev'essere limitato e noto. Non tutti i RTOS sono in grado di fornire garanzie in merito.
La struttura di un RTOS è simile a quella di OS normale:

- kernel
- file system
- networking
- USB
- graphics

Le attività/compiti del kernel sono:

- gestione e scheduling dei task
- sincronizzazione e comunicazione inter-task
    - semafori
    - code/scambio messaggi
- gestione timer
- gestione dispositivi I/O
    - efficienza
    - protezione

    - condivisione
- gestione memoria
- gestione interruzioni ed eventi

Quali sono i benefici di un RTOS?

- migliorare la responsiveness e diminuire l'overhead
- semplificare la condivisione delle risorse
- semplificare lo sviluppo, debugging e maintenance (sistemi software embedded complessi)
- aumentare la portabilità
- abilitare l'uso di servizi e middleware stratificati
- rendere più veloce il time-to-market delle applicazioni

I programmi basati su sistemi senza RTOS sono basati tipicamente su un loop di controllo in cui si adotta il polling come strategia per verificare la necessità di eseguire determinate funzioni/task:

- il numero di application function/task determina il tempo complessivo di questo polling
- il tempo necessario per svolgere un servizio dipende da questo polling time
- looping, checking, polling, state machine tracking sono attività che consumano cicli del processore e aggiungono overhead

Un RTOS in generale permette di:

- evitare polling/looping sfruttando la possibilità di eseguire "context switch" al processore, da un task ad un altro, in modo trasparente all'applicazione
- realizzare in modo trasparente alle applicazioni multi-tasking, sfruttando il processore anche quando l'applicazione è in waiting e usando il multithreading
- dedicare maggior tempo del processore all'applicazione, con meno tempo speso alla gestione delle applicazioni

La gestione delle risorse è condivisa da più funzioni/task (memoria, porte I/O, sezioni critiche). Gli RTOS sono meccanismi centralizzati per gestire/arbitrare le richieste da task diversi:

- allocazione/deallocazione della memoria a runtime
- semafori/mutex per controllare l'accesso a risorse HW o a sezioni critiche
- meccanismi per evitare problemi dovuti a interferenze fra priorità e sezioni critiche

In un RTOS lo sviluppatore non deve gestire aspetti di basso livello come interruzioni, timer ecc.
Un RTOS abilita l'utilizzo di framework/piattaforme/middleware che forniscono servizi, spesso a livelli diversi: ad esempio il file system, TCP/IP, USB stack. Un'applicazione usa le RTOS API, non accede direttamente all'HW specifico: questo permette di poter cambiare HW trasparentemente, e che l'applicazione possa essere mandata in esecuzione in modo trasparente su qualsiasi HW supportato dal RTOS. Quando non conviene usare un RTOS? Con applicazioni simple looping o polling o applicazioni single-purpose/single-task, tipicamente applicazioni piccole, ossia <32 KB.

Un processo è un programma in esecuzione, che ha la propria memoria e può avere più thread.
Un thread è un flusso di controllo indipendente, che condivide la memoria con gli altri thread del medesimo processo.
Il multithreading è l'esecuzione concorrente di più thread, spesso chiamati task nei RTOS. Questo è elemento di modularizzazione di programmi e introduce la nozione di priorità.
Lo scheduling è ad opera del kernel per realizzare multitasking e multithreading, effettuato con varie strategie ed algoritmi (preemptive vs cooperativo, round-robin + priorità).
Il thread context è l'insieme di informazioni critiche all'esecuzione del thread, sul contenuto corrente dei registri del processore, del program counter, dello stack pointer: è salvato quando un thread è preempted e ripristinato quando un thread è resumed. Il context switch è l'interruzione dell'esecuzione del thread corrente, col passaggio del processore al kernel e poi eventualmente ad un altro thread.

I tipi di scheduling sono:

- big loop: ogni task è polled per verificare se richiede di essere seguito. Il polling procede sequenzialmente o in ordine di priorità. È inefficiente e c'è mancanza di responsiveness
- round-robin: il processore è dato a turno ai task ready. i thread possono essere eseguiti fino al completamento o al blocco, oppure può essere imposto un time-slice ad ogni thread (in questo caso viene sfruttata la preemption)
- priority-based: il processore esegue sempre il thread con priorità maggiore, e se i thread hanno la stessa priorità si usa il round robin. Se c'è un ready thread più prioritario di quello in esecuzione viene cambiato. Questo tipo di scheduling è supportato da tutti i RTOS, eventualmente integrato col RR, poiché ha il vantaggio di garantire massima responsiveness. Però ha anche alcuni svantaggi, fra cui la complessità e il costo della preemption, la possibilità di starvation e il problema dell'inversione della priorità.

I meccanismi di comunicazione e sincronizzazione sono:

- semafori: ci sono due tipi d'uso
  - regolare la competizione, utilizzando la mutua esclusione (mutex, spin-locks) e l'accesso regolato (semafori risorsa/counting)
  - sincronizzazione e coordinazione, utilizzando semafori evento
- comunicazione a scambio di messaggi: abbiamo
  - code di messaggi, ossia strutture dati gestite dal kernel che permettono la comunicazione di messaggi tra task. Sono condivisibili da più task
  - primitive, ossia send e receive
  - architettura produttore-consumatore, dove le code di messaggi sono bounded buffer

Approfondimento sullo scheduling di task negli RTOS

Le proprietà temporali dei task sono definite da quattro parametri fondamentali:

- release time (r): istante in cui il task entra nella coda di ready, essendo pronto per essere eseguito
- execution time (e = WCET): durata massima di esecuzione del task
- response time: intervallo di tempo che intercorre da quando il task è rilasciato a quando ha completato la sua esecuzione
- deadline (D): massimo intervallo di tempo permesso per l'esecuzione di un task

Ci sono vari tipi di task:

- periodici: c'è un intervallo di tempo prefissato (periodo p) che intercorre fra release time dei vari task
- aperiodici: task rilasciati a tempi arbitrari
- sporadici: task rilasciati a tempi arbitrari, con hard deadline

Il progettista di un sistema real-time deve decidere cosa fare se una deadline non è soddisfatta: il task deve continuare la propria esecuzione (quindi ritardando gli altri task)? Oppure deve essere forzata la terminazione del task (liberando la CPU)?
Uno scheduler deve prendere tre tipi di decisioni:

- assegnamento: quale processore usare per eseguire un certo task
- ordine: in quale ordine ogni processore deve eseguire i suoi task
- timing: quando un determinato task deve o può essere eseguito

Ognuno di questi tre tipi di decisione può essere preso o a design time o a run time, durante l'esecuzione del programma. A seconda di quando vengono prese le decisioni si distinguono tipi di scheduler diversi:

- offline scheduler
  - fully-static scheduler:
    - tutte e tre le decisioni sono prese a design time
    - usato quando si conoscono perfettamente tutte le tempistiche dell'esecuzione dei task, tipicamente indipendenti: non si usano in questo caso semafori o altri meccanismi per la sincronizzazione, si sincronizzano i task a partire dai loro tempi di esecuzione
    - difficili da usare con i moderni microprocessori, poiché il tempo necessario per eseguire un task è difficile da prevedere a priori e poiché i task hanno spesso dipendenze di tipo data-dependence
  - static-order scheduler:
    - task assignment e ordering a design time, tuttavia la decisione di quando eseguire i task a livello temporale viene fatta a runtime: tale decisione può dipendere, ad esempio, dalla presenza di blocchi a causa di lock o semafori
- online scheduler
  - static assignment scheduler: assegnamento a design time e tutto il resto a runtime, presenza di un runtime scheduler
  - fully-dynamic scheduler: anche l'assegnamento viene deciso a runtime, quando un processore si rende disponibile.

In generale è possibile che a seconda del tipo di sistema real-time si adottino strategie di scheduling diverse. Esistono due macrocategorie di sistemi real-time:

- sistemi real-time sincroni: in questo caso, un clock hardware è utilizzato per suddividere il tempo del processore in intervalli chiamati frame. Il programma dev'essere suddiviso in task in modo che ogni task possa essere completamente eseguito nel caso peggiore in un singolo frame:
  - mediante una tabella di scheduling si assegnano i task ai frame, in modo che tutti i task all'interno di un frame siano completamente eseguiti prima della fine del frame
  - quando il clock segnala l'inizio del nuovo frame, lo scheduler richiama l'esecuzione dei task specificata in tabella
  - se un task ha una durata che eccede un frame, lo si deve esplicitamente suddividere in sottotask più piccoli che possano essere individualmente schedulati in più frame successivi

  La dimensione del frame diventa un parametro scelto a livello di progettazione. Questo approccio è utilizzato primariamente in sistemi di controllo hard real-time.
  In questo caso la complessità del lavoro è tutta nella decomposizione in tasks che entrino nei frame e nella costruzione della tabella di scheduling. Ci sono vari svantaggi:
  - sistemi molto fragili dal punto di vista della progettazione, poiché una modifica/estensione del sistema può portare a task troppo lunghi e quindi la tabella dev'essere ricomputata
  - il tempo del processore può essere sprecato, poiché il task che costituisce il caso peggiore condiziona l'intero sistema
- sistemi real-time asincroni: in questo caso non si richiede che i task completino l'esecuzione entro frame temporali prefissati: ogni task prosegue la propria esecuzione e lo scheduler è invocato per selezionare il prossimo task, pronto per l'esecuzione. Il vantaggio è l'efficienza,

poiché non si ha spreco di tempo nel processore, lo svantaggio è che in questo caso il soddisfacimento delle deadline è un problema più complesso.
Nei sistemi asincroni, le priorità e lo scheduling preemptive sono usati per fare in modo che i task più importanti siano eseguiti quando necessario, a discapito di task meno importanti:

- priorità descritta da un numero naturale assegnato al task
- un task non viene eseguito nel caso in cui in un sistema ci sia un task nella coda di ready con priorità maggiore

L'implementazione si basa su scheduling events che causano l'invocazione dello scheduler: sono generati quando un task viene accodato nella coda di ready, oppure generato dall'interruzione del timer ad ogni colpo di clock.
Lo scheduler preemptive usa una tecnica di time-slicing e uno scheduling round-robin: ha un insieme di ready queue, una per ogni priorità e ognuna con strategia RR. Vuole garantire un response time adeguato e allo stesso tempo un uso efficiente del processore mediante un opportuno assegnamento delle priorità.
Il watch-dog task è un task usato nei sistemi asincroni per controllare l'esecuzione di altri task: è un task periodico, con priorità massima, che ha il compito di verificare che determinati task siano sempre eseguiti e quindi di segnalare problemi.

L'integrazione di sistemi (operativi) basati su interruzioni con questo tipo di scheduling può essere fatto modellando gli interrupt handler come task la cui priorità è superiore a qualsiasi altro (normale) software task (sistemi interrupt-driven): questo garantisce che gli interrupt handler possano essere eseguiti come sezioni critiche, non interrompibili da altri task; se ci sono interrupt molteplici, allora l'HW può essere programmato in modo da mascherare le interruzioni. La comunicazione fra interrupt handler e task avviene tipicamente adottando architetture produttore/consumatore (ad esempio, il sensore di temperatura che segnala un nuovo dato da leggere, l'handler legge il dato e lo memorizza in un buffer globale in modo che il dato sia leggibile dagli altri task). Il problema è che l'interrupt handler non può mai bloccarsi (ad esempio usando i semafori) per una questione di performance, per cui devono essere utilizzati algoritmi e tecniche non bloccanti (ad esempio il bounder buffer: l'handler che funge da produttore non può bloccarsi se il buffer è pieno, deve o sovrascrivere il dato più vecchio o gettare via il nuovo dato).
La priorità assoluta data alle interruzioni rispetto ai task software può creare situazioni problematiche, specialmente quando gli interrupt sono generati da componenti HW sui quali il designer ha poco controllo.

L'uso di scheduling preemptive basato su priorità può interferire con le strategie di sincronizzazione adottate dai software task, portando a problemi. Un problema importante è dato dall'inversione di priorità, per effetto della quale un task a bassa priorità può ritardare l'esecuzione di un task a priorità più elevata, ad esempio task con sezioni critiche o task che accedono a monitor (e sono nella loro coda di attesa). Questo problema dev'essere evitato, per fare in modo che l'analisi delle performance di un sistema real-time possa essere condotta sui task a prescindere dal livello di priorità (di altri task). Le due soluzioni proposte sono:

- priority inheritance: il problema può essere risolto aumentando temporaneamente la priorità del task in sezione critica ad essere pari a quella massima fra tutti i task che devono/possono essere schedulati in futuro e che vogliono entrare in sezione critica.
- priority ceiling locking: appropriata quando si lavora coi monitor. L'idea è di assegnare al monitor una priorità che sia maggiore o uguale alla più alta priorità posseduta dai task che richiedono di interagire con il monitor chiamando le sue operazioni. Quando un task chiama una di queste operazioni, eredita la ceiling priority del monitor.

Un aspetto fondamentale dello scheduling in un sistema asincrono è assegnare le priorità ai task in modo che tutti i task rispettino le scadenze designate (assegnamento feasible). Sono stati proposti vari algoritmi di scheduling, specificatamente per i sistemi real-time, con risultati in merito alle condizioni necessarie e sufficienti affinché esistano assegnazioni feasible. Fra i principali abbiamo:

- rate monotonic (RM): è un algoritmo basato su fixed-priority, cioè le priorità sono definite staticamente e non cambiano, e il loro valore è dato dall'inverso del periodo, per cui più piccolo è il periodo, maggiore è la sua priorità, a prescindere dalla sua durata prevista. È dimostrato che è un algoritmo ottimo (ovvero trova la soluzione ottima, se esiste) in merito alla feasability.
- earliest deadline first (EDF): si basa sulla modifica dinamica delle priorità dei task, ed è utilizzabile negli scheduler che consentono priorità dinamica. EDF assegna la priorità maggiore dinamicamente (quando interviene lo scheduler) al task con deadline più vicina. Anche per EDF è stato dimostrato che è un algoritmo ottimo per la feasability, sotto la condizione necessaria e sufficiente che dato un insieme T di N task il parametro U di utilizzo del processore sia <= 1. EDF non è sempre applicabile, poiché HW task come gli interrupt handler possono richiedere di avere priorità prefissate, non modificabili, e inoltre comporta un certo overhead sullo scheduler che in alcuni casi può non essere accettabile (l'overhead è dato dal fatto che le priorità dei task devono essere ricomputate ad ogni evento di scheduling)

DAI SISTEMI EMBEDDED AD IOT

Internet of Things (IoT) è un termine introdotto nel 1999, con l'obiettivo iniziale di automatizzare l'inserimento di dati real-time in rete relativi al mondo fisico (identificazione di oggetti, misure, eventi), mediante opportuni sensori, ed evitando l'inserimento manuale ad opera di persone. Ebbe un impatto molto significativo a più livelli della società, non solo tecnologico. L'IoT è alla base di Industrial Internet: è un'infrastruttura che supporta la connettività ad ampia scala fra dispositivi e data, prevede l'integrazione di sistemi embedded con sensori, software e sistemi di comunicazione; è spesso chiamata Industria 4.0.
I "Things" sono i sistemi embedded, i sensori in particolare. Riguardo alla connettività e comunicazione, sono stati studiati protocolli appositi (MQTT, XMPP) basati su Internet, e si tengono in considerazione aspetti relativi all'interoperabilità (vocabolari, ontologie). Ci sono aspetti correlati a queste due dimensioni, come l'aspetto della sicurezza.

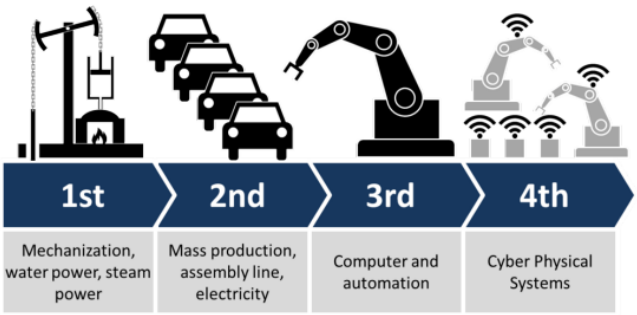
Nel contesto di IoT, gli smartphone possono svolgere vari ruoli:

- fungere essi stessi da sistemi embedded, siccome sono dotati di opportuni sensori per raccogliere e inviare dati geolocalizzati
- fungere da "telecomando" universale, con una UI unificata per interagire con altri smart things
- fungere da unità di controllo collegata in rete che interagisce (tipicamente via Bluetooth) con altri dispositivi wearable e non

In origine, l'elemento caratterizzante di IoT era dato dalla tecnologia RFID (Radio-Frequency Identification): è una tecnologia per l'identificazione e/o memorizzazione automatica di informazioni inerenti oggetti basata sulla capacità di memorizzazione di dati da parte di particolari etichette elettroniche, chiamate tag (o anche transponder), e sulla capacità di queste di rispondere all'interrogazione a distanza da parte di appositi apparati fissi o portatili chiamati tag passivi, attivi o semi-passivi. Per i passivi la distanza è fino ai 2 m, e la capacità di memorizzazione è dell'ordine del kilobyte. La recente evoluzione è NFC (Near-Field Communication), che opera in piccole distanze (<10 cm) ma con maggiore velocità di trasmissione (424 Kbit/s) e interazione direttamente fra 2 lettori.

L'evoluzione dell'IoT è in 5 stadi:

1. product stage: il condizionatore
2. smart product: il condizionatore programmabile
3. smart connect products: il condizionatore accessibile da Internet, controllabile dal cellulare
4. product systems: il termostato smart parla con HVAC e le tapparelle intelligenti e il pavimento riscaldato
5. system of systems: le home appliances parlano con la home security che parla con l'automobile e i dispositivi wearable che parlano con l'ospedale smart

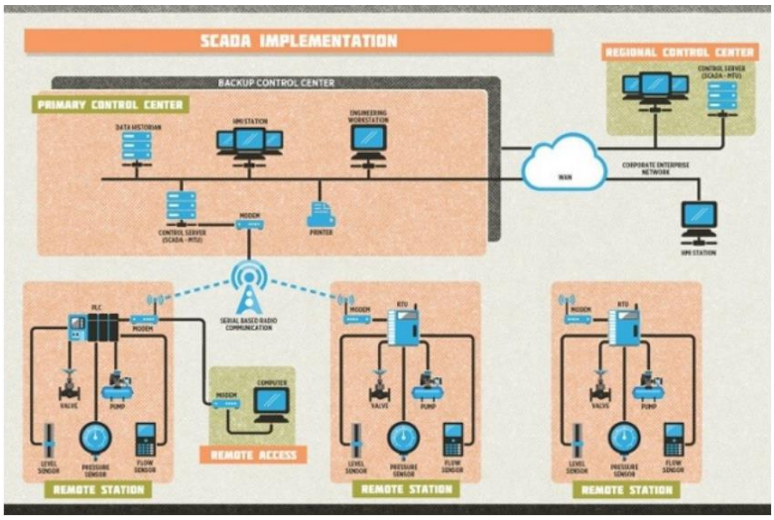


L'IoT è ora una tecnologia chiave per le industrie e le imprese: è infatti l'Industria 4.0, ossia la quarta rivoluzione industriale. Enterprise IoT è un termine generale per indicare l'uso di IoT a livello enterprise, mentre Industrial

IoT è la caratterizzazione di Enterprise IoT specificatamente nel contesto industriale/manifatturiero in particolare.

In ambito industriale, un tipo di sistema M2M molto diffuso è dato dai sistemi SCADA:

- sono sistemi di supervisione che operano con codifiche di segnali trasmesse per canali di comunicazione al fine di realizzare un controllo remoto di sistemi/equipaggiamenti/impianti
- il sistema di supervisione può essere combinato con sistemi di acquisizione dei dati attraverso l'aggiunta di ulteriori codifiche di segnali e comunicazioni al fine di acquisire informazioni circa lo stato dei dispositivi remoti per poterle visualizzare o registrare



È un tipo di sistema di controllo industriale (ICS), ossia sistemi computerizzati che monitorano e controllano i processi industriali in un certo ambiente fisico. I sistemi SCADA si distinguono storicamente da altri ICS per scala, ovvero tipicamente riguardando processi a larga scala che includono molteplici siti, distribuiti in un territorio che può essere geograficamente molto vasto. I principali elementi sono:

- Sensor e Remote terminal units (RTUs): alla base dello SCADA c'è una rete di sensori che permette di monitorare il funzionamento e lo stato dei macchinari e dei processi industriali; collegati ai sensori troviamo le RTU, che hanno il compito di raccogliere i dati rilevati dai sensori e trasformarli in segnali digitali da inviare alla postazione di controllo remota.
- Programmable logic controller (PLCs): alla rete di sensori ed RTU è collegato uno o più PLC, che ha il compito di supervisionare la raccolta di informazioni, fornendo istruzioni sia alle RTU sia direttamente ai sensori: è il PLC a indicare alla rete sensoriale quali sono gli intervalli di tempo nei quali effettuare le misurazioni e controllare i valori dei macchinari. I PLC sono programmabili con linguaggi di programmazione della famiglia IEC 61131-3.
- un sistema di telemetria, che connette PLC e RTU con centri di controllo, data warehouse e sistemi enterprise (può essere una rete informatica come una LAN o WAN, o più semplicemente una rete di linee seriali, sia wired che Wi-Fi)
- Computer supervisore (MTU): server di controllo che raccoglie periodicamente i dati dai PLC, li elabora per ottenerne informazioni utili, memorizza le informazioni, invia comandi di controllo
- Interfaccia uomo-macchina (HMI), che permette ad operatori umani di accedere ai dati processati, monitorarli e interagirvi.
- historian: servizio software che memorizza time-stamped data, eventi e allarmi, in un database che viene poi interrogato al fine di creare report grafici nel HMI.

La caratteristica fondamentale dell'IoT è la capacità di comunicare direttamente o indirettamente con la rete Internet. Ciò permette di gestire in modo opportuno grandi volumi e stream di dati generati dai sensori, e che non potrebbero essere memorizzati in locale. Tali informazioni diventano quindi accessibili mediante opportuni servizi Internet o Web, che ne permettono lo scambio con altre applicazioni, in particolare con applicazioni mobile.

Il cloud ha un ruolo importante, poiché consiste in servizi a disposizione per gestione di dispositivi, storage di dati, analisi offline e online, e si tratta di sistemi aperti.
Il cloud computing è un paradigma di erogazione di risorse informatiche, come l'archiviazione, l'elaborazione o la trasmissione di dati, caratterizzato dalla disponibilità on demand attraverso Internet a partire da un insieme di risorse preesistenti e configurabili. Le risorse vengono date come servizio (as-a-service) in rete e possono essere a tre livelli diversi:

- IAAS (infrastructures as a service): macchine virtuali, server, storage, network
- PAAS (platform as a service): execution runtime, database, web servers, IDEs
- SAAS (software as a service): CMR, giochi, office apps, mail apps

Lato client (desktop, mobile, wearable, embedded) abbiamo browser moderni (HTML5 e JS) e qualsiasi app che usi HTTP API.
Nel caso di IoT, i servizi cloud vengono primariamente utilizzati per:

- raccolta di dati inviati dai dispositivi
- accesso ai dati inviati dai dispositivi (pull, event-driven, analisi offline e online)
- gestione "aperta" dei dispositivi

Il volume e la velocità dei dati generati può essere tale da richiedere tecniche di memorizzazione, elaborazione e accesso che vanno al di là di quelle fornite dai tradizionali database: queste tecniche e architetture sono esplorate nell'ambito dei big data.

Gli aspetti critici dell'IoT sono:

- sicurezza
- privacy
- problema della proprietà dei dati
- scalabilità e interoperabilità

In quest'ultimo ambito, fra le iniziative significative c'è il Web of Things (WoT): consiste in sistemi che incorporano gli oggetti fisici di uso quotidiano nel World Wide Web dando loro una REST-ful API. Questo facilita la creazione di profili virtuali degli oggetti e la loro integrazione e riuso in tipi diversi di applicazioni. È un'evoluzione di IoT dove il punto fondamentale concerne come gli oggetti comunicano mediante opportuno livello di rete (es. Zigbee, Bluetooth).

Il mobile computing è lo sviluppo di sistemi software in esecuzione su dispositivi mobile, come smartphone e tablet. Fra gli aspetti peculiari abbiamo:

- interazione con lo user
- energy-awareness
- connessione alla rete intermittente
- location-based/context-aware-app

Nel contesto di IoT, due aspetti:

- dispositivi mobili come interfaccia per interagire con smart things ed environment, con Bluetooth, NFC + Internet
- dispositivi mobili come thing (con sensori, attuatori, connessione Internet)

Il wearable computing descrive dispositivi indossabili, come smart-watch o smart-glasses, che spesso sono accoppiati con dispositivi mobile tramite Bluetooth. Fra gli aspetti peculiari rispetto al mobile c'è la modalità di utilizzo hands-free. Nel contesto IoT si ha un'interfaccia per estendere la capacità di interazione con ambienti fisici/digitali/aumentati, e la realtà aumentata.

TECNOLOGIE E PROTOCOLLI DI COMUNICAZIONE NEI SISTEMI EMBEDDED

La comunicazione fra dispositivi è un aspetto cruciale di M2M e IoT, ovvero la capacità di scambiare informazioni via rete fra dispositivi e sottosistemi (embedded e non). La comunicazione può essere sia wired, sia wireless. In entrambi i casi i dispositivi vengono dotati di opportuni moduli di comunicazione: nel caso wired tipicamente sono shield/schede con porta Ethernet e protocollo 802.11, nel caso wireless, possono essere shield oppure dongle USB contenente il modulo con ricevitore/trasmettitore e relativa antenna. La comunicazione fra microcontrollore e moduli di comunicazione avviene usualmente mediante la porta seriale.
Ci sono due modi principali per connettere un dispositivo embedded alla rete in un'ottica IoT:

- indirettamente, mediante la comunicazione di un nodo intermediario che funge da gateway: è il caso, ad esempio, di architetture in cui si usa la tecnologia wireless ZigBee o Bluetooth
- direttamente, montando un modulo di comunicazione opportuno: è il caso, ad esempio, di moduli 3G/4G oppure Wi-Fi. In questo caso, a prescindere dalla tecnologia wireless utilizzata, il protocollo di riferimento è quello di Internet, ovvero IP, TCP/IP o UDP/IP

Nell'ambito delle telecomunicazioni, le comunicazioni wireless sono forme di radiocomunicazione: fanno uso del mezzo o canale radio, diffuse nell'etere al fine di trasportare a distanza l'informazione tra utenti attraverso segnali elettromagnetici appartenenti alle frequenze radio o microonde dello spettro elettromagnetico detta anche banda radio. I segnali inviati a tali frequenze vengono perciò detti segnali a radiofrequenza (RF) e il collegamento ottenuto radiocollegamento.
Una radiocomunicazione può essere:

- terrestre, se si appoggia ad infrastrutture di telecomunicazioni poste sulla superficie terrestre
- satellitare, se si appoggia almeno in parte ad infrastrutture poste in orbita sulla Terra come i satelliti artificiali per le telecomunicazioni

Un esempio tipico di radiocomunicazione sono i ponti radio, le infrastrutture di radiodiffusione, telediffusione, l'accesso a reti radiomobili cellulari e le reti satellitari.
Un sistema di radiocomunicazione è composto da:

- un canale trasmissivo che comprende oltre alla tratta radio anche l'emettitore e il ricevitore, le antenne (emittente e ricevente)
- un modem, cioè un modulatore in trasmissione e un demodulatore in ricezione
- un codec, ovvero un codificatore del segnale in ingresso e un decodificatore del segnale in uscita al canale radio: serve per codificare e/o decodificare digitalmente un segnale perché possa essere salvato su un supporto di memorizzazione o richiamato per la sua lettura

In un sistema radio ricetrasmittente tipicamente l'antenna fisica è unica deputata sia alla trasmissione che alla ricezione.

Le tecnologie e protocolli wireless possono essere:

- short range (decine di metri): usati in ambito consumer, meno usati in ambito industriale. Un esempio è il Bluetooth.
- medium range (centinaia di metri fino a 1 miglio): è la fascia di riferimento per IoT. Un esempio è il Wi-Fi.
- long range (distanze superiori al miglio): comprende le reti cellulari (2G, 3G, 4G, 5G) e l'outdoor Wi-Fi.

Il Wi-Fi corrisponde allo standard IEEE 802.11x. È il protocollo più utilizzato per la connessione wireless di dispositivi ad Internet (con supporti per protocolli IP, TCP/IP, UDP/IP). Le sue caratteristiche sono:

- data rate elevati, a 54 Mbps
- communication range ~ 150 m
- frequenza 5 GHz

È utilizzato anche in ambito embedded, sebbene non sia progettato allo scopo e sebbene i consumi siano rilevanti.

Le reti cellulari (pre 5G) permettono la comunicazione mediante rete dati cellulare broad-band. Lo standard si è evoluto: dal GPRS, 3G, WiMax, LTE, 4G. Le sue caratteristiche sono:

- data rate da 80 Kbps (GPRS) ad alcuni Mbps (3G e 4G)
- range: km
- frequenze: da 800 MHz a 1900 MHz

I consumi sono elevati, per cui non sono ideali per IoT.

Il 5G è una tecnologia "disruptive" anche per il mondo IoT, poiché garantisce:

- velocità di 1-10 Gbps
- riduzione delle latenze (1-10 ms)
- aumento dell'affidabilità, anche in condizioni critiche
- maggiore flessibilità rispetto al Wi-Fi, pervasività nei dispositivi (sensori, wearable)

Le reti radio sono interfacce semplici di comunicazione via radio, che permettono la connessione con microcontrollori e dispositivi mediante la seriale. Le sue caratteristiche sono:

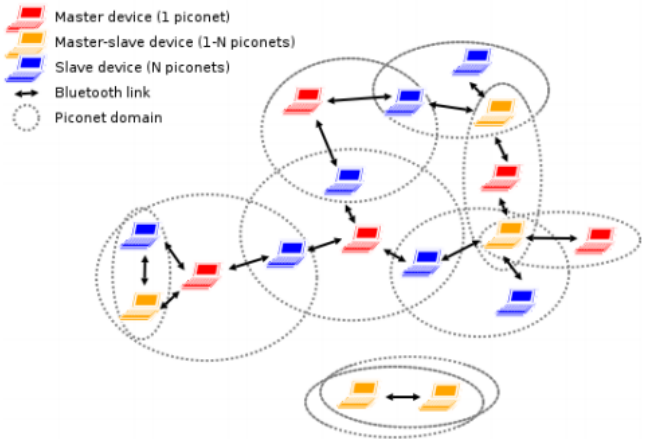
- data rate fino ad 1 Mbps
- range: fino a 1 km, a seconda dell'antenna
- frequenza: 2.4 GHz

Non supportano protocolli di alto livello come TCP/IP, bisogna implementare il proprio protocollo.

Il Bluetooth è lo standard tecnico-industriale di trasmissione dati per reti personali senza fili a corto raggio. Si basa sulla scoperta dinamica di dispositivi coperti dal segnale radio entro un raggio di qualche decina di metri mettendoli in comunicazione tra loro: esempi di dispositivi sono i palmari, i telefoni cellulari, i PC, i portatili, stampanti, fotocamere digitali, console.
È uno standard progettato con l'obiettivo primario di ottenere bassi consumi, un corto raggio d'azione (fino a 100 metri di copertura per un dispositivo di Classe 1 e fino ad un metro per dispositivi di Classe 3) e un basso costo di produzione per i dispositivi compatibili. Lavora nelle frequenze libere di 2.45 GHz: per ridurre le interferenze il protocollo divide la banda in 79 canali e provvede a commutare tra i vari canali 1600 volte al secondo. Riguardo alla velocità, le versioni 1.1 e 1.2 del protocollo gestiscono velocità di trasferimento fino a 723 kbit/s, la versione 2.0 gestisce una modalità ad alta velocità che consente fino a 3 Mbit/s: con questa modalità però aumenta la potenza assorbita. La nuova versione utilizza segnali più brevi, e quindi riesce a dimezzare la potenza richiesta rispetto al Bluetooth 1.2 (a parità di traffico inviato).
I dispositivi BT si dividono in 4 classi, a seconda della potenza consumata e del raggio di funzionamento (a destra).
La rete di base BT è chiamata piconet, basata su architettura master-slave:

Classe	Potenza ERP		Distanza
	(mW)	(dBm)	(m)
1	100	20	~100
2	2,5	4	~10
3	1	0	~1
4	0,5	-3	~0,5

- ogni dispositivo Bluetooth è in grado di gestire simultaneamente la comunicazione con altri 7 dispositivi slave
- un dispositivo per volta può comunicare con il master
- la comunicazione è sincronizzata con il clock del master



È possibile connettere tra loro piconet e formare una scatternet: gli slave possono appartenere a più piconet contemporaneamente mentre il master di una piconet può al massimo essere lo slave di un'altra. Ogni dispositivo Bluetooth è configurabile per cercare costantemente altri dispositivi e per collegarsi a questi: può essere impostata una password per motivi di sicurezza.
In generale i collegamenti che possono essere stabiliti tra i diversi dispositivi sono di due tipi:

- orientati alla connessione: richiede di stabilire una connessione tra i dispositivi prima di inviare i dati
- senza connessione: non richiede alcuna connessione prima di inviare i pacchetti: il trasmettitore può in qualsiasi momento iniziare ad inviare i propri pacchetti purché conosca l'indirizzo del destinatario

La tecnologia Bluetooth definisce due tipi di collegamenti a supporto delle applicazioni voce e trasferimento dati:

- un servizio asincrono senza connessione (ACL), per traffico di tipo dati, ed è un servizio di tipo best-effort
- un servizio orientato alla connessione (SCO), per traffico di tipo real-time e multimediale

Il Bluetooth ha uno stack di protocolli, organizzato a livelli come avviene per l'architettura ISO/OSI: differenti protocolli sono utilizzati per differenti applicazioni, e indipendentemente dal tipo di applicazione lo stack include sempre livelli data-link e fisico.
Il protocollo RFCOMM è utilizzato per sfruttare Bluetooth come una classica linea di comunicazione seriale, emulando una tradizionale porta seriale RS-232: è utile per interfacciarsi e interagire con dispositivi come stampanti, model, PC, laptop.
Altri protocolli sono definiti da altre organizzazioni di standardizzazione e incorporati nell'architettura Bluetooth:

- PPP: lo standard Internet per trasportare i pacchetti IP su una connessione punto-punto
- TCP/UDP-IP
- OBEX: object exchange, un protocollo a livello sessione sviluppato dalla Infrared Data Association per scambio di oggetti, simile ad HTTP ma più semplice; è usato ad esempio per trasferire dati in formato vCard e vCalendar, cioè biglietto da visita e calendario degli impegni
- WAE/WAP: Wireless Application Environment e Wireless Application Protocol

Per facilitare l'utilizzo dei dispositivi Bluetooth sono stati definiti una serie di profili, che identificano una serie di possibili applicazioni.
Il Bluetooth utilizza l'algoritmo SAFER+ per autenticare i dispositivi e per generare la chiave utilizzata per cifrare i dati.
Il Bluetooth si è evoluto dalla versione 1.0 alla versione 5. La versione 4.0, detta Bluetooth Smart, include i protocolli:

- Classic Bluetooth: legacy BT
- Bluetooth High Speed: basato su Wi-Fi

- Bluetooth Low Energy (BLE): protocollo alternativo a quello BT classico, pensato appositamente per IoT
    - consumo di energia molto ridotto
    - facilità di creazione di link di comunicazione fra device
    - reattività

Ci sono due implementazioni:

- single-mode, in cui solo lo stack BLE è implementato
- dual-mode, in cui le funzionalità di BT Smart sono integrate a quelle Classic

Bluetooth 5 è un evoluzione verso IoT, poiché:

- 4x area di trasmissione, ma stesso livello di consumo
- 2x velocità fino a 2 Mbps in modalità basso consumo, riduzione dei tempi necessari per ricevere e trasmettere i dati
- 8x capacità di trasmissione, per un minor tempo richiesto per il comportamento dei task, adatto per l'adozione e gestione dei beacon

Lo standard 802.15.4 è lo standard che definisce a livello di physical layer e media access control per reti wireless "personali" ubique a bassa velocità e basso costo:

- l'enfasi è sulla comunicazione a bassi consumi e agile fra dispositivi vicini, a corto raggio
- comunicazione entro i 10 metri, con transfer rate di 250 kbit/s
- supporti per la comunicazione real-time e comunicazioni sicure

È la base di ulteriori specifiche che includono anche i livelli sopra (ad esempio ZigBee). Gli usi tipici sono:

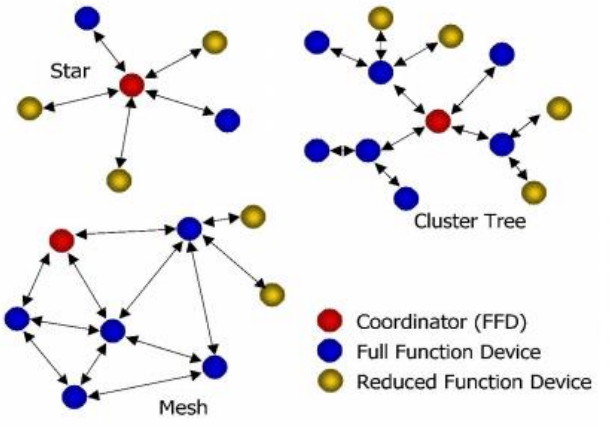
- home and building automation
- automotive networks
- industrial wireless sensor networks
- interactive toys and remote controls

Ci sono due tipi di nodi in rete

- FFD (full-function device): funge da coordinator della PAN e implementa un modello di comunicazione che gli permette di parlare con un qualsiasi altro nodo
- RFD (reduced-function devices): questi nodi rappresentano semplici dispositivi con modeste risorse di elaborazione e comunicazione, e possono comunicare solo con un FFD

Le tipologie sono:

- star o P2P: ogni rete deve avere almeno un FFD che funge da coordinatore
- cluster tree: struttura in cui un RFD può avere un solo FFD



Le reti P2P possono formare pattern arbitrari di connessioni: sono la base per formare reti ad hoc in grado di fare self-management and organization.

Le caratteristiche sono le seguenti:

- frequenza: opera nelle bande libere 2.4 GHz, 900 MHz e 858 MHz
- data rate: può arrivare ad un massimo di 250 kbit/s
- range di comunicazione: da 100 m a 1 km, dipendentemente dalla potenza in uscita utilizzata (piccoli progetti: 1mW=>100 m)

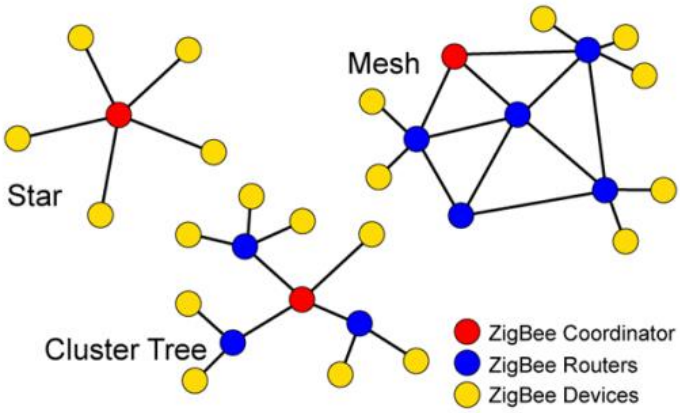
- consumo: 50 mA, per cui una batteria ricaricabile di 850 mAH può fornire circa 17 ore di uso continuato

ZigBee è fra le tecnologie wireless più recenti e utilizzate nei sistemi embedded:

- è una specifica per un insieme di protocolli di comunicazione ad alto livello che utilizzano piccole antenne digitali a bassa potenza e basato sulla standard IEEE 802.15.4 per wireless personal area networks (WPAN)
- ideata nell'ottica IoT, per la comunicazione pervasiva fra device
    - open global standard
    - basso consumo, basso costo

Ci sono tre tipi di dispositivo ZigBee:

- ZigBee Coordinator (ZC): è il dispositivo più "intelligente" tra quelli disponibili, costituisce la radice di una rete ZigBee e può operare da ponte tra più reti. Ci può essere un solo Coordinator in ogni rete. Esso è inoltre in grado di memorizzare informazioni riguardo alla sua rete e può agire come deposito per le chiavi di sicurezza



- ZigBee Router (ZR): questi dispositivi agiscono come router intermedi passando i dati da e verso altri dispositivi
- ZigBee End Device (ZED): includono solo le funzionalità minime per dialogare con il suo nodo parente (Coordinator o Router). Non possono trasmettere dati provenienti da altri dispositivi. Sono i nodi che richiedono il minor quantitativo di memoria e quindi risultano spesso più economici rispetto ai ZR o ai ZC

Confrontando ZigBee e Bluetooth, è evidente che il primo abbia una migliore efficienza, un range di comunicazione più ampio, creazione di reti P2P più semplice e agile, il che la rende la soluzione migliore per IoT: tuttavia è meno utilizzata a livello consumer.

Come protocolli di livello applicativo ci sono varie scelte possibili:

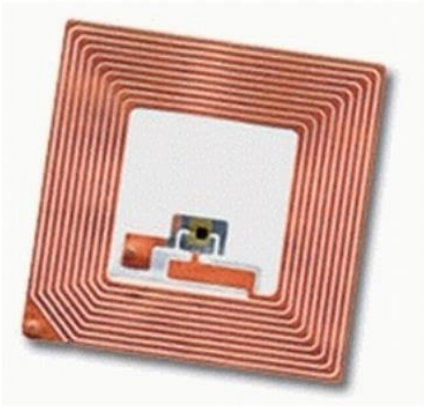
- nessun protocollo applicativo specifico
- SCADA
- generici protocolli web-based, ossia protocolli di trasporto UDP/TCP: TCP è preferibile per reti cellulari perché sono tipicamente robuste e possono gestire bene l'overhead, per le reti LLN (low-power and lossy), in cui sia i dispositivi che le reti sono constrained, tipicamente si usa UDP
- protocolli pensati per IoT
    - pensati per nodi e reti constrained
    - MQTT: MQ Telemetry Transport, è un protocollo open source sviluppato e ottimizzato per dispositivi vincolati e reti a bassa ampiezza di banda, ad alta latenza o inaffidabili. È un protocollo di trasporto di messaggistica publish/subscribe estremamente leggero e ideale per la connessione di dispositivi di piccole dimensioni a reti con ampiezza di banda minima. MQTT è efficiente in termini di utilizzo dell'ampiezza di banda, è indipendente dai dati e dispone di un riconoscimento continuo delle sessioni poiché utilizza TCP. Questo protocollo è mirato a ridurre al

minimo i requisiti delle risorse dei dispositivi tentando al contempo di garantire affidabilità e un certo livello di garanzia di consegna con gradi di servizio.
MQTT è destinato a grandi reti di dispositivi di piccole dimensioni che devono essere monitorate o controllate da un server backend su Internet. Non è progettato per il trasferimento tra dispositivi, né per il multicast dei dati a molti ricevitori. MQTT è semplice e offre poche opzioni di controllo. In genere, le applicazioni che utilizzano MQTT sono lente, ovvero la definizione di "tempo reale" in questo caso è misurata solitamente in secondi.

- CoAP: Constrained Application Protocol, per l'utilizzo con reti e nodi (vincolati) con perdita di dati e basso consumo energetico (LLN). Come HTTP è un protocollo RESTful. È semanticamente allineato con HTTP e dispone perfino di una mappatura one-to-one bidirezionale con HTTP. I dispositivi di rete sono vincolati da microcontrollori di dimensioni minori con piccole quantità di memoria flash e RAM, mentre i vincoli su reti locali, ad esempio 6LoWPAN, sono dovuti a percentuali di errore di pacchetto elevate e a una bassa velocità di elaborazione (decine di kbit al secondo). CoAP può essere un buon protocollo per dispositivi a batteria o con approvvigionamento energetico.
  Le caratteristiche di CoAP sono:
  - poiché CoAP usa UDP, alcune delle funzionalità TCP sono replicate direttamente in CoAP: ad esempio CoAP distingue tra messaggi confermabili e non confermabili
  - richieste e risposte vengono scambiate in modo sincrono su messaggi CoAP (a differenza di HTTP, dove viene utilizzata una connessione TCP)
  - tutte le intestazioni, i metodi e i codici di stato hanno una codifica binaria, aspetto che riduce l'overhead del protocollo. Tuttavia ciò richiede l'utilizzo di un analizzatore del protocollo per la risoluzione dei problemi di rete
  - a differenza di HTTP, la capacità di memorizzare risposte CoAP nella cache non dipende dal metodo di richiesta, ma dal codice della risposta

  CoAP soddisfa pienamente le esigenze di un protocollo estremamente leggero e con una connessione di natura permanente. Tale protocollo ha familiarità semantica con HTTP ed è RESTful (risorse, identificatori di risorse e manipolazione di tali risorse tramite interfaccia di programmazione delle applicazioni (API) uniforme). Se si dispone di una certa esperienza con il Web, l'utilizzo di CoAP è relativamente semplice.

Radio Frequency Identification (RFID) sono tecnologie inizialmente introdotte per l'identificazione e tracking a distanza di oggetti mediante l'ausilio di piccoli dispositivi elettronici chiamati tag, che possono essere letti/scritti a distanza (di cm) da appositi RFID reader. Un tag RFID contiene tipicamente un ID univoco che contraddistingue l'oggetto su cui è montata l'etichetta. È stata una delle tecnologie chiave di IoT, oggi sempre più sostituiti da NFC e iBeacon. Un tag può essere:



- passivo: non è alimentato. È caratterizzato da un circuito che viene alimentato indirettamente, a distanza, dalle onde radio inviate da un RFID reader che ne vuole leggere il contenuto, che tipicamente corrisponde ad un ID univoco. Può essere anche scritto da un RFID reader, usando lo stesso principio

- attivo: hanno batterie proprie e trasmettono in broadcast il proprio contenuto informativo continuamente
- battery assisted passive (BAP): ibrido, come il caso attivo ma invia le informazioni solo se sollecitato da un RFID reader. Supporta distanze maggiori rispetto al passivo

Near Field Communication (NFC) fornisce connettività wireless (RF) bidirezionale a corto raggio (max 10 cm). È una tecnologia evoluta da una combinazione della tecnologia RFID e altre tecnologie di connettività: contrariamente ai più semplici RFID, NFC permette una comunicazione bidirezionale, e quando due apparecchi NFC vengono accostati entro i 4 cm, viene creata una rete P2P tra i due ed entrambi possono inviare e ricevere informazioni. Il target può essere dato da semplici tag, come nel caso di RFID. Si è verificata una diffusione pervasiva in smartphone e dispositivi vari.

i beacon sono tecnologie basate su BLE introdotte recentemente per fare localizzazione e identificazione wireless tipicamente in ambienti indoor. Un iBeacon è un piccolo trasmettitore a bassissimo consumo che emette continuamente un segnale, contenente il suo ID, che può essere rilevato da un qualsiasi dispositivo BLE. Ha un impatto notevole sul panorama IoT.

MODELLI DI COMUNICAZIONE A SCAMBIO DI MESSAGGI

I protocolli e le tecnologie viste abilitano fisicamente la comunicazione in e fra sistemi embedded. Al di là di queste tecnologie, la progettazione e programmazione di sistemi software embedded di rete e distribuiti richiede l'introduzione di opportuni modelli e meccanismi di comunicazione di alto livello, e i relativi supporti tecnologici a livello di librerie/piattaforme/infrastrutture.
Il modello a scambio di messaggi è una comunicazione mediante invio e ricezione di messaggi: è un modello alternativo alla comunicazione mediante memoria condivisa, e un modello di riferimento per i sistemi distribuiti. Le primitive sono send e receive.
La comunicazione può essere:

- diretta
    - la comunicazione avviene direttamente fra i processi (thread, ..) specificando il loro identificatore. Un aspetto importante è come identificare i partecipanti alla comunicazione in modo univoco
    - le primitive sono send e receive
- indiretta
    - si utilizzano dei canali che fungono da mezzi di comunicazione indiretta
    - l'invio e la ricezione avvengono specificando uno specifico canale

Un'altra differenziazione è fra comunicazione:

- sincrona: la send ha successo (e completa) quando il messaggio specificato è ricevuto dal destinatario mediante una receive
- asincrona: la send ha successo (e completa) quando il messaggio è stato inviato, ma non necessariamente ricevuto dal destinatario mediante una receive

Un altro aspetto importante è la strategia di bufferizzazione adottata, fondamentale nel caso asincrono, sia nel caso indiretto per i canali, sia nel caso diretto per la coda che i processi usano per ricevere i messaggi. Concerne la dimensione del buffer dove vengono memorizzati i messaggi ricevuti, ma ancora da ricevere con la receive.
Le primitive sono:

- send(DestId, Msg)
  ha successo quando (semantiche alternative):
    1. il msg è stato inviato (=> caso asincrono): in questo caso la verifica che un messaggio sia arrivato a destinazione dev'essere implementata a livello di protocollo, e si ha un errore se il destinatario non esiste
    2. il msg è arrivato, ma non necessariamente ricevuto, e si ha errore se il destinatario non esiste
    3. il msg è stato ricevuto (caso sincrono)
- receive():Msg
  semantica usuale: bloccante, si blocca sin quando non è disponibile almeno un messaggio

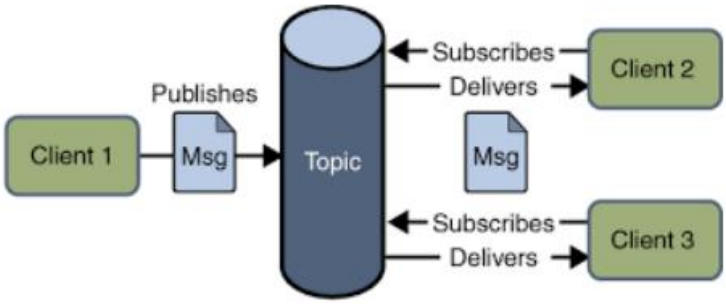
Nel caso asincrono, dev'essere considerato anche il caso in cui il buffer di ricezione dei messaggi si riempia. In quel caso occorre scegliere il tipo di semantica per la send:

- fallisce: msg scartato
- si blocca in attesa che il buffer non sia pieno: stile produttori/consumatori
- ha successo: riscritto messaggio del buffer (più vecchio, più nuovo)

Aspetto fondamentale della comunicazione è dato dal linguaggio con cui si rappresentano i messaggi: dev'essere il medesimo utilizzato da tutti i partecipanti alla comunicazione, come primo requisito per avere interoperabilità. Esempi di linguaggi usati per la rappresentazione di messaggi per avere interoperabilità sono JSON e XML. L'utilizzo dello stesso linguaggio non è sufficiente per

avere interoperabilità. Occorre mettersi d'accordo su formato dei messaggi/vocabolario utilizzato e, in caso di protocolli, sulla semantica dei protocolli.

Il modello publish/subscribe è un modello a scambio di messaggi tipicamente usato per realizzare architetture ad eventi ad alto livello in sistemi distribuiti: è simile al pattern observer, è utile per la comunicazione in sistemi open, dynamic, loosely-coupled, è molto usato in ambito IoT.



È un metamodello:

- channels/topics: canali o argomenti sui quali è possibile inviare messaggi e che è possibile "osservare" mediante sottoscrizione
- publishers: inviano messaggi su canali
- subscribers: si registrano su canale/topic per ricevere tutti i messaggi pubblicati su quel canale/topic

Le primitive (astratte) sono publish e subscribe.

Un aspetto importante è l'integrazione di modelli di comunicazione con i modelli e architetture software di controllo visti per i sistemi embedded, ovvero:

- loop di controllo semplici
- FSM
- task
- event-loop

L'integrazione nei sistemi embedded organizzati a loop ha un problema: la receive bloccante. Essa comporta il blocco del loop e non è usabile nell'implementazione di FSM (sincrone o asincrone). Un esempio semplice è un sistema blinking che è possibile controllare dall'esterno con invio di messaggio stop. La receive non bloccante fa si che, nel caso in cui non ci siano messaggi, la receive non si blocchi: restituisce errore o genera eccezione o restituisce un riferimento null. Viene aggiunta una primitiva per testare la presenza di messaggi disponibili: isMsgAvailable():boolean.

Sfruttando l'estensione, l'integrazione nel modello a stati può avvenire considerando che:

- una transizione può essere scatenata dalla presenza di un messaggio => uso della primitiva di test messaggi nelle guardie
- nell'insieme delle azioni associate alla transizione o allo stato c'è anche la ricezione (non bloccante) e l'invio di messaggi

Nelle guardie non può essere modificato lo stato, sono solo predicati, quindi si può usare isMsgAvailable, non receiveMsg.

Nel modello a task lo scambio di messaggi può essere usato come modello di comunicazione fra task stessi, in alternativa o al completamento del modello basato su memoria condivisa.

Anche nel caso di architetture event-loop, il modello di comunicazione da considerare è necessariamente asincrono con receive implicita oppure non bloccante: questo poiché negli event handler non è possibile eseguire primitive bloccanti. Con l'integrazione l'arrivo di un messaggio è modellato come eventi (la coda di eventi è usata come coda di messaggi) e l'invio di un messaggio tramite send ha come effetto accodare un evento nella coda degli eventi del destinatario.

ANDROID

Introduzione

Android è un sistema operativo per dispositivi mobili basato su Linux Kernel, attualmente sviluppato da Google Inc. e distribuito sotto licenza open-source. È un Sistema Operativo Embedded, progettato per essere eseguito principalmente su dispositivi con interfaccia touch-screen: ne esistono diverse varianti, per altre categorie di dispositivi, ad esempio Android Wear OS, Android TV, ecc.
Si tratta di un sistema operativo (embedded) Linux multi-user:

- non è una distribuzione GNU/Linux né un sistema unix-like (tutte le GNU utils di Linux sono sostituite da software Java)
- ogni applicazione è un utente del sistema a cui è associato un ID univoco
- ogni processo del sistema è eseguito mediante una propria virtual machine

Ciascuna applicazione vive in una propria sandbox protetta:

- ciascuna applicazione è eseguita in un diverso processo di sistema
- il codice di ciascuna applicazione è isolato da quello delle altre
- principio di Least Privilege: ogni applicazione ha accesso ai soli componenti esplicitamente richiesti per eseguire i propri compiti

La Dalvik VM è stata la VM di Android fino alla versione 4.x: diversamente dalla JVM (che è una stack machine) ha un'architettura basata su registri e richiede meno istruzioni, ma più complesse. ART (Android RunTime) è il nuovo runtime system di Android, dalla 5.0. Rispetto a Dalvik è basata su tecnologia AOT (ahead-of-time):

- l'intera compilazione del codice avviene durante l'installazione dell'app su un device e non durante l'esecuzione della stessa
- notevoli vantaggi in termini di performance durante l'esecuzione delle applicazioni

Garbage Collection (GC) e meccanismi di debugging ottimizzati.

I linguaggi di riferimento sono Java e Kotlin, oltre a porzioni di codice scritte in linguaggio nativo C o C++. Sono compilate tramite l'SDK Android in un file denominato Android Package (con estensione .apk) che contiene tutte le risorse necessarie all'applicazione: è il file mediante il quale è possibile installare l'applicazione sui device con Android OS.
Possono essere progettate per supportare dinamicamente una diversa varietà di dispositivi. Ciascuna applicazione può essere distribuita previa apposizione di un certificato digitale che attesti l'identità dello sviluppatore.

Il comportamento di ciascuna applicazione Android può essere progettato mediante istanze di 4 componenti principali:

- Activity: rappresenta una singola schermata di un'applicazione con associata una propria interfaccia utente. L'interfaccia utente di ciascuna applicazione è definita mediante uno o più file di risorse (codificati in XML). Ciascuna applicazione può definire ed eseguire più activity, una sola delle quali può trovarsi in stato di foreground in un preciso momento.
- Service: componente che esegue la propria computazione in background rispetto alle activity. Non è associato ad alcuna interfaccia grafica. Può prevedere meccanismi per la richiesta di esecuzione di compiti da componenti esterni.

- Content Provider: Consente di accedere, gestire e modificare i dati persistenti di un'applicazione. I dati salvati in un Content Provider possono essere privati per una specifica applicazione o condivisi con le altre.
- Broadcast Receiver: componente che permette di catturare e/o rispondere agli annunci (messaggi) propagati all'interno del sistema. Un annuncio può riguardare un qualsiasi elemento del sistema e può essere propagato sia dal sistema operativo stesso, sia da una qualunque applicazione. Alcuni esempi sono:
    - il display del dispositivi è stato spento
    - un determinato file è stato scaricato e ora è disponibile
    - è stata identificata una nuova rete Wi-Fi

  Non è associato ad alcuna interfaccia utente ma può attivare una notifica sulla barra di stato.

Ogni componente è istanziato nell'ambito di una precisa applicazione ma può essere eseguito da qualunque altro componente di qualunque altra applicazione. Ogni componente è eseguito dal sistema all'interno del processo della relativa app.

Il file Manifest descrive l'applicazione, mediante la sintassi XML: dev'essere obbligatoriamente presente nella root directory dell'applicazione in un file denominato AndroidManifest.xml. Esso dichiara l'API Level di riferimento (minVersion e targetVersion). Elenca tutti i componenti dell'applicazione che il sistema può eseguire. Elenca i permessi che l'applicazione richiede per la propria esecuzione. Dichiara i componenti HW e SW che l'applicazione intende utilizzare.

Ciascun componente può essere attivato inviando un messaggio asincrono al sistema chiamato Intent: è un meccanismo valido per attivare istanze di Activity, Service e Broadcast Receiver. L'Intent definisce l'azione da eseguire (il componente da attivare, il messaggio da propagare, ecc.): può specificare una serie di informazioni aggiuntive (flag e extra) destinate al sistema e/o al componente da attivare. Esistono due tipi di Intent:

- espliciti: creati ed invocati attraverso il nome esplicito del componente da attivare
- impliciti: descrivono un'azione generica da eseguire che può essere intercettata da un componente che sia in grado di eseguirla

Activity

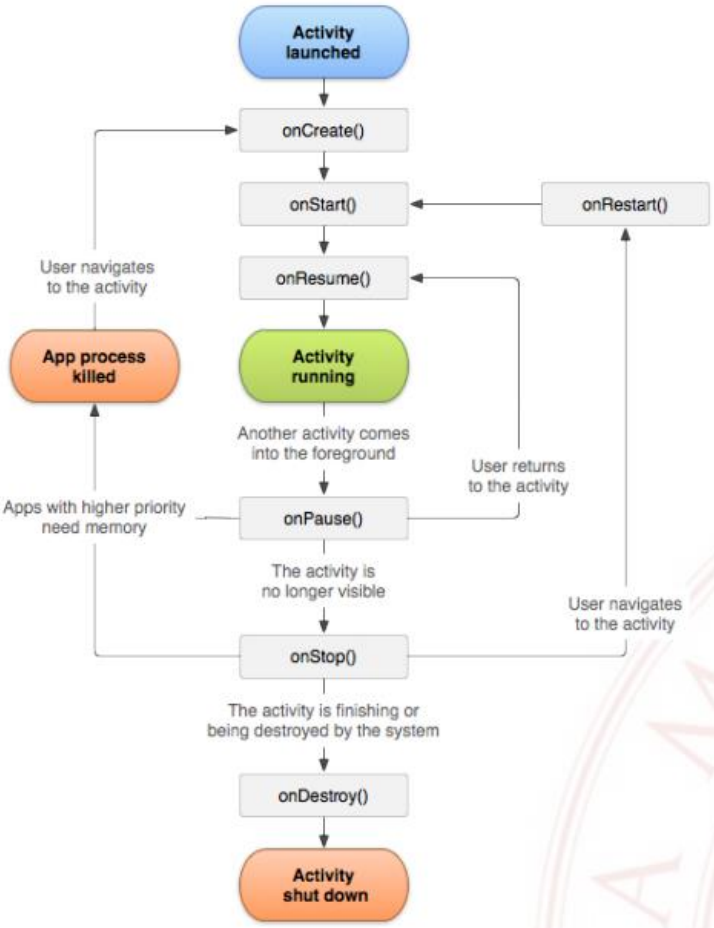
L'Activity è un componente che abilita l'interazione dell'utente con l'applicazione attraverso una specifica interfaccia grafica. Tipicamente, ciascuna Activity occupa l'intera dimensione dello schermo del device. Generalmente un'applicazione consiste in una collezione di Activity, dove una di esse è quella principale (Main Activity).
Un'Activity può essere in stato di:

- resumed (running): l'Activity è in foreground e ha il focus
- paused: caso in cui un'altra Activity è in stato di running, ma questa è ancora visibile
- stopped: l'Activity è in background, ovvero completamente oscurata da un'altra Activity

Quando si trova in paused o stopped, il sistema può eliminarla dalla memoria in qualunque momento.
Il lifecycle di un'Activity si compone di una serie di stati e di loop interni: definisce il comportamento dell'Activity da quando viene lanciata a quando questa viene chiusa o interrotta dal sistema. Le transizioni da uno stato all'altro provocano l'invocazione automatica di uno o più metodi (callbacks): tali metodi possono essere ridefiniti per specificarne il comportamento, e in particolare abbiamo

- onCreate: è invocata quando l'Activity è creata per la prima volta. essa provvede al setup dell'Activity e riceve dal sistema l'eventuale stato precedentemente salvato dell'Activity

- onStart: invocata subito prima che l'Activity sia resa visibile dall'utente. Prepara l'Activity affinché possa essere posta nello stato di foreground
- onResume: invocata dopo che l'Activity è stata resa visibile all'utente ma prima che quest'ultimo possa cominciare ad interagire con essa. Al termine della sua esecuzione l'Activity si trova nello stato resumed (o running)
- onPause: invocata dal sistema sull'Activity quando un'altra Activity dev'essere portata allo stato di running. È utilizzata generalmente per rilasciare le risorse eventualmente detenute dall'Activity. Al termine della sua esecuzione l'Activity si trova nello stato paused
- onStop: invocata quando l'Activity non è più visibile all'utente. Al termine della sua esecuzione l'Activity si trova nello stato stopped
- onRestart: invocata quando l'Activity, precedentemente interrotta (ovvero che si trova nello stato stopped), dev'essere riportata allo stato running
- onDestroy: invocata immediatamente prima della fase di distruzione dell'Activity, ovvero prima che sia rimossa dalla memoria centrale. Invocata sia che la distruzione dell'Activity sia stata richiesta dall'utente, sia nel caso sia stata imposta dal sistema



Ogni nuova Activity deve implementare onCreate, e come best practice anche onPause, mentre le altre sono opzionali. Ogni Activity dev'essere dichiarata nel Manifest. Tra le Activity qui specificate, una dev'essere etichettata come Main Activity.
Un'Activity si può avviare con startActivity, o con startActivityForResult se si vuole un risultato retro-propagato all'Activity chiamante attraverso un Intent.
In genere la terminazione delle Activity è lasciata al sistema.
Ciascuna applicazione contiene generalmente più Activity, e solo una può essere nello stato di running e visibile all'utente, mentre tutte le altre sono inserite in uno stack LIFO di Activity chiamato Back Stack. Salvo diversa specifica, le Activity sono inserite nello stack secondo l'ordine di

attivazione/avvio. Alla base dello stack è sempre presente l'Activity che rappresenta l'Home Screen del sistema. Alla pressione del tasto back, l'Activity in foreground viene rimossa dallo stack e quella precedente viene portata nello stato di foreground.

Processi, Thread, Task

Tutti i componenti di un'applicazione Android sono eseguiti nell'ambito dello stesso Processo (Linux Process): è creato dal sistema operativo all'avvio del primo componente dell'applicazione, e con opportuni parametri è possibile richiedere che l'applicazione usi più processi. Android assicura che ciascun processo sia mantenuto attivo per il maggior tempo possibile, ma il sistema può decidere in qualunque momento di terminare qualunque processo attivo, ad eccezione di quello attualmente in foreground. A tutti i processi attivi è associato un ranking che consente di stabilire quale/i processo/i interrompere per rilasciare risorse necessarie o per questioni di performance.

All'avvio di ciascuna applicazione il sistema crea un thread principale chiamato Main Thread: tutti i componenti di un'applicazione sono istanziati nel Main Thread, che è deputato alla gestione di tutti gli eventi generati; esso gestisce tutte le chiamate di sistema e l'esecuzione di tutte le callbacks. Ha alcune restrizioni in merito al codice che può eseguire: in particolare non può eseguire compiti "long-running", e se resta bloccato per più di 5 secondi il sistema interrompe l'applicazione.
Il Main Thread è l'unico thread dell'applicazione che può eseguire operazioni sull'interfaccia utente. Non può eseguire compiti che potenzialmente potrebbero richiedere molto tempo, tra cui comunicazioni di rete (richieste HTTP), accesso a DB, esecuzione di Query...

In Android è possibile sfruttare il supporto di Java al multithreading e creare o eseguire nuovi thread a cui far eseguire compiti long-running.
Un Handler consente di accodare un messaggio alla coda dei messaggi di un thread definito secondo il pattern MessageLoop: il Main Thread di Android è di fatto un MessageLoop. È generalmente utilizzato per la comunicazione tra il Main Thread e uno o più Worker Thread. Se non diversamente specificato, quando si istanzia un Handler, quest'ultimo è automaticamente associato al flusso di controllo thread che lo crea: quindi un Handler può essere istanziato in un MessageLoop Thread oppure dev'essere definito esplicitamente il MessageLoop a cui deve fare riferimento.

Un Async Task consente di eseguire computazione asincrona sull'interfaccia utente. Permette di specificare quale porzione di codice dev'essere eseguita in background, da un worker thread, e quale dev'essere demandata al Main Thread. Non richiede esplicitamente il worker thread.
Dev'essere implementato il metodo doInBackground, in cui dev'essere inserito il codice la cui esecuzione dev'essere demandata al worker thread. Può essere eseguito richiamando il metodo execute sull'istanza dell'oggetto.
In un Async Task possono essere ridefiniti quattro diversi metodi:

- onPreExecute
- doInBackground
- onProgressUpdate
- onPostExecute

La classe AsyncTask<Params, Progress, Result> è una classe parametrica che richiede tre parametri che rappresentano:

- il primo (Params) è il tipo associato ai parametri che possono essere passati al metodo doInBackground

- il secondo (Progress) è il tipo associato ai parametri che possono essere passati al metodo onProgressUpdate
- il terzo (Result) è il tipo associato al parametro di ritorno del metodo doInBackground e quindi al parametro in ingresso del metodo onPostExecute

Tali parametri possono essere specificati con qualunque tipo di dato, o con il tipo Void che rappresenta l'assenza di parametri.
Tutti gli Async Task devono essere creati nel Main Thread: solo il Main Thread può richiamare il metodo execute sull'istanza di un task.
Su ciascuna istanza di Async Task, il metodo execute può essere richiamato una sola volta, le successive invocazioni provocano un'eccezione.
Non devono essere chiamati manualmente i metodi degli Async Task sull'istanza del task.
Tutti gli Async Task creati ed eseguiti all'interno di una stessa applicazione fanno riferimento allo stesso worker thread, risulta quindi necessario far attenzione ai compiti (potenzialmente bloccanti) che devono essere eseguiti nel metodo doInBackground.

Comunicazione via HTTP e Bluetooth

Per effettuare richieste HTTP in modo semplice si utilizza HttpURLConnection di Java, altrimenti la libreria Volley. Non è consentito effettuare richieste HTTP sul Main Thread: devono essere demandate ad un worker thread (es. via Async Task). Le richieste HTTP sono intrinsecamente asincrone: la risposta, se prevista, può arrivare dopo un certo quantitativo di tempo oppure può scattare un timeout per indisponibilità del server o della rete.

Android include il supporto per lo stack di comunicazione basato su standard Bluetooth dalle prime versioni del sistema. In particolare, in Android è possibile:

- analizzare le frequenze radio per identificare altri dispositivi nelle vicinanze (discovery): è una procedura di scanning delle frequenze Bluetooth per identificare gli eventuali device presenti nel raggio di visibilità. Possono essere identificati tutti i dispositivi che sono visibili agli altri. I dispositivi identificati rispondono condividendo alcune informazioni (MAC Address, Device Name). Per poter attivare la connessione con uno dei device identificati, dev'essere eseguita l'operazione di pairing.
- connettersi a dispositivi precedentemente accoppiati (pairing): è una procedura di accoppiamento di due dispositivi Bluetooth, a carico dell'utente. Consente di memorizzare su ciascun device tutte le informazioni necessarie ad attivare, eventualmente, una connessione tra i due. Il fatto che due dispositivi siano accoppiati non significa che siano connessi tra loro e possano scambiarsi dati su un apposito canale RFCOMM. L'accoppiamento è condizione necessaria ma non sufficiente per consentire la trasmissione dei dati sul canale Bluetooth da un device ad un altro.
- trasferire dati da un dispositivo all'altro tramite stack Bluetooth
- gestire connessioni multiple

Per utilizzare il supporto Bluetooth è necessario richiedere i permessi BLUETOOTH e BLUETOOTH_ADMIN, e da Android 6 anche ACCESS_FINE_LOCATION.
La creazione del canale di comunicazione segue lo stesso schema Client-Server dei canali di comunicazioni basati sul protocollo TCP/IP via socket. Si attiva un canale di comunicazione basato sullo standard RFCOMM.
Lato Server:

1. si attiva un thread che crea una serverSocket che attende richieste di connessioni

2. quando la serverSocket riceve una richiesta di connessione, questa restituisce la socket specifica su cui è stato attivato il canale
3. tale socket è quindi passata all'istanza di un gestore della connessione Bluetooth, il quale può essere usato sia per attendere i messaggi in ingresso sia per inviare messaggi al client

Lato Client:

1. si esegue un task deputato ad eseguire il tentativo di connessione al server
2. se la connessione va a buon fine, viene eseguita un'istanza dello stesso gestore di connessione presente sul server, per gestire la connessione lato client

L'UUID (Universal Unique Identifier) è utilizzato per identificare univocamente il canale di comunicazione. Può essere ottenuto mediante l'applicazione di un algoritmo standard. Client e Server devono condividere il medesimo UUID per comunicare. Per tutti i dispositivi per i quali non è possibile effettuare un pairing esplicito, l'UUID da utilizzare è stabilito per convenzione.

Sensori e geolocalizzazione

La maggior parte dei device Android dispone di una serie di sensori built-in con i quali è possibile interagire. Generalmente, i sensori producono stream di dati raw ad elevata accuratezza e precisione. In Android sono supportate tre macrocategorie di sensori:

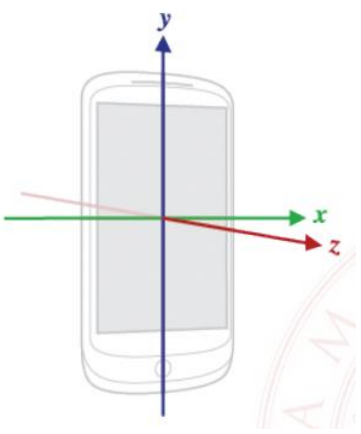
- sensori di movimento: misurano le forze di accelerazione e le forze di rotazione relative ai tre assi del SdR (ad esempio accelerometro e giroscopio)
- sensori ambientali: misurano parametri ambientali come temperatura, pressione e grado di illuminazione (ad esempio il barometro o il sensore di luce)
- sensori di posizione: determinano la posizione fisica del dispositivo (ad esempio i sensori di orientamento o il magnetometro)

L'Android Sensor Framework (ASF) costituisce la porzione di Framework Android per l'accesso e la gestione dei sensori di ciascun device Android. Tra le altre funzionalità, consente di:

- determinare quali sensori sono disponibili
- stabilire quali funzionalità sono disponibili per ciascun sensore e configurarne i parametri
- acquisire i dati raw prodotti continuamente dai sensori (specificandone il rate desiderato)
- registrare listener specifici per ciascun settore

I sensori hardware sono componenti fisici montati sul device che producono i propri flussi di dati misurando specifiche proprietà e condizioni ambientali, mentre i sensori software non sono associati a nessun componente fisico, bensì propongono i propri flussi dati come combinazione logica dei flussi dati sintetizzati dai sensori hardware.

La categoria di sensori più utilizzata e maggiormente diffusa sui diversi device è quella che fa riferimento ai sensori di movimento: appartengono a questa categoria, tra gli altri, l'accelerometro e il giroscopio, generalmente disponibili come sensori hardware. I sensori di movimento possono essere utilizzati per identificare il movimento del dispositivo con riferimento alle coordinate spaziali definite in termini solidali al dispositivo stesso. Esempi di applicazione dei sensori di movimento:



- determinare se un dispositivo viene agitato
- determinare la rotazione del dispositivo rispetto all'utente
- determinare se si sta viaggiando in auto oppure si sta camminando

Si utilizza un sistema di coordinate basato su un SdR a tre assi (X, Y, Z) solidali con il dispositivo. Il SdR è definito in funzione

dell'orientamento standard dello specifico dispositivo (portrait per la maggior parte dei dispositivi, ma in alcuni casi assumono come default il landscape). Quando il device è nella posizione standard, vale che:

- l'asse X è orizzontale e il suo asse positivo è definito verso destra
- l'asse Y è verticale e il suo asse positivo è identificato verso l'alto
- l'asse Z esce dallo schermo del device ortogonalmente agli altri due assi con direzione positiva

Si noti che non viene fatto nessuno swap degli assi X e Y quando il dispositivo è ruotato.
Alcuni sensori restituiscono i propri valori con riferimento al SdR terrestre e dunque non quello solidale al device: risulta sempre opportuno verificare questo aspetto prima di interpretare i valori restituiti da un sensore e applicare eventualmente matrici di trasformazione.

L'accelerometro, concettualmente, misura l'accelerazione (in m/s^2) applicata al dispositivo lungo i tre assi del SdR, includendo anche la forza di gravità. In condizione di equilibrio (device fermo, appoggiato su una superficie piana con lo schermo rivolto verso l'alto) i valori di accelerazione per gli assi X e Y sono prossimi al valore zero mentre il valore dell'accelerazione lungo l'asse Z è prossimo al valore (assoluto) dell'accelerazione di gravità, ossia 9,81. Qualora si vogliano ottenere i valori di accelerazione senza considerare la forza di gravità, si può far riferimento al sensore SW accelerometro lineare.

Il giroscopio, concettualmente, misura la rotazione (in rad/s) rispetto ai tre assi del dispositivo: la rotazione positiva è quella che è eseguita in direzione oraria. Generalmente, i valori ottenuti dal giroscopio sono combinati con i dati temporali per calcolare la rotazione del dispositivo con l'evolvere del tempo.

La Near Field Communication (NFC) è una tecnologia per connettività wireless a corto raggio di tipo contactless. Quando due dispositivi dotati di sensori NFC (rispettivamente initiator e target/tag) si trovano nelle vicinanze (entro i 5 cm) tra i due viene creata una rete ad-hoc di tipo P2P per lo scambio di un quantitativo limitato di dati. La lettura di un generico tag NFC può essere attuata interpretando i valori di uno o più record NDEF (NFC Data Exchange Format) memorizzati nel tag.

A partire dalla versione di Android 6.0 i permessi sono stati divisi in due categorie:

- normal: devono essere richiesti mediante dichiarazione esplicita nel manifest (ad esempio BLUETOOTH, INTERNET, NFC)
- dangerous: devono essere richiesti mediante dichiarazione esplicita nel manifest e l'utente deve esplicitamente accettare tale richiesta (ad esempio CAMERA, ACCESS_FINE_LOCATION, RECORD_AUDIO, READ_CONTACTS)

Il sistema genera un'eccezione se si utilizza codice che richiede un permesso dangerous per essere eseguito, senza aver preventivamente verificato se l'utente abbia o meno concesso esplicitamente tale permesso. Nel caso in cui l'utente non abbia concesso tale permesso è possibile richiederne l'attivazione direttamente all'utente. Il sistema è in grado di notificare l'applicazione in merito al fatto che l'utente abbia concesso o ignorato la richiesta di utilizzo del permesso.

DOMANDE FREQUENTI

- Quali sono le caratteristiche tecniche di Arduino?
- Quali sono le differenze tra polling e interrupt?
- Come si fa ad integrare un pulsante?
- Cosa succede quando faccio digitalWrite(HIGH) per accendere un led?
- Cos'è il PWM?
- Come si calcolano le resistenze?
- Cosa sono le resistenze di pull-up e pull-down? Quando si usano?
- Come funzionano gli interrupt handler?
- Come funzionano i timer? Cos'è il prescaler?
- Come fa il microcontrollore ad interagire con i pin? (porte)
- Che differenza c'è fra bus di comunicazione seriale sincrono e asincrono?
- Cos'è il bouncing? E il debouncing?
- Quali sono le differenze tra I^2C e SPI?
- Come funziona il sensore HC-SR04? Come funziona il PIR?
- Quali sono le principali differenze tra i vari tipi di motori?

- Quali sono i vantaggi della decomposizione in task?
- Cosa sono gli agenti?
- Che differenza c'è fra FSM sincrone e asincrone?
- Come viene scelto il periodo?
- A cosa dobbiamo fare attenzione scegliendo il periodo dello scheduler e delle FSM?
- Che cos'è il MEST?
- Cosa sono le eccezioni di overrun? Cos'è il parametro di utilizzo?
- Cos'è il jitter?
- Come può essere la priorità?

- Cos'è l'ESP? Quali sono le sue caratteristiche?
- Cos'è un RTOS? Quali sono le sue caratteristiche?
- Quali sono i benefici di un RTOS?

- Quali sono le principali tipologie di mezzi trasmissivi?
- Come si integrano i modelli di comunicazione nei sistemi a loop?

- Cosa sono gli Async Task e quando vengono usati in Android?