ESIoT 2023/24

Appunti - Alessandro Monticelli

Intro

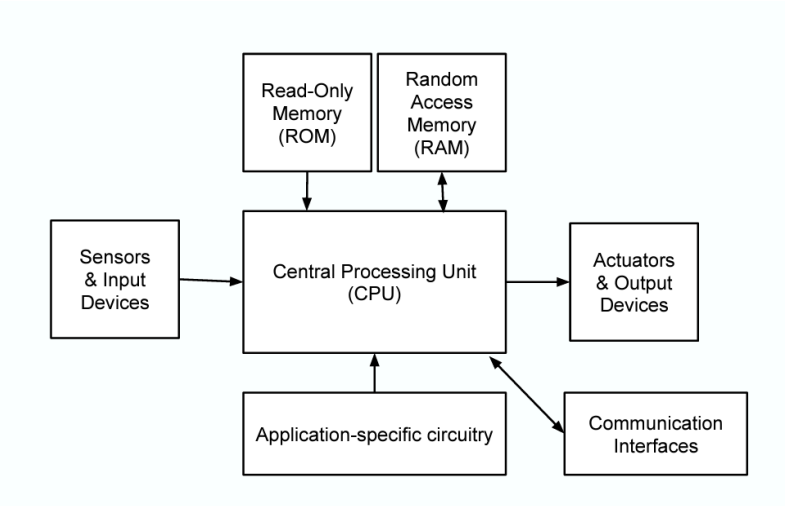
I sistemi embedded sono sistemi che permettono l'interazione del mondo fisico con quello digitale, attraverso sensori (elementi in grado di misurare fenomeni fisici) e attuatori (elementi che hanno un effetto misurabile sul mondo fisico).
Lo sviluppo dei sistemi embedded è cresciuto moltissimo negli ultimi anni, grazie allo sviluppo di tecnologie di comunicazione sempre più veloci, efficienti ed economiche.

Un sistema embedded esegue uno (o pochi) semplici tasks, che utilizzano poche risorse e sono spesso progettate ad-hoc per l'hardware su cui devono eseguire, ripetendo l'esecuzione "all'infinito" (never ending program),

L'hardware su cui eseguono questi programmi è generalmente molto semplice e con risorse limitate (CPU poco potente e a bassissimo consumo, poca memoria e richiedono poca potenza per essere alimentati).

I sistemi embedded sono usati in qualsiasi ambito, dalla clinica all'automotive, alla domotica, quindi, è importante che siano affidabili e reattivi.

Architettura Hardware dei sistemi embedded



CPU

3 tipi:

- General-purpose: processori programmabili, non specializzati nell'esecuzione di applicazioni o task particolari, il comportamento è definito dal software che eseguono.
- Single-purpose: Circuiti digitali per implementare una funzione o un programma specifico
- **Application-specific**: Processori programmabili, specializzati nell'esecuzione di una classe di applicazioni con caratteristiche in comune (molto usati nei sistemi embedded), spesso inseriti all'interno di SoC o MCU.

MCU

Micro Controller Unit. Integrano su un solo chip tutti i componenti necessari per per avere l'autonomia funzionale per l'esecuzione di task, molto usati nel mondo embedded. Hanno CPU, memoria volatile e persistente, pin di I/O, controller per gli interrupt.
Esempio: ATMega328P, usato in Arduino UNO, processore a 8 bit, 16Mhz, 2KB RAM, 32KB ROM.

Le soluzioni che integrano una MCU e tutta la circuiteria necessaria per eseguire task sono dette Single-Board-Micro-Controller.

SOC

System On a Chip. Integrano un sistema completo (CPU, memoria volatile e non, canali per l'I/O, controller per la rete, eccetera).
Sono in generale più potenti delle MCU, utilizzando spesso CPU single o multi core a 32 o 64 bit, con clock nell'ordine delle centinaia di Mhz, fino a diversi Ghz e memoria da centinaia di MB.
Esempio: Raspberry Pi, con i SoC BROADCOM BCM2837 e BROADCOM BCM2711, ESP8266 e ESP32 con le cpu Xtensa (LX6 per ESP32), 512Kb di RAM, 448Kb di ROM, pin GPIO, connessione WiFi e BLE, programmabile con diversi linguaggi/frameworks (C++, micropython eccetera).

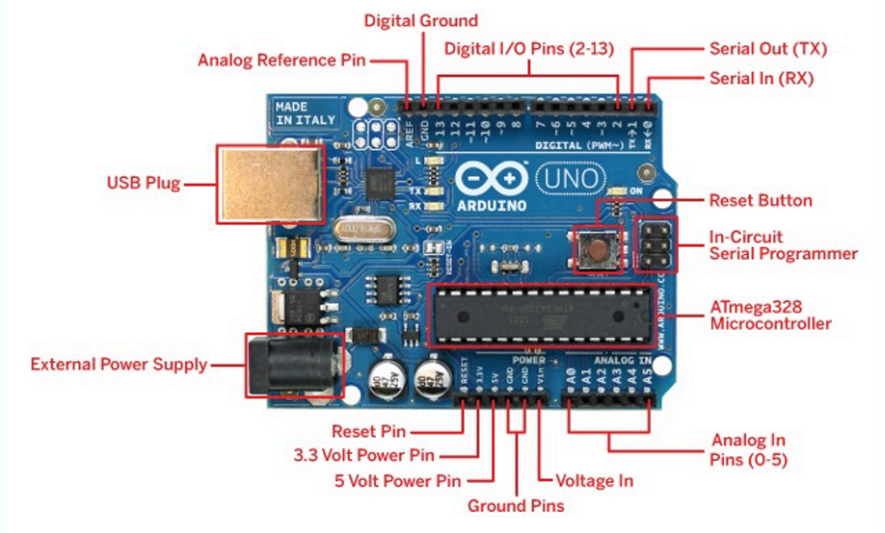
Componenti di un Microcontrollore

Arduino UNO

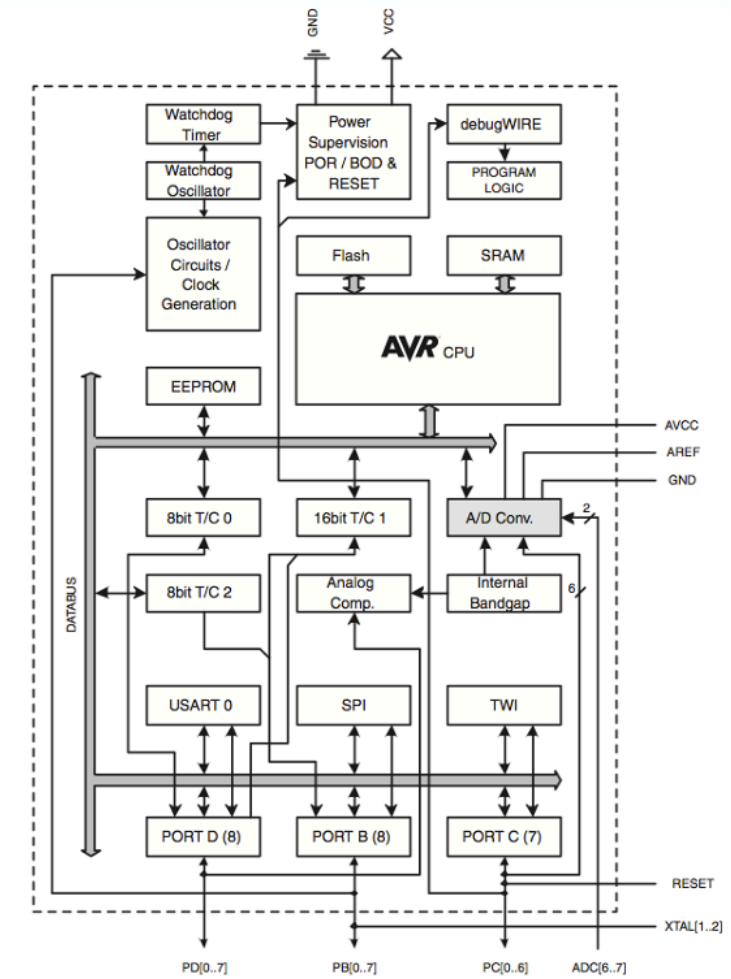
Single-Board MCU:

- ATMega328P:
  - 8bit, 16MHz;
  - 32Kb flash memory
  - 2Kb SRAM, 1Kb EEPROM
- 14 GPIO pin digitali, di cui 6 utilizzabili come PWM
- 6 GPIO analogici

- Header ICSP
- reset button
- Il connettore USB comunica con l'ATMega328P attraverso la sua interfaccia seriale (Arduino ha un chip apposito per fare ciò).



ATMega328P



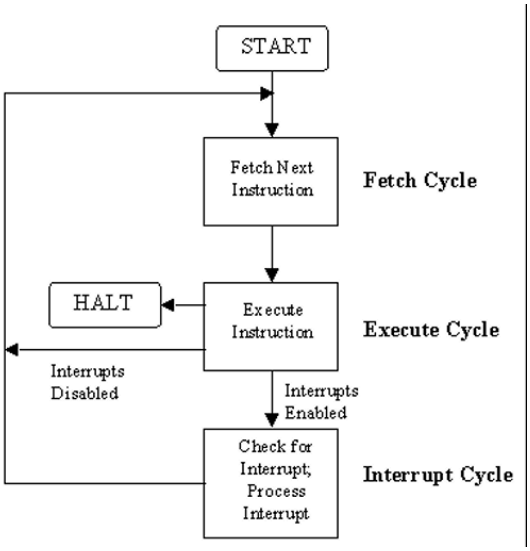
Programmazione di MCU

La programmazione di MCU avviene su un sistema host esterno (un PC), su cui i programmi sono scritti, compilati e poi caricati sulla MCU, attraverso HW specifico o comunicazione seriale.
Lo stack MCU non prevede la presenza di un sistema operativo, il programma viene caricato in memoria ed eseguito direttamente dalla CPU.

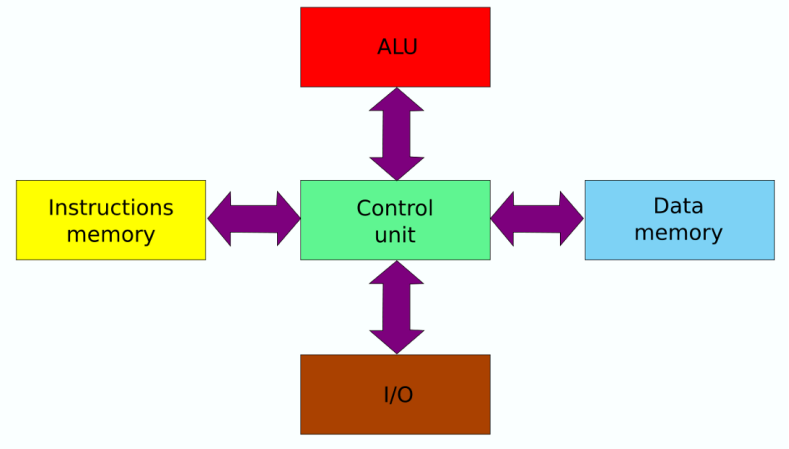
Su Arduino il programma viene caricato via USB, per mezzo di un bootloader (un programmino pre caricato in memoria).
I programmi sono sviluppati usando il framework di C++ Wiring e compilati utilizzando un compilatore apposito (avr-gcc), basato su tool GNU e altri progetti open source.

Architettura ATMega328P

La CPU di Arduino UNO (ATMega328P) non è basata sulla “classica” architettura di Von-Neumann:



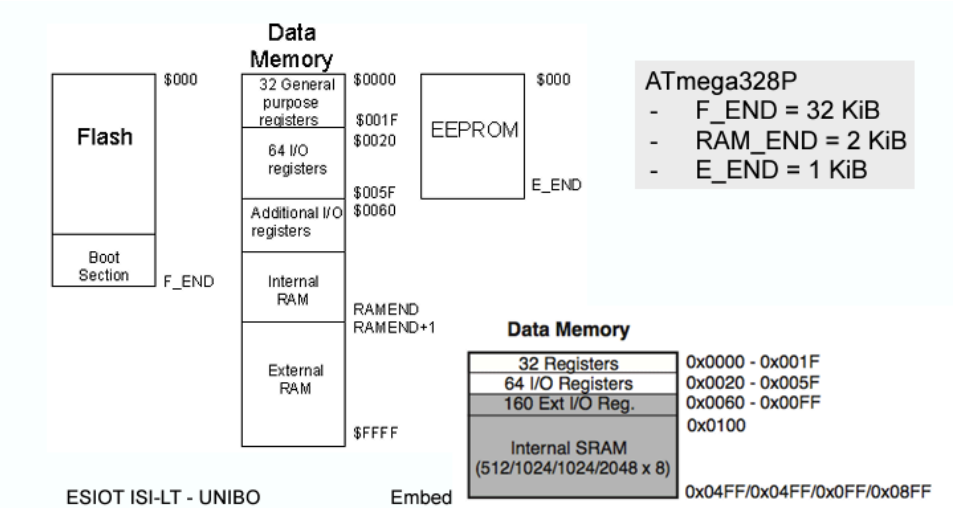
Ma su una sua variante, detta architettura Harvard, in cui le istruzioni e i dati sono su memorie fisicamente diverse (istruzioni in FLASH memory, veloce da leggere e lenta da scrivere, e dati in SRAM, veloce da scrivere e da leggere, ma più energivora e costosa).



L'ATMega328 ha una memoria FLASH da 32 KB (di cui 0.5KB sono usati dal bootloader) e una SRAM da 2KB, il chè significa che bisogna essere molto parsimoniosi nell'uso di strutture dati complesse e di stringhe. Inoltre lo stack è inserito in RAM.
La memoria FLASH è non volatile, veloce da scrivere, a basso consumo energetico ed economica, con una vita di circa 100.000 scritture.
La SRAM è volatile, veloce da leggere e scrivere (ATMega328P impiega circa 2 cicli di clock ad accesso), ma consuma più corrente.
La EEPROM è una memoria non volatile e lenta, ci si possono inserire altri dati come ad esempio costanti eccetera eccetera.

AVR8

L'AVR8 è il microprocessore all'interno dell'ATMega 328P, basato su architettura RISC a 8 bit per il codice e 16 bit per i dati, con clock di 16MHz.
Ha 32 registri general-purpose da 8 bit e un piccolo set di registri speciali:
Program Counter, Stack Pointer e SREG (registro di stato, contenente Carry Flag, Zero Flag e Interrupt Flag).
Sia i registri general purpose che i registri per l'I/O sono mappati nella parte bassa della memoria e sono accessibili usando indirizzi di memoria in quel range.



Controllo basilare: il super loop

Il super loop è il framework di controllo più semplice, molto comune in quanto non richiede il supporto di hw specifico. Si ha una fase di inizializzazione e un loop infinito, che esegue i task.

Pro:

- Semplice
- Portabile
- Affidabile e sicuro
- Efficiente

Contro:

- Timing poco preciso
- Fragile e poco flessibile
- Consumo energetico maggiore

Wiring framework

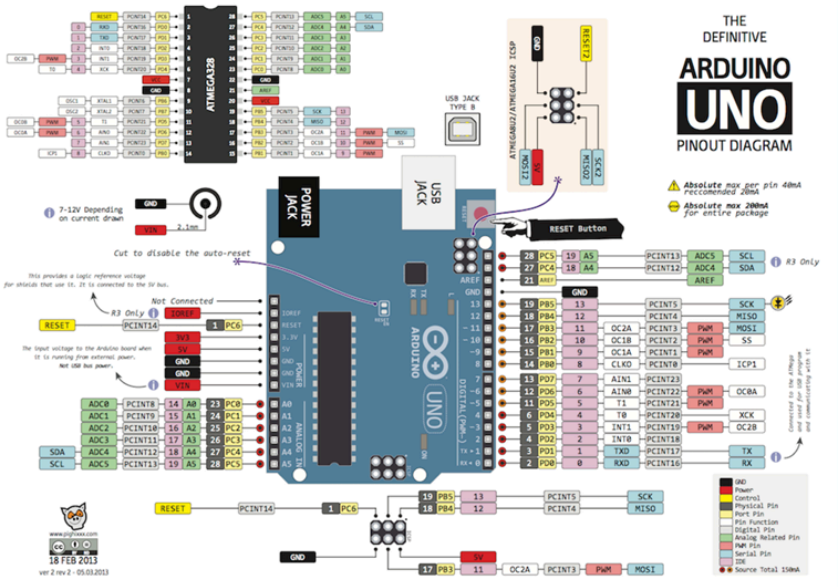
Framework C/C++ open source per programmare microcontrollori, ha una control architecture basata sul super loop, con due procedure di base: setup() e loop(), implementate dal programmatore.
Wiring mette a disposizione anche un set di API di base, per l'I/O, la gestione dei pin, la comunicazione seriale eccetera, oltre a procedure e funzioni di C/C++.

General-Purpose I/O (GPIO)

Ogni microcontrollore ha un set di pin che possono essere utilizzati direttamente per operazioni di I/O. I pin sono generalmente general-purpose, quindi possono essere programmati per essere utilizzati sia come input che come output in base alle necessità.
I pin possono essere sia analogici (assumono valori continui nel range GND-VCC, il valore viene poi convertito da un ADC in un segnale digitale in un range di valori deciso dalla risoluzione dell'ADC) che digitali (assumono due valori discreti, HIGH o LOW, 0 o 1).
Alcuni pin hanno altre funzioni: D0,D1: TTL serial interface; D2,D3: interrupts, D3,D5,D6,D9,D10,D11: PWM; D10,D11,D12,D13: SPI bus, D13: built in LED; A4,A5: I2C bus.

Pulse Width Modulation (PWM)

PWM permette di emulare un segnale analogico su pin digitali, attraverso la generazione di onde quadre periodiche. La tensione emulata in uscita viene definita dal duty cycle del segnale (la percentuale di tempo in cui il segnale digitale è HIGH sulla percentuale in cui il segnale è LOW).
La tensione media del segnale è pari al duty-cycle * VCC. Questo approccio non funziona su tutti i device e gli attuatori analogici, ma torna utile in diverse circostanze (come pilotare LED controllandone l'intensità o pilotare motori DC).



I due parametri con cui operano i pin sono la tensione (misurata in Volt, 5V su Arduino, 3.3V su RaspberryPi e ESP32/8266) e la corrente (misurata in Ampere, 40mA su Arduino, 12mA su ESP8266).
In genere è presente una resistenza interna detta di pull-up, che permette di fissare la tensione ad un valore massimo (+VCC), anche quando il pin configurato in input non è connesso a nessun circuito, in modo da evitare fluttuazioni nella tensione e conseguenti malfunzionamenti o rotture. Si usano resistori da decine di migliaia di kOhm.

Porte I/O

La MCU interagisce coi pin attraverso delle “porte”, che possono essere configurate come Input o come Output.
Le porte sono gestite da uno o più Special-Purpose-Registers, che si occupano di mantenere lo stato (il valore) e la configurazione (Input o Output) delle porte.
L’ATMega328P utilizza 3 porte da 8 bit (chiamate B,C e D), la porta D gestisce {D0-D7}, la porta B {D8-D13} e la porta C {A0-A6}.
Ogni porta è gestita da 3 registri DDR,PORT,PIN (e.g. la porta D è gestita da DDRD, PORTD e PIND).
DDR mantiene la direzione del pin (se 0 INPUT se 1 OUTPUT); PORT mantiene lo stato del pin in base alla configurazione, se INPUT il valore 0 significa che la resistenza di pull-up viene disattivata e 1 significa che viene attivata. Se OUTPUT il bit a 1 rappresenta HIGH e 0 rappresenta LOW; PIN è read-only, e mantiene per i pin di INPUT il valore del segnale in ingresso al pin, 1 se è HIGH, 0 se è LOW.

Interrupts

Gli interrupts sono un meccanismo fondamentale per reagire ad eventi triggerati da device esterni.
In generale una CPU in una MCU mette a disposizione uno o più pin detti IRQ (Interrupt Request) per ricevere segnali di interrupt.
Quando viene ricevuta una richiesta di interrupt la CPU sospende l’esecuzione delle istruzioni correnti (ad esempio il super loop) salvando nello stack l’istruzione successiva da

chiamare, e salta alla routine configurata per reagire all'evento, chiamata interrupt handler o interrupt service routine (ISR). L'indirizzo della ISR viene salvato in una tabella chiamata interrupt table. Nei microcontrollori gli IRQ sono tipicamente connessi ad uno o più pin GPIO.

Nell'ATMega328P ci sono 2 pin (D2 e D3) che possono essere configurati per generare interrupt esterni. In particolare i pin possono essere configurati per generare un interrupt in diverse condizioni:

- Sul RISING o FALLING edge del segnale
- Quando c'è un cambiamento del segnale
- Quando il segnale è LOW

Si possono anche disabilitare gli interrupt del tutto, ma a questo punto il sistema smette di reagire a qualsiasi evento esterno.

Timer

I timer giocano un ruolo fondamentale per ottenere comportamenti time-oriented.
Sono usati per:

- Misurare intervalli
- Creare segnali PWM
- Creare timeout e alarms
- Ottenere un multi-tasking con scheduling preemptive

Possono essere rappresentati come un semplice contatore che si aggiorna a livello hardware ad una data frequenza. Timer programmabili permettono di impostare questa frequenza, leggere il valore del timer corrente e associare interrupt per implementare comportamenti time-triggered, basati su time events.
ATMega328 ha 3 timer:

- timer0: contatore a 8 bit
- timer1: contatore a 8 bit
- timer2: contatore a 16 bit

Si possono associare interrupt ai timer in varie modalità, tra cui la CTC (Clear Timer on Compare Match), in cui gli interrupt sono generati quando il timer raggiunge un certo valore. A questo punto il timer si resetta e ricomincia a contare.
Sull'ATMega328P i timer sono aggiornati con una frequenza di 16MHz, ogni tick avviene ogni 1/16.000.000 secondi (63 ns), timer0 e timer1 sono da 8 bit, impiegano 256*63 ns (16.1 us) per ricominciare da 0, timer2 è da 16 bit, impiega 65536*63 ns (4.1ms) per ripartire da 0.

Questa frequenza si può modulare utilizzando un prescaler (1,8,64,256,1024):
timer speed [Hz] = 16MHz / prescaler
In questo modo è possibile ridurre la frequenza di aggiornamento dei timer.

Bus e protocolli

Ci sono due tipi principali di interfacce adibite alla comunicazione:

- Seriali: permettono di trasferire stream di bit in modo sequenziale usando un solo canale.
- Parallele: permettono il trasferimento di più bit utilizzando più linee.

Interfacce seriali

Asincrone

Una linea in ricezione(RX) e una in trasmissione(TX). La sincronizzazione avviene per mezzo di un protocollo, che può essere configurato tramite vari parametri

- baud rate: velocità di trasmissione in bit al secondo.
- data frame: endianness e dimensione del pacchetto dati, inclusi bit di controllo, parità e sincronizzazione
- bit di sincronizzazione: indicano dove inizia e dove finisce un data frame.
- bit di parità: controllo errori.

Il circuito che si occupa di gestire i dati che passano per l'interfaccia seriale è lo UART (Universal Asynchronous Receiver/Transmitter). È dotato di un bus dati da 8 o più bit, 3 bit di controllo e le 2 linee per la comunicazione.
Le interfacce possono essere full-duplex (i dispositivi possono trasmettere e ricevere contemporaneamente usando due linee separate) o half-duplex (o ricevo o trasmetto).

Sincrone

Si usa un segnale di clock per sincronizzare la comunicazione tra le parti, permettendo una velocità maggiore (ne sono esempi I2C e SPI).

I2C

Protocollo standard che permette di ridurre il numero di pin utilizzati (ne richiede solo 2), mantenendo una buona velocità. Usa un bus a 7 o 10 bit, con una velocità che varia da 10 kbit/s (slow-mode) a 100 kbit/s, versioni recenti hanno una fast-mode a 400 kbit/s, una fast-mode+ a 1 Mbit/s e high-speed mode a 3.4 Mbit/s.
Si basa su un'architettura master-slave, in cui il bus viene controllato da un device master (tipicamente il microcontrollore). Tutti i device collegati condividono due linee, una per il segnale di clock (SCL) e una seriale bidirezionale per i dati (SDA).
La comunicazione viene inizializzata dal master e lo slave può solo rispondere.
Il master invia i comandi in broadcast a tutti i device, ma risponde solo il target (indicato con un ID univoco formato da 7 bit).

SPI funziona in maniera simile ma usa 2 linee separate per trasmettere/ricevere, una MOSI (Master Out Slave In) e una MISO (Master In Slave Out), più una Slave Select (SS), oltre a quella di clock SCK.

Modellazione di sistemi embedded

Modellazione Object-Oriented

Due approcci principali per la modellazione:

- top-down: Si inizia dal dominio usando paradigmi ad alto livello, come OO.
- bottom-up: Si inizia dalle caratteristiche dell'hardware e si astrae sempre di più.

Paradigmi di programmazione come quello orientato agli oggetti sono prima di tutto paradigmi di modellazione. Specificano come si possono concettualizzare, rappresentare e modellare sistemi, e come si possa analizzare un problema definendone una soluzione in un dominio applicativo.
Modello: Rappresentazione degli elementi rilevanti di un sistema, astraendoli da quelli non rilevanti. Gli aspetti rilevanti riguardano la struttura, il comportamento e le interazioni del sistema.

Modellazione di software embedded basato su microcontrollori

Suddividiamo il sistema in due parti principali:

- Controller
    - Incapsula il controllo e l'application logic che riguarda i task da eseguire.
    - Accede, usa o osserva le risorse passive.
    - Integra un comportamento proattivo e reattivo
    - Si basa su architettura control-loop, come il super loop o l'event loop.
- Elementi controllati
    - modellano le risorse e i dispositivi (passivi) gestiti dal controller per eseguire i task, incapsulando funzionalità utili al controller.
    - Interfacce ben definite
    - Stato ed eventi osservabili
    - Riusabilità e controllo
    - Rappresentano il Model nel paradigma OO.

Control Loop

Modello di esecuzione basato su un control loop (o super loop), in cui ad ogni ciclo il controller legge l'input e sceglie l'azione da eseguire in base allo stato, eventualmente aggiornandolo.

Modellare il controller: Agenti

Nel paradigma OO gli oggetti sono in genere elementi passivi, che non incapsulano il control flow dell'applicazione. Il controller è invece un'entità attiva, con il proprio flow control logico. Per modellare il controller, utilizziamo il concetto di "agente".
Un agente è una entità attiva, con un proprio control-flow logico, modellata per eseguire alcuni task, che siano reattivi ai dati e agli eventi rilevati nell'ambiente (attraverso i sensori), e che eseguano azioni che hanno un effetto misurabile sull'ambiente (attraverso gli attuatori).

Decomposizione ed esecuzione Task-based

Lavori complessi possono essere decomposti in task, eseguiti dal control loop.
L'agente esegue uno o più task concorrentemente, ed il comportamento di ogni task si descrive con una macchina a stati finiti.

Sistemi complessi multi agente

Sistemi embedded molto complessi possono essere distribuiti, includendo un insieme di sottosistemi interagenti e cooperativi, utilizzando quindi sistemi multi-agente.

Modellazione con macchine a stati finiti

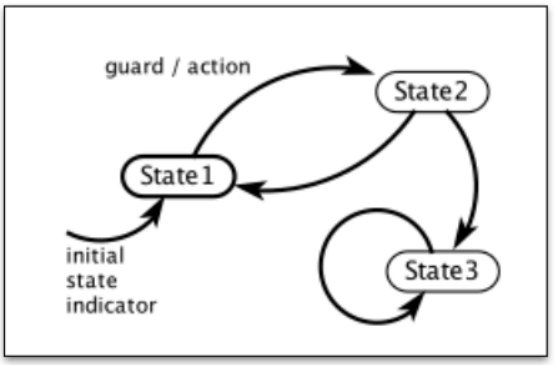
I sistemi embedded possono includere componenti sia discreti (con cambiamenti di stato veloci, improvvisi e atomici) che continui (con cambiamenti di stato più lenti e continui nel tempo).
I componenti discreti possono essere modellati efficacemente con **macchine a stati finiti**.

Stati

Intuitivamente, lo stato di un sistema rappresenta la sua condizione in un certo istante di tempo e, in generale, lo stato influisce su come il sistema reagisce agli input.
Formalmente definiamo lo stato come la codifica di tutto ciò del passato che ha un effetto sulla reazione del sistema agli input correnti o futuri.
Lo stato è un “riassunto” del passato.

Macchine a stati finiti

Una **macchina a stati** è un **modello** di un sistema in cui le **dinamiche discrete mappano** valutazioni dell’**input su** valutazioni dell’**output** ad ogni reazione, dove ogni mappa potrebbe dipendere dallo stato corrente.
Una **macchina a stati finiti** è una macchina a stati dove l’**insieme degli Stati** possibili è **finito**. Se il numero di stati è ragionevolmente piccolo, la FSM può essere rappresentata in un diagramma degli stati:



States =
{ State1, State2, State3 }

Initial state = State1

Transizioni di stato

Le transizioni di stato governano le dinamiche discrete della macchina a stati e la mappatura dell’input sull’output.
Le transizioni possono avvenire tra stati differenti o verso lo stesso stato (self-transition) e sono caratterizzate da una **guardia** e da un’**azione**.
La guardia determina se la transizione possa avvenire in seguito a una reazione.
È rappresentata da un predicato (espressione booleana) che diventa vera quando deve avvenire la transizione (transizione abilitata).
L’azione specifica l’output prodotto da ogni reazione, è un assegnamento di valori all’output, ogni output non menzionato nella transizione è considerato assente.

Macchine a stati finiti: sincrone e asincrone

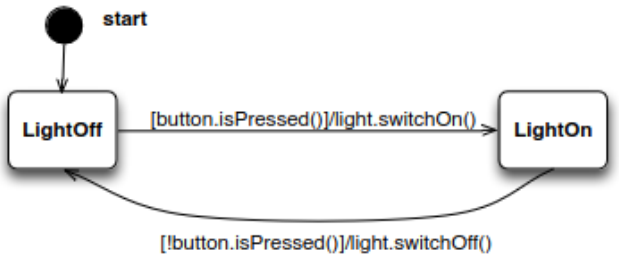
Una FSM non specifica quando la valutazione debba avvenire per poter triggerare una reazione.
Ci sono due modelli possibili:

- FSM asincrone, chiamate event-triggered: la valutazione avviene ad ogni evento di input, quindi è l'ambiente a determinare quando le reazioni sono valutate ed attivate.
- FSM sincrone, chiamate time-triggered: la valutazione avviene periodicamente, ad un intervallo regolare chiamato periodo.

Modellazione dell'I/O

Quando applichiamo le FSM ai sistemi embedded, **input e output** sono **mappati in variabili** usate **per definire guardie e azioni**. Le variabili di **input** sono **modificate dall'ambiente** in cui il sistema lavora, mentre le variabili di **output** sono modificate **dal sistema** attraverso le azioni, controllando l'ambiente esterno.
Oltre all'I/O le **variabili** si possono usare per **definire lo stato** globale **del sistema**. Le guardie sono definite usando sia variabili di input che di output.

Esempio: button-LED



In un modello di FSM corretto, la **computazione** delle azioni **deve sempre terminare**, **non** devono esserci **loop infiniti** e, in teoria, le **azioni** dovrebbero essere **istantanee**; la valutazione di una condizione non deve modificare lo stato delle variabili.

Macchine a stati finiti sincrone

Il comportamento dei **sistemi embedded** è **tipicamente time-oriented**.
Il tempo è parte delle specifiche di guardie e azioni, e **può essere usato per** definire **comportamenti periodici** o regolari.
Le **FSM sincrone** forniscono un modo efficace di **gestire il passare del tempo**. In queste FSM lo "**step**" (tick) della macchina è **eseguito periodicamente**, come se la FSM avesse un **clock interno** e le **transizioni** possano **avvenire solo al tick** del clock.
Per implementare **FSM sincrone si fa** generalmente **uso dei programmable timers**, programmati per generare un interrupt alla frequenza desiderata. L'interrupt dovrebbe portare all'esecuzione dello step della FSM.

Time-scale differenti

Un sistema potrebbe dover gestire **vari intervalli di tempo o periodi** al proprio interno (ad esempio si potrebbe volere che un LED sia acceso per 500 ms e spento per 750 ms).
L'approccio generale prevede di considerare il **massimo comune divisore** come **periodo** della macchina (nell'esempio 250 ms).

Sampling

Con **sampling** si indica la **lettura periodica dei sensori** a frequenze specifiche.
Sampling rate: frequenza della lettura periodica.
Scegliere il periodo è cruciale nel design di una FSM sincrona. Il **periodo** deve essere abbastanza **piccolo per non perdere eventi**, e **grande** abbastanza **per evitare overrun**.
Per scegliere il **sampling-rate** si sceglie, in genere, **il più grande** valore del **periodo** che garantisca di **non perdere eventi** (e quindi il corretto funzionamento del sistema).

Minimum Event Separation Time

Il **MEST** è definito come il **più piccolo intervallo** di tempo che passa **tra** due **eventi di input**, dato l'ambiente in cui il sistema opera.
In una **FSM sincrona**, scegliendo un **periodo** più **piccolo** del **MEST**, avremo la garanzia che **tutti gli eventi** siano **rilevati**.
Nell'esempio del button-LED, scegliendo un periodo più piccolo del tempo che trascorre tra la pressione ed il rilascio del button (il MEST appunto) abbiamo la garanzia di rilevare tutte le pressioni del bottone, e quindi tutti gli stati in cui il bottone viene premuto.

Architetture Task-based per FSM

Per **sistemi embedded complessi** è necessario sviluppare approcci per **decomporre** e **modularizzare** il comportamento e **le funzionalità del sistema**.
Il **comportamento** del software embedded viene **decomposto in** un insieme di **task concorrenti**.
Ogni task rappresenta un'**unità di lavoro specifica** e ben definita.
Il comportamento di **ogni task** può essere **descritto da una FSM**. Il comportamento globale è il risultato dell'esecuzione e interazioni delle FSM concorrenti.

Vantaggi

Modularità: Ogni task è un modulo indipendente e l'interfaccia del modulo in questo caso è l'insieme di variabili e oggetti in Input o Output che il task utilizza, i quali possono anche essere condivisi con altri task.
Ciò riduce la complessità del singolo task, rendendo più semplice il debug e favorendo la riusabilità del codice.

Problemi

I task sono concorrenti, la loro esecuzione si sovrappone nel tempo e concettualmente ogni task ha la propria logica.
I task potrebbero presentare dipendenze che creano interazioni tra task che devono essere gestite appropriatamente tramite meccanismi di coordinazione, come ad esempio l'uso di variabili o oggetti condivisi.

Scheduler cooperativo

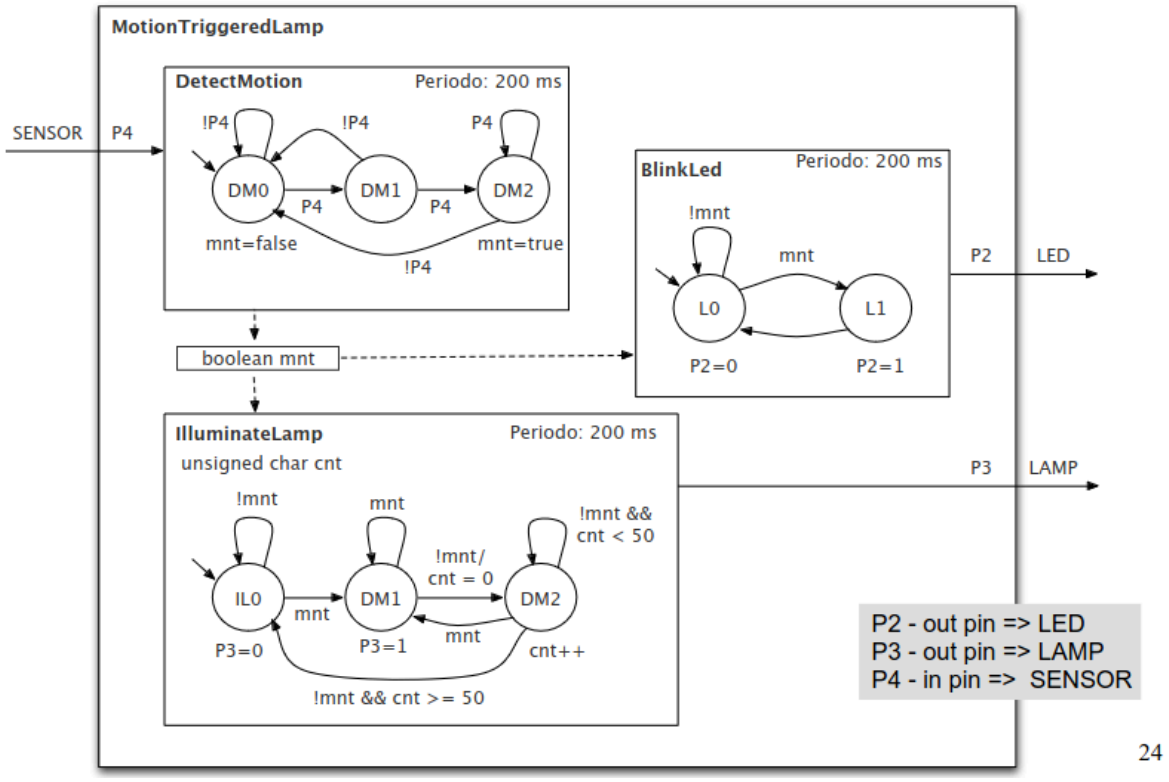
Dato che nella maggior parte dei casi i **task hanno periodi differenti** tra loro, è **necessario** che vi sia una **struttura di controllo**, in quanto bisogna tener traccia del periodo di ogni task, e **chiamare ogni task con il proprio periodo**. Per far ciò si utilizza uno **scheduler cooperativo**.
Lo scheduler **tiene traccia della lista di task** da eseguire e, utilizzando un algoritmo **round-robin**, chiama i task in base al proprio **periodo P** (**pari al massimo comune divisore dei periodi dei task**), ad ogni periodo P, scorre la lista dei task e chiama il metodo tick del task corrente. **Internamente utilizza un timer** per realizzare il comportamento periodico.
Attenzione, è **necessario che l'esecuzione** di ogni tick abbia **durata** sempre **inferiore del periodo** dello scheduler.

Dipendenze tra task

I **task** possono avere **dipendenze** che richiedono varie forme di interazione

- **Dipendenze temporali**
- **Produttore/Consumatore**
- **Data-oriented**

Si possono gestire queste dipendenze **attraverso variabili condivise**.

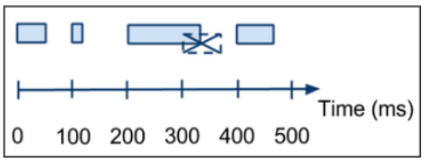


Scheduling e utilizzo della CPU

Nel design di FSM sincrone assumiamo che le azioni siano istantanee, ma in realtà le azioni hanno sempre una durata e dobbiamo controllare che il periodo scelto sia compatibile con quella durata per evitare problemi.

Eccezioni di overrun

Quando il tempo di esecuzione di un'azione supera il periodo dello scheduler, causando la generazione di un interrupt prima della conclusione dell'interrupt handler del task attualmente in esecuzione.



sync machine with period = 100ms

Depending on the state, different actions are executed. At t = 200ms, the duration of actions exceed the period…

Le eccezioni di overrun possono essere rintracciate analizzando il codice assembly, stimando la durata totale delle azioni e controllando se nel caso peggiore questa durata eccede il periodo dello scheduler.

Worst Case Execution Time

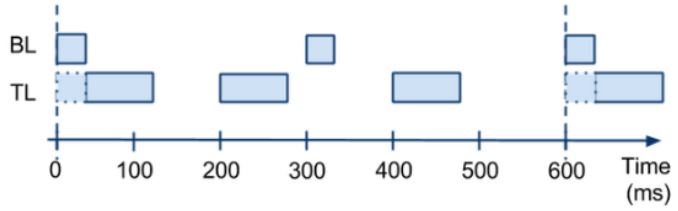
Definiamo il il parametro di utilizzo della CPU come:

U = (tempo CPU del task / tempo totale) * 100 (%)

Il WCET è il tempo di esecuzione di un task, in un periodo, nel caso peggiore. Nel caso di stati o transizioni di stato multiple, consideriamo la sequenza più lunga di istruzioni.
Se otteniamo un valore di U > 100% allora si potrebbe verificare una eccezione di overrun, in tal caso per risolvere il problema possiamo:

- Aumentare il periodo della FSM
- Ottimizzare la sequenza di istruzioni per ridurre il WCET
- Decomporre le operazioni lunghe in sequenze più piccole
- Usare una MCU più veloce
- Rimuovere funzionalità del sistema.

L'analisi dell'utilizzo della CPU è fondamentale in sistemi con molti task, se i task hanno lo stesso periodo allora il WCET si calcola sommando i WCET dei task individuali.
Altrimenti (come nella maggioranza dei casi) il WCET si può calcolare considerando gli iper-periodi, ossia periodi pari al minimo comune multiplo dei periodi. Ad esempio:



BL ha periodo 300ms e TL ha periodo 200ms.
In questo caso, il pattern si ripete ogni 600 ms.
BL ha un WCET di 20ms e TL di 90ms.
In 600 ms, BL esegue 600/300 = 2 volte, mentre TL esegue 600/200 = 3 volte.
Il parametro U = (2 * 20 ms + 3 * 90 ms) / 600 ms = 55%
Riassumendo: per calcolare il parametro U per $T_1 \ldots T_n$ su un microcontrollore M, si determina la durata R di una singola istruzione di M, si analizza il codice di ogni task T_i calcolando T_i.WCET, contando il numero di istruzioni in un tick e moltiplicandolo per R; si determina l'iper-periodo H come il minimo comune multiplo dei periodi dei task (T_1.H, T_2.H, …).

$$U = ((H/T_1.\text{periodo})*T_1.\text{WCET} + (H/T_2.\text{periodo})*T_2.\text{WCET} + \ldots)/H*100$$

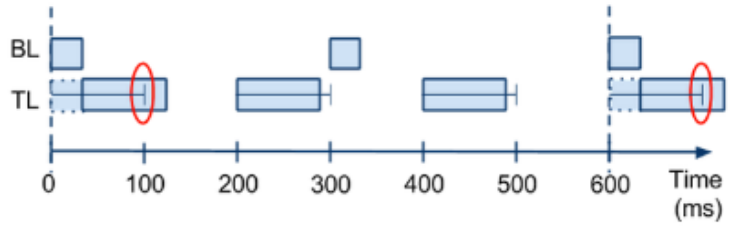
Se U > 100 ci sarà overrun, altrimenti per i singoli task (non è detto che globalmente sia così) non si verifica overrun.

Jitter

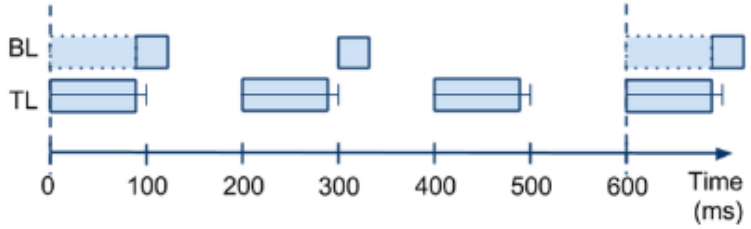
Il jitter è il tempo che intercorre tra l'istante in cui un task è pronto ad eseguire e l'istante in cui è effettivamente chiamato.
Strategie di scheduling diverse possono portare a jitter diversi. In generale dare priorità ai task più brevi porta a minimizzare il jitter medio.

Deadline

La deadline è definita come l'intervallo di tempo all'interno del quale un task deve essere eseguito dopo che è pronto per essere chiamato.
Se un task non viene chiamato prima della deadline, si ha una missed-deadline exception che può risultare in una system failure.
Se la deadline non viene specificata allora si assume il periodo del task come tale.
La strategia di scheduling impatta sul gitter e quindi di conseguenza sulla possibilità di ottenere missed-deadline exceptions.
Esempio:
Dando priorità a BL si ha una missed deadline exception.



Mentre dando priorità a TL, aumenta il jitter per BL ma non si verificano missed deadline exception:



Scheduling delle priorità

Può essere statico o dinamico, nel primo caso le priorità sono assegnate a design time e non cambiano a runtime; nel secondo caso le priorità vengono assegnate dinamicamente, in base a strategie differenti.

Architetture event-based

Il meccanismo degli interrupt può essere sfruttato per creare architetture event-based di alto livello.
Gli interrupt possono essere concepiti come eventi a basso livello che interrompono il control flow o il super-loop per eseguire l'handler corrispondente.
Gli interrupt rendono possibile evitare il polling nella gestione di sensori ed input, non è il processore a interrogare continuamente lo stato del sensore, ma concettualmente è il sensore a notificare i cambiamenti appena avvengono.

Pattern observer

Elementi:

- Fonte degli eventi: mette a disposizione una API per permettere agli observers (listeners) di essere notificati quando viene rilevato un evento.
- Eventi generati
- Observer di una fonte: Mette a disposizione una API per essere notificato degli eventi.

Implementato attraverso interrupt.
Il codice del listener viene eseguito dagli interrupt handlers, ciò rappresenta una limitazione in quanto i listener non possono eseguire computazioni a lungo termine né usare interrupt al loro interno (gli interrupt innestati non sono supportati).
Per evitare race conditions bisogna fare in modo che il codice del super loop acceda alle variabili modificate dai listener in modo atomico, disabilitando e riabilitando gli interrupt.

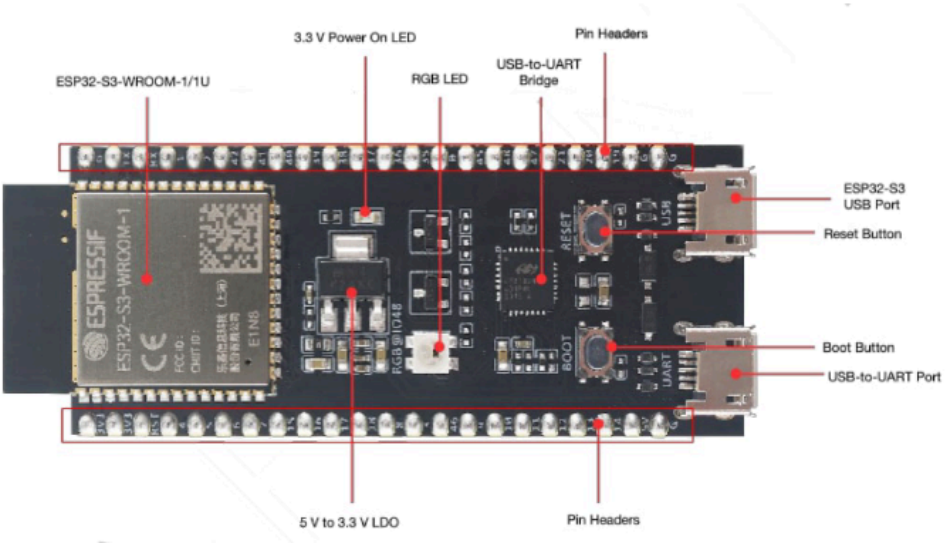
FSM asincrone

Sfruttano gli interrupt e implementano FSM event-triggered.
A differenza delle FSM sincrone la valutazione delle reazioni viene eseguita quando un evento avviene.
Non esiste un concetto di periodo, comportamenti periodici si possono simulare utilizzando eventi temporali generati da un timer.
In questo caso sia uno scoppiamento totale tra la generazione dell'evento (fatta dagli interrupt) e le reazioni/azioni eseguite dal super loop. Non ci sono le limitazioni dovute all'esecuzione dei listeners negli interrupt handler, e preserviamo la reattività e la responsività del sistema.
Le race condition sono inesistenti per costruzione, dato che gli interrupt handler non eseguono codice che accede alle variabili.
L'implementazione si basa sul pattern Reactor, anche chiamato architettura a event-loop.

Sistemi Embedded basati su SoC e rtOS

Componenti di un SoC

ESP32



ESP32 è una famiglia di SoC low-cost pensati per l'utilizzo nel mondo dell'IoT.

- Consumano poca energia
- Integrano WiFi e Bluetooth (anche BLE)
- Hanno una cpu Xtensa LX6
    - anche dual core
    - 32bit
    - clock di 160-240 MHz
- 320 KB di RAM
- 448 KB di ROM
- 34 pin GPIO
- 12 pin ADC
- 2 DAC a 8 bit
- 4 interfacce SPI
- 2 interfacce I2S
- 2 interfacce I2C
- 3 UART.

Sistemi Operativi Real Time

I SoC hanno risorse sufficienti per far girare un sistema operativo real-time, appositamente pensato per i sistemi embedded.
Sono compatti, efficienti, affidabili e prevedibili (deterministici).
A differenza degli OS desktop, i rtOS sono in grado di eseguire un'applicazione alla volta, tipicamente multi-thread o multi-core.
Nota: thread e task sono definizioni sostanzialmente uguali nell'ambito dei rtOS.

Caratteristiche rtOS

- Real Time:
    - Devono reagire e gestire gli eventi e gli input entro intervalli di tempo definiti.
    - Devono gestire l'esecuzione di task che hanno delle deadline.
- Sottoclassi:
    - Hard real-time: Le deadlines sono requisiti di correttezza;
    - Soft real-time: Si possono violare le deadlines in casi speciali, da gestire.
- Determinismo, bisogna poter prevedere:
    - Il tempo di esecuzione dei task;
    - Il tempo massimo per eseguire un'azione, ricevere input o reagire a un interrupt;
    - Il numero di cicli per eseguire la stessa operazione deve essere costante.

Un rtOS permette di evitare il polling sfruttando la possibilità di utilizzare un context-switch a livello della CPU, supporta lo scheduling preemptive e minimizza l'overhead per il task management.
Ha un sistema di allocazione dinamica della memoria, si possono usare semafori o mutex per controllare l'accesso all'hardware e permette l'utilizzo di framework o middleware che forniscono vari servizi (stack TCP/IP, stack USB, file system eccetera…)

Struttura

Hanno struttura simile a un OS normale, con un kernel, un file system, interfacce di rete, USB e a volte grafiche.
La CPU ha due modalità di esecuzione (user e kernel).

Task e Scheduling nei rtOS

Definizioni

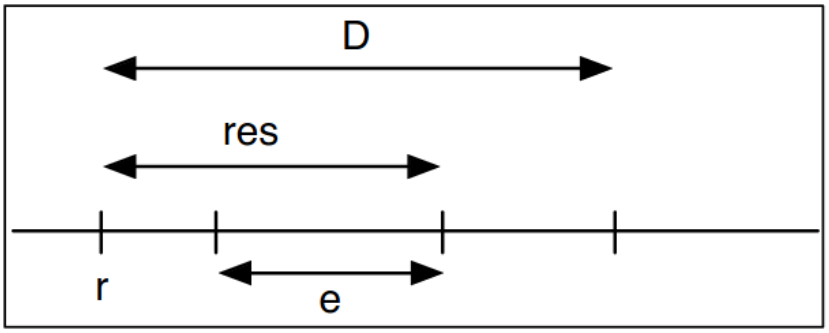
- Processo: un programma in esecuzione, con memoria propria.
- Thread (o task): un flusso di controllo indipendente all'interno di un processo
- Multi-threading: esecuzione parallela di più thread.

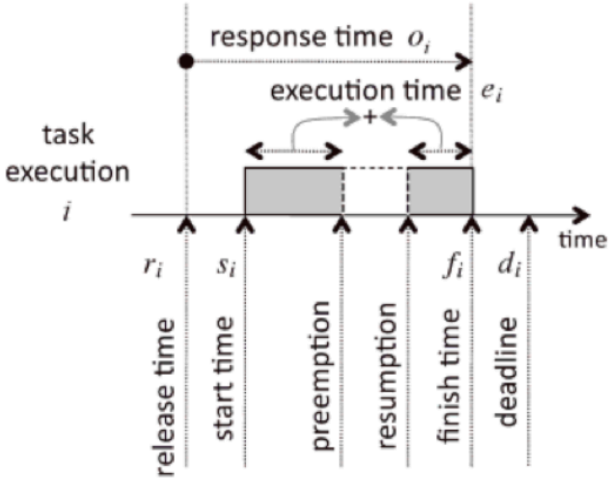
Context switch
Il contesto di un task è formato dalle informazioni necessarie a far ripartire l'esecuzione di un thread, quando avviene un context switch. Viene salvato quando un task viene interrotto, e ripristinato quando riprende l'esecuzione.
Il context switch si verifica con l'interruzione del thread corrente, facendo passare il controllo al kernel o a un altro task.

Parametri Temporali

Le proprietà temporali dei task sono definite da 4 parametri fondamentali:

- Release time: **r**
  - Il momento in cui il task entra nella ready-queue
- Execution time: **e**
  - Il WCET del task
- Response time: **res**
  - L'intervallo di tempo tra il release e il completamento del task
- Deadline: **D**
  - Tempo massimo per l'esecuzione





Task periodici e aperiodici

I task si dividono in 3 categorie:

- periodici: Rilasciati a intervalli regolari (periodo)
- aperiodici: Rilasciati arbitrariamente
- sporadici: aperiodici ma con deadlines

È necessario definire il comportamento da assumere quando una deadline non viene rispettata.

Tipi di scheduling

- Big-loop: scheduling cooperativo in un super-loop.
    - Si fa il polling di ogni task per vedere se può essere eseguito. Si può effettuare il polling sequenzialmente o in base a una coda di priorità. È poco efficiente e lento.
- Round-robin: Il processore viene allocato ad ogni turno per i task pronti.
    - Cooperativo: I thread sono eseguiti fino al completamento o una chiamata bloccante
    - Preemptive: I thread hanno una time slice assegnata e vengono inseriti in una coda d'attesa (preemption)
- Priority-based: Il processore esegue il thread con la priorità più alta.
    - Per i thread con la stessa priorità si usa round-robin (preemption)
    - Se c'è un thread con una priorità più alta da eseguire viene eseguito.

Priority scheduling

Tutti i rtOS supportano lo scheduling priority-based, possibilmente integrandolo con round-robin.
Ciò permette di aumentare la responsività, ma allo stesso tempo porta con sé dei costi (computazionali ed energetici) per la preemption, oltre alla possibilità di starvation dei processi e il presentarsi di un problema, chiamato "Di inversione delle priorità".

rtOS sincroni e asincroni

I rtOS **sincroni** utilizzano un clock hardware per suddividere il tempo CPU in frame. Un programma deve essere suddiviso in task in modo che ogni task possa essere completato in un singolo frame nel caso peggiore.
Lo scheduling è abbastanza semplice, si fa uso di una scheduling table (una lista) che tiene traccia dei task da eseguire in ogni frame; se l'esecuzione del task prende più tempo di un frame allora il task va suddiviso in sotto-task da eseguire in più frame. La durata del frame diventa un parametro importante nel design del sistema. Si usa un approccio hard-realtime.

Nei rtOS **asincroni** si usa uno scheduler preemptive priority based, che schedula i task in base alle loro priorità. I task con la stessa priorità sono schedulati usando uno scheduling round-robin con preemption.
L'implementazione dello scheduler è basata su scheduling events che invocano lo scheduler e sono generati quando un task viene inserito in coda nella ready-queue o da un timer, ad una certa frequenza, in modo da tenere traccia del tempo che passa per realizzare la preemption nel round-robin.

Assegnamento delle priorità

Un aspetto fondamentale dello scheduling nei rtOS asincroni è assegnare le priorità ai task in modo che ogni task sia eseguito in tempo, senza violarne la deadline.
Un assegnamento corretto delle priorità (che rispetti questo obbligo) è detto feasible assignment.

Esempio:

- Con due task T_1, T_2
  - T_1: $p_1 = D_1 = 2$, $e_1 = 1$ (p_1: periodo, D_1: deadline, e_1: tempo di esecuzione)
    - Eseguito ogni 2 unità di tempo e ha bisogno di 1 unità di tempo per eseguire.
  - T_2: $p_2 = D_2 = 5$, $e_2 = 2$
    - Eseguito ogni 5 unità di tempo e ha bisogno di 2 unità di tempo per eseguire.
- Ci sono due possibilità: $T_1 > T_2$ o $T_2 > T_1$

Se eseguissi prima T_2 di T_1 violerei continuamente la deadline di T_1.
Deve essere quindi $T_1 > T_2$ per avere un feasible assignment.

Algoritmi di scheduling

Rate Monotonic (RM)

Algoritmo di scheduling con priorità fisse. Le priorità sono decise staticamente e non cambiano, si calcolano con l'inverso del periodo. Più piccolo è il periodo maggiore sarà la priorità.
Dato uno scheduler preemptive a priorità fisse, ed un set finito di task $T = \{T_1,T_2,T_3,...\}$ con associati i periodi $P = \{p_1,p_2,p_3,...\}$ senza ulteriori limitazioni, se esiste un feasible assignment, allora l'assignment definito da RM è sempre feasible.

Earliest Deadline First

Questo algoritmo calcola dinamicamente le priorità dei task in base alle dipendenze. Si può usare solo con scheduler dinamici e asincroni.
EDF assegna priorità più alta ai task con deadline più vicine. Come RM anche per EDF esiste un teorema analogo, se esiste un feasible assignment allora EDF lo inserisce in un set di N task T con CPU usage parameter $U < 1$:

$$U = \sum e_i / p_i \text{ per ogni } T_i \text{ con } i = \{0,1,...,N-1\}.$$

EDF non si può sempre applicare, in quanto alcuni task (ad esempio gli interrupt handler) devono avere priorità fisse.
EDF porta con sé un overhead importante che potrebbe non essere accettabile in alcuni casi.

Protocolli di comunicazione IoT

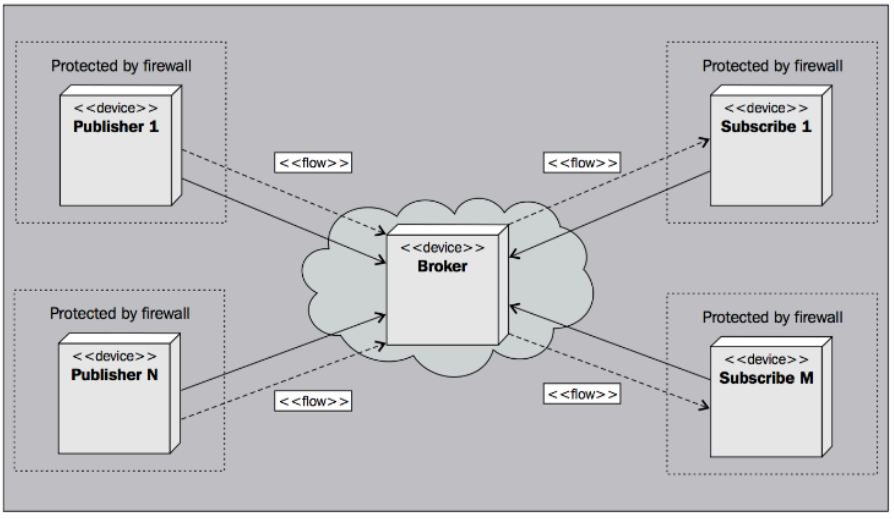
CoAP

Constrained Application Protocol, un Internet Application Protocol specializzato per dispositivi e reti limitati (a bassa potenza).
L'idea è quella di fornire un protocollo che permetta ai nodi più limitati di comunicare con internet, limitando l'overhead legato ai protocolli standard che funzionano con REST API.
CoAP fornisce un set limitato delle REST API HTTP, ottimizzate per comunicazione M2M (machine-to-machine). Una semplificazione di HTTP con header compatto e un numero di opzioni ridotto, così come il numero di metodi ridotto a GET, POST, PUT e DELETE.
Si possono inviare messaggi confermabili (si vuole un ack di ritorno) e non-confermabili (Non serve un ack).
Oltre a ciò offre nuove funzionalità ed estensioni di HTTP: supporto al multicasting, architettura con event subscription and notification (risorse osservabili emettono notifiche quando si scatenano eventi).

MQTT

Il problema principale di CoAP (o HTTP) sta nella difficoltà di attraversare i firewall. Questi non solo bloccano le richieste in ingresso, ma nascondono la rete interna dietro un solo IP. A meno chè il firewall non blocchi le comunicazioni in uscita, allora si possono attraversare i firewall se tutti i terminali in una comunicazione agiscono da client di un message broker in comune, che è al di fuori del firewall ed è quindi accessibile da tutti.
Il broker agisce da server, ma tutto quello che fa è consegnare i messaggi ai client.
Un protocollo che fa uso di questo meccanismo è MQTT (Message Queue Telemetry Transport).
Il protocollo MQTT si basa sul pattern publisher/subscriber, in cui:

- I publisher si connettono al broker inviando messaggi
- I subscriber si connettono allo stesso broker e si iscrivono ai contenuti ai quali sono interessati
- Il message broker si assicura che i messaggi pubblicati siano inviati ai subscribers interessati.



Topic e albero dei topic

I contenuti sono identificati da topic.
Quando si pubblica un contenuto, il publisher sceglie se il contenuto debba essere trattenuto dal server o no. Se trattenuto il subscriber riceverà l'ultimo valore pubblicato quando si iscrive al topic.
I topic sono ordinati in una struttura ad albero. Lo slash in avanti (/) è il delimitatore, quando ci si iscrive a un contenuto, un subscriber può iscriversi a un topic specifico usando il suo path, o a tutto un branch usando la wildcard #

Livelli di QoS

Ci sono 3 livelli di QoS:

- **unacknowledged service**: il messaggio viene consegnato al massimo una volta ai subscribers.
- **acknowledged service**: ogni ricevente manda un ack alla ricezione, se non si riceve l'ack l'informazione può essere inviata di nuovo. Si ha la certezza che il messaggio sia consegnato almeno una volta ad ogni subscriber.
- **assured service**: non si manda solo un ack alla ricezione, ma se ne manda uno anche alla trasmissione. In questo modo si ha la certezza che il content sia consegnato esattamente una volta ad ogni subscriber.

Sicurezza

MQTT offre un'autenticazione con username e password in chiaro (poco sicura), si può usare MQTT su una connessione crittografata con SSL/TLS, ma è necessario che i client validino i certificati dei server.
Altri metodi di protezione includono l'uso di chiavi pre concordate o di certificati client-side per identificarli.
In alternativa si potrebbero usare metodi proprietari per criptare il contenuto dei messaggi, ma ciò riduce ovviamente l'interoperabilità che è uno degli obiettivi di MQTT.

Un riassunto

Feature	HTTP	CoAP	MQTT
Request/Response	✔	✔	✘
Publish/Subscribe	✘	✘	✔
Multicast	✘	✔	✘
Events or Push	✔	✔	✔
Bypasses firewall	✘	(✔)	✔
Federation	✘	✘	✘
Authentication	✔	✔	✔
Network Identity	(✔)	(✔)	✘
Authorization	✘	✘	✘
Encryption	✔	✔	✔
End-to-end encryption	✘	✘	✘
Compression	✔	✘	✘
Streaming	✔	✘	✔
Reliable messaging	✘	✘	✔✔
Message Queues	✘	✘	✘

Credits

Thanks to prof. A. Ricci (a.ricci@unibo.it) for the material.