

ESP32

Potenza a pochi spiccioli



Nel 2016 nasce l'ESP32, il nuovo processore dell'azienda Espressif System per sostituire lo storico ESP8266. Questo piccolo componente è una vera rivoluzione nel mondo dell'elettronica programmabile hobbystica ma viste le caratteristiche che descriveremo di seguito, possiamo allargare il suo campo d'utilizzo anche ad altri settori.

Con questo processore l'azienda Espressif, ha racchiuso in un unico chip a bassissimo costo una CPU potente, il Wi-Fi e il Bluetooth.

Come spesso accade, nel corso degli anni c'è stata un'evoluzione del componente, ma le caratteristiche comuni a quasi tutte le versioni sono le seguenti:

- **Architettura Xtensa Dual-Core:** La maggior parte dei chip ESP32 possiede **due core fisici indipendenti**. Questo permette di eseguire più compiti in parallelo (ad esempio, delegare la gestione della rete Wi-Fi al Core 0 e il codice utente/sensori al Core 1).
- **Connettività integrata:** Wi-Fi (standard 802.11 b/g/n) e Bluetooth (spesso in versione BLE Bluetooth Low Energy) nativi sul chip.
- **Sistema operativo in tempo reale (FreeRTOS):** Grazie alla potenza di calcolo, l'hardware nasce per essere gestito da un sistema operativo leggero (FreeRTOS), che permette il multitasking avanzato.

ARCHITETTURA

La tecnologia utilizzata per la realizzazione di questo componente è la **XTENSA** sviluppata dall'azienda **Tensilica** oggi acquisita dalla **Cadence**.

L'architettura di un processore può essere definita come la struttura logica e il progetto funzionale del chip. L'architettura definisce il set di istruzioni, l'organizzazione della memoria, la quantità e la tipologia dei registri interni, e getta le basi per determinare l'efficienza energetica e le prestazioni del processore.

Ad esempio abbiamo sicuramente sentito nominare l'architettura x86 quando parliamo del processore dei nostri PC o l'architettura ARM quando invece parliamo dei processori degli Smartphone o Tablet, l'architettura XTENSA ha invece preso piede oggi nel mondo dell'IoT. Vediamo di seguito un confronto tra le 3:

CARATTERISTICA	ARCHITETTURA x86	ARCHITETTURA ARM	ARCHITETTURA XTENSA
ISTRUZIONI	CISC (Complex Instruction Set Computer)	RISC (Reduced Instruction Set Computer)	RISC personalizzabile (Reduced Instruction Set Computer)
STRUTTURA MEMORIA	Harvard Modificata nel chip Von Neumann sulla scheda esterna	Harvard Modificata nel chip Von Neumann sulla scheda esterna	Harvard Modificata nel chip Von Neumann esternamente
PRODUTTORI	INTEL, AMD.	APPLE, SAMSUNG, QUALCOMM	TENSILICA, ESPRESSIF
VANTAGGIO	Potenza di calcolo, retrocompatibilità	Efficienza energetica	Personalizzabile e bassi consumi
CONSUMI	Medio alti	Bassi	Bassissimi
UTILIZZO	PC, Server, Workstation	Tablet, Smartphone	IoT, Sistemi embedded

La quasi totalità dei microprocessori moderni, x86, ARM e Xtensa, adottano un'**architettura Harvard modificata** a livello interno (con cache e bus separati per velocizzare le operazioni sulla CPU), pur mantenendo spesso uno spazio di memoria principale unificato (stile Von Neumann) per la RAM esterna e lo storage.

I processori con tecnologia Xtensa adottano un'architettura Harvard a livello interno (con bus e cache separati per dati e istruzioni), e mappano la Flash e la RAM all'interno di uno spazio di indirizzamento unificato a 32 bit. Questo permette alla CPU di accedere in modo efficiente e flessibile a entrambe le risorse.

Nonostante *ARM* e *Xtensa* condividano la stessa filosofia di base (**RISC** ed estrema efficienza energetica), la grande differenza di Xtensa risiede nella sua **natura personalizzabile**, che permette ai produttori come Espressif di inserire istruzioni hardware dedicate (ad esempio per il Wi-Fi) direttamente nel silicio del chip.

Questo è il vero punto di forza di questa architettura, in effetti i produttori che utilizzano XTENSA, non acquistano un chip ben definito, ma acquistano un **"core" di base personalizzabile con il linguaggio TIE** (Tensilica Instruction Extension).

Con questo linguaggio il produttore può dire al generatore: *"Voglio aggiungere un nuovo blocco hardware che esegua una complessa operazione matematica di crittografia Wi-Fi in un solo colpo, anziché far fare 20 passaggi alla CPU tradizionale"*. Il generatore TIE prende quel pezzo di codice e **disegna automaticamente i circuiti elettronici (porte logiche e registri)** necessari per farlo funzionare nel silicio.

A fronte del vantaggio della personalizzazione c'è però un grosso svantaggio che è legato al fatto che potendo inserire nuove istruzioni, ogni chip dovrà avere un adeguato compilatore che riconosca quelle istruzioni. Ciò significa che il produttore del processore, in questo caso Espressif, dovrà fornire il compilatore (GCC ad esempio) adatto per quel tipo di dispositivo.

DUAL CORE

Il core (nucleo) è il "cervello" vero e proprio all'interno del processore: è la parte fisica che si occupa di prelevare le istruzioni, decodificarle ed eseguirle.

I processori possono essere monocore o multicore:

- **Monocore.** Avendo un solo nucleo il processore può fare una sola cosa alla volta. Se deve gestire una connessione Wi-Fi e contemporaneamente leggere i dati da un sensore, deve "saltare" continuamente da un compito all'altro ad elevata velocità (multitasking simulato).
- **Multicore.** Ha letteralmente più cervelli indipendenti all'interno dello stesso chip. Questo significa che può eseguire più operazioni nello stesso identico istante (parallelismo reale). Nel caso dei sistemi Multicore uno dei problemi è la gestione condivisa della memoria da parte dei due core. Per risolvere questo problema oltre che utilizzare le singole memorie cache dei due core, si utilizzano sistemi di arbitraggio hardware o software gestiti dal sistema operativo.

L'ESP32 è un processore DUAL CORE, ciò significa che è in grado di seguire in parallelo due differenti programmi e questo lo rende un processore adatto al mondo dell'IoT in quanto in genere uno dei due core può occuparsi della connessione wifi mentre l'altro può tranquillamente gestire le funzioni previste nel programma.

Sistema operativo FreeRTOS

Gestire un sistema dual core non è una cosa semplice dal punto di vista software, basti pensare la gestione della memoria condivisa, la gestione degli interrupt e tante altre cose. Per questo motivo nei sistemi dual core è preferibile (se non quasi indispensabile) utilizzare un sistema operativo FreeRTOS (**RTOS**, *Real-Time Operating System*). Questo è un sistema operativo molto leggero e soprattutto "realtime". Ciò significa che viene garantita l'esecuzione di un'operazione (un task) entro un tempo massimo prestabilito chiamato *deadline* o scadenza.

Nei Sistemi Operativi normali (es. Windows11) Se il computer è occupato a scaricare un file o a fare una scansione antivirus, il programma che stai usando può "fermarsi" per qualche millisecondo o ritardare un'operazione. Al sistema operativo non importa *quando* esegue un calcolo, purché lo finisca prima o poi.

In un Sistema Operativo RTOS (Real-Time): Il fattore tempo è critico. Se un sensore medico o un sistema di frenatura di un'auto deve inviare un segnale, l'RTOS deve elaborarlo entro un numero fisso e garantito di millisecondi (o microsecondi). Se manca quella scadenza, il sistema fallisce.

Nel mondo dei microcontrollori e dell'IoT, i sistemi operativi real-time si dividono solitamente in due categorie:

- **Hard Real-Time:** Il mancato rispetto del tempo limite provoca il blocco totale del sistema. Viene utilizzato in sistemi di sicurezza dove è tassativo rispettare i tempi previsti.
- **Soft Real-Time:** Il rispetto delle scadenze è importante, ma un piccolo ritardo occasionale non causa un disastro, riduce solo le prestazioni. Viene utilizzato nei sistemi dove un ritardo di un task può non arrecare danno (esempio la decodifica di un video).

FreeRTOS è un kernel Real-Time deterministico (perché il tempo per compiere una determinata operazione è noto) e a priorità preventiva, o anche preemptive (perché ogni task viene etichettato con un livello di priorità in modo che quello più prioritario può bloccare un task ad un livello di priorità inferiore), **il che lo rende idoneo per applicazioni Hard Real-Time**. Tuttavia, in contesti IoT complessi come l'ESP32 (dove il sistema gestisce stack di rete wireless, periferiche asincrone e memorie Flash esterne), il sistema opera generalmente nel campo del Soft Real-Time, garantendo comunque un'ottima reattività ed evitando il blocco totale del dispositivo.

I compiti del Sistema Operativo FreeRTOS in un sistema dual core sono i seguenti:

- **Gestione dei Task (Multitasking).** È il compito principale del kernel. FreeRTOS prende il codice del programma permette la suddivisione in blocchi indipendenti chiamati **Task** (es. un task legge il sensore, uno gestisce il display, uno invia i dati via Wi-Fi). L'RTOS decide quale task deve eseguire la CPU in un determinato momento, alternandoli a velocità elevate
- **Schedulazione a Priorità Preventiva (Preemptive Scheduling).** FreeRTOS assegna a ogni task un livello di priorità. Grazie al meccanismo *preemptive* (preventivo). Se un task importante deve essere eseguito, FreeRTOS interrompe all'istante il task meno importante che sta girando sulla CPU è il sistema operativo che gestisce la gerarchia in modo automatico.

- **Sincronizzazione e Comunicazione tra Task.** Quando più task girano in parallelo spesso hanno bisogno di passarsi dei dati o di accedere alle stesse risorse. FreeRTOS fornisce strumenti di sicurezza hardware/software come ad esempio:
 - **Le Code (Queues):** Per inviare dati in modo sicuro da un task all'altro.
 - **I Mutex e i Semafori:** I "lucchetti" digitali che impediscono a due task di modificare la stessa variabile nello stesso istante, evitando la corruzione dei dati.
- **Gestione del Tempo e dei Ritardi (Delays).** FreeRTOS offre funzioni precise per gestire il fattore temporale (es. vTaskDelay). Permette a un task di mettersi in pausa per un numero esatto di millisecondi, liberando completamente la CPU in modo che possa dedicarsi ad altre attività utili invece di sprecare cicli di clock in attese a vuoto (busy waiting).
- **Gestione della Memoria (Heap Management).** FreeRTOS si occupa di allocare e liberare dinamicamente lo spazio nella memoria RAM del microcontrollore ogni volta che si crea o si elimina un task, una coda o un semaforo, ottimizzando le risorse limitate tipiche dei sistemi embedded.

In sintesi, FreeRTOS fa da direttore d'orchestra: prende l'hardware (compresi i due core dell'ESP32) e lo trasforma in una piattaforma ordinata, sicura e reattiva dove decine di operazioni diverse sembrano avvenire contemporaneamente.

Quando si utilizza l'IDE di Arduino per programmare l'ESP32, viene utilizzato il framework ufficiale di Espressif ESP-IDF, che viene installato nell'IDE quando si aggiunge la scheda sul gestore di schede di Arduino. In questo modo oltre che poter scrivere le normali istruzioni in Wiring (digitalWrite, pinMode, delay, ecc...) avremo che insieme al nostro programma strutturato con il loop ed il setup, verrà caricato anche il codice per il FreeRTOS.

Per comprendere quanto detto finora vediamo un esempio base con un programma dove ci sono due task che girano per ognuno dei due core, con priorità differenti.

La struttura consigliata per l'ESP32 è quella di dedicare il core 0 per il Wi-Fi di sistema ed ESP-IDF, il Core 1 per il codice utente/Arduino.

Inoltre come vedremo nel successivo esempio nel loop non verrà inserito codice ma solamente l'istruzione per eseguire il task del loop ogni 10 secondi. L'uso di vTaskDelay nel loop() è fondamentale proprio per evitare che il watchdog interpreti l'inattività o un ciclo chiuso come un blocco del sistema, causando un reboot imprevisto.

```
1 #include "freertos/FreeRTOS.h"//Include la libreria principale di FreeRTOS per poter usare
2 #include "freertos/task.h" //le funzioni di gestione dei task, dei ritardi e dei meccanismi di sincronizzazione.
3 #include "freertos/semphr.h" //nell'IDE di Arduino per ESP32 viene spesso inclusa automaticamente.
4
5 int contatoreGlobale = 0; //dichiaro una variabile nella memoria globale a cui entrambi i core potranno accedere
6
7 SemaphoreHandle_t mutexMemoria; //Dichiara una variabile di tipo SemaphoreHandle_t
8 //che conterrà il riferimento al Mutex (il "lucchetto").
9 //Servirà per fare in modo che un solo task alla volta
10 //possa accedere alla variabile contatoreGlobale.
11
12
```

```

13 //Di seguito vengono definiti 4 task, un task non deve mai terminare, per questo si usa un ciclo infinito.
14 //Se un task finisce il suo codice, il microcontrollore andrebbe in errore o si riavvierebbe.
15 //void *pvParameters) è un requisito obbligatorio imposto da FreeRTOS
16 //per qualsiasi funzione che voglia diventare un "Task".
17 //è un puntatore generico e serve per passare un qualsiasi tipo di
18 //dato all'interno del task, nell'esempio non vengono passati dati
19
20 // =====
21 // CORE 0 - TASK 1: Gestione di sottofondo (Priorità 1 - Bassa)
22 // =====
23 void taskCoreZeroBasso(void *pvParameters) {
24     while (true) {
25         //Il task prova a entrare nella "zona protetta" chiedendo il mutex.pdMS_TO_TICKS(100)
26         //significa:"Se c'è il lucchetto la variabile è occupata da un altro task, aspetta al massimo 100 msec".
27         //Se invece ottiene il permesso, la funzione restituisce pdTRUE e il codice all'interno dell'if viene eseguito.
28         if (xSemaphoreTake(mutexMemoria, pdMS_TO_TICKS(100)) == pdTRUE) {
29             contatoreGlobale += 1;
30             Serial.print(" [Core 0 - BASSO] Contatore: ");
31             Serial.println(contatoreGlobale);
32             xSemaphoreGive(mutexMemoria); //toglie il lucchetto sulla variabile condivisa
33         }
34         vTaskDelay(pdMS_TO_TICKS(2000)); // Gira ogni 2 secondi
35     }
36 }
37
38 // =====
39 // CORE 0 - TASK 2: Gestione di rete/log (Priorità 2 - Media)
40 // =====
41 void taskCoreZeroMedio(void *pvParameters) {
42     while (true) {
43         //Il task prova a entrare nella "zona protetta" chiedendo il mutex.pdMS_TO_TICKS(100)
44         //significa:"Se c'è il lucchetto la variabile è occupata da un altro task, aspetta al massimo 100 msec".
45         //Se invece ottiene il permesso, la funzione restituisce pdTRUE e il codice all'interno dell'if viene eseguito.
46         if (xSemaphoreTake(mutexMemoria, pdMS_TO_TICKS(100)) == pdTRUE) {
47             contatoreGlobale += 5;
48             Serial.print(" [Core 0 - MEDIO] Contatore: ");
49             Serial.println(contatoreGlobale);
50             xSemaphoreGive(mutexMemoria); //toglie il lucchetto sulla variabile condivisa
51         }
52         vTaskDelay(pdMS_TO_TICKS(1000)); // Gira ogni 1 secondo
53     }
54 }
55
56 // =====
57 // CORE 1 - TASK 3: Elaborazione dati (Priorità 2 - Media)
58 // =====
59 void taskCoreUnoMedio(void *pvParameters) {
60     while (true) {
61         //Il task prova a entrare nella "zona protetta" chiedendo il mutex.pdMS_TO_TICKS(100)
62         //significa:"Se c'è il lucchetto la variabile è occupata da un altro task, aspetta al massimo 100 msec".
63         //Se invece ottiene il permesso, la funzione restituisce pdTRUE e il codice all'interno dell'if viene eseguito.
64         if (xSemaphoreTake(mutexMemoria, pdMS_TO_TICKS(100)) == pdTRUE) {
65             contatoreGlobale += 10;
66             Serial.print("[Core 1 - MEDIO] Contatore: ");
67             Serial.println(contatoreGlobale);
68             xSemaphoreGive(mutexMemoria); //toglie il lucchetto sulla variabile condivisa
69         }
70         vTaskDelay(pdMS_TO_TICKS(800)); // Gira ogni 800 ms
71     }
72 }
73

```

```
74 // =====
75 // CORE 1 - TASK 4: Sensore Critico (Priorità 3 - Alta)
76 // =====
77 void taskCoreUnoAlto(void *pvParameters) {
78     while (true) {
79         //Il task prova a entrare nella "zona protetta" chiedendo il mutex.pdMS_TO_TICKS(100)
80         //significa:"Se c'è il lucchetto la variabile è occupata da un altro task, aspetta al massimo 100 msec".
81         //Se invece ottiene il permesso, la funzione restituisce pdTRUE e il codice all'interno dell'if viene eseguito.
82         if (xSemaphoreTake(mutexMemoria, pdMS_TO_TICKS(100)) == pdTRUE) {
83             contatoreGlobale += 50;
84             Serial.println("==> [Core 1 - ALTA] ATTENZIONE! Sensore critico letto! Contatore: " + String(contatoreGlobale));
85             xSemaphoreGive(mutexMemoria); //toglie il lucchetto sulla variabile condivisa
86         }
87         vTaskDelay(pdMS_TO_TICKS(500)); // Gira ogni mezzo secondo
88     }
89 }
90
91 void setup() {
92     Serial.begin(115200);
93     delay(1000);
94     Serial.println("\n--- Avvio Multi-Task su Multi-Core (ESP32) ---");
95
96     //Crea fisicamente il Mutex in memoria prima di far partire qualsiasi task.
97     //Senza questo passaggio, i task non avrebbero il "lucchetto" da condividere.
98     mutexMemoria = xSemaphoreCreateMutex();
99     //Verifica che il Mutex sia stato creato con successo
100     //se la RAM fosse esaurita, restituirebbe NULL.
101     if (mutexMemoria != NULL) {
102         //ora vengono creati i task ed assegnati ai core
103         //i primi due campi sono il nome del task ed un nome descrittivo
104         //4096 sono i bites per lo stack di ogni core
105         //NULL perché in questo caso non si passano variabili
106         //il livello di priorità è il numero tra i due NULL
107         //il secondo NULL per non avere il codice identificativo del TASK, per non gestirlo
108         //il numero del core dove dovrà girare il TASK
109
110         // --- CREAZIONE TASK SUL CORE 0 ---
111         xTaskCreatePinnedToCore(taskCoreZeroBasso, "T_C0_Low", 4096, NULL, 1, NULL, 0);
112         xTaskCreatePinnedToCore(taskCoreZeroMedio, "T_C0_Med", 4096, NULL, 2, NULL, 0);
113         // --- CREAZIONE TASK SUL CORE 1 ---
114         xTaskCreatePinnedToCore(taskCoreUnoMedio, "T_C1_Med", 4096, NULL, 2, NULL, 1);
115         xTaskCreatePinnedToCore(taskCoreUnoAlto, "T_C1_High", 4096, NULL, 3, NULL, 1);
116     }
117 }
118
119 void loop() {
120     // Lasciamo vuoto: tutto il lavoro è gestito dai 4 task indipendenti.
121     //il loop è come se fosse un task eseguito in questo caso ogni 10 secondi
122     //l'unica operazione in questo è l'assegnazione del tempo con vTaskDelay
123     vTaskDelay(pdMS_TO_TICKS(10000));
124 }
```

CARATTERISTICHE HARDWARE E TABELLE COMPARATIVE

Quanto detto finora riguarda il microprocessore ESP32 che si trova in diversi modelli, ma quello che generalmente troviamo in commercio è il Modulo o la Scheda.

Prima di iniziare facciamo chiarezza sui termini utilizzati.

SoC (System On Chip)

L'ESP32 è a tutti gli effetti un SoC cioè un dispositivo che al suo interno contiene i seguenti componenti:

- La CPU (in questo caso il processore Xtensa dual core)
- La memoria SRAM interna (per i dati e le variabili a esecuzione rapida).
- L'hardware per il Wi-Fi ed il Bluetooth, modem e radiofrequenza (RF).
- Gli acceleratori crittografici hardware, circuiti interni per eseguire rapidamente i calcoli legati alla crittografia.
- Diverse periferiche digitali e analogiche (Controller PWM, ADC, DAC, interfacce I2C, SPI, UART, Touch Sensors).



Il SoC da solo non può funzionare. Ha bisogno di componenti esterni fondamentali per avviarsi, come il cristallo per il clock, i condensatori di disaccoppiamento, la memoria Flash esterna (perché il codice non risiede dentro il SoC) e un'antenna RF calibrata. Per questo motivo, raramente viene acquistato e saldato direttamente dai piccoli produttori, ma viene fornito internamente a dei moduli già predisposti.

MODULO

Il modulo è una piccola scheda elettronica racchiusa in un contenitore metallico con dei pin di collegamento esterni ed al suo interno contiene:

- Il SoC.
- La memoria Flash SPI esterna (solitamente 4MB o 8MB saldata a bordo) per il programma.
- Il cristallo oscillatore e i componenti passivi di supporto.
- Un'antenna integrata (stampata direttamente sul PCB o con connettore U.FL per antenna esterna).
- Una schermatura metallica (la finestrella d'argento marchiata) che serve a isolare i disturbi elettromagnetici e a superare le rigide certificazioni FCC/CE per la commercializzazione radio.



Viene acquistato dalle aziende che progettano prodotti elettronici commerciali personalizzati (es. un termostato smart o un inverter). L'azienda non deve preoccuparsi di progettare la complessa sezione radio o di saldare il chip a mano: salda direttamente il modulo finito sulla propria scheda madre (PCB custom).

SCHEDA

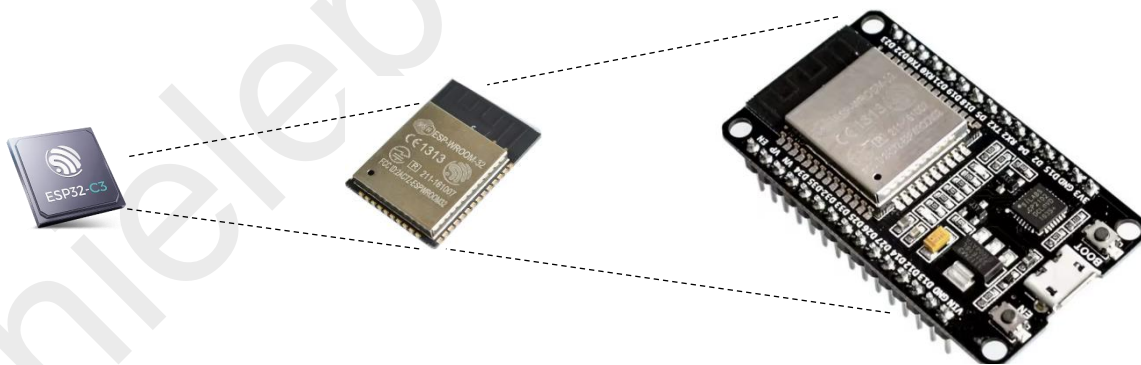
E' la scheda elettronica che contiene il Modulo con tutti i componenti elettronici esterni necessari per la programmazione ed il debug al suo interno troviamo:

- Un **Modulo** (o direttamente il SoC con la sua Flash).
- Un **chip di conversione USB-UART** indispensabile per collegare l'ESP32 alla porta USB del PC, caricare il codice e leggere il Monitor Seriale.
- Un **regolatore di tensione** che prende i 5V della porta USB e li riduce a 3.3V stabili per alimentare l'ESP32.
- I **pulsanti fisici** di Reset e Boot (per mettere la scheda in modalità di programmazione).
- I **pin header** (i pin metallici) disposti a passo 2.54mm, che portano all'esterno tutti i GPIO del chip consentendo di infilarlo facilmente su una breadboard o di collegare sensori con normali cavetti.



Maker, hobbisti, studenti e sviluppatori che hanno bisogno di una piattaforma pronta all'uso, robusta e protetta per scrivere software, testare sensori e fare debugging prima di passare alla produzione industriale anche se in diversi casi viene direttamente utilizzata nella produzione finale per piccoli quantitativi.

Possiamo riassumere quanto detto con la seguente immagine:



Nel nostro caso noi acquisteremo le SCHEDE ma per l'applicazione finale dobbiamo tener conto del SoC contenuto al suo interno che può avere caratteristiche completamente differenti.

A questo link, la ditta Espressif mette a disposizione un comodo strumento di comparazione dei vari SoC.

<https://products.espressif.com/#/product-comparison?names=&type=SoC>

Ad esempio la scheda dell'immagine sopra è una **ESP32 DEVKIT V1** che monta un modulo **ESP-WROOM-32** che al suo interno ospita un SoC **ESP32-D0WD** con le seguenti caratteristiche:

- *Processore Xtensa® dual-core 32-bit LX6*
- *Frequenza massima 240MHz*
- *WIFI IEEE 802.11b/g/n; 2.4 GHz; HT20/40; up to 150 Mbps*
- *Bluetooth BR/EDR + Bluetooth LE v4.2*
- *SRAM 520kB Memoria per dati.*
- *ROM 448kB (contiene il codice di bootloader e le funzioni essenziali)*
- *RTC SRAM 16kB*
- *2 ADC 12 bit (Convertitori Analogico Digitali)*
- *2 DAC 8bit (Convertitori Digitali Analogici)*
- *3 periferiche UART (periferiche per comunicazione seriale asincrona)*
- *4 periferiche SPI (periferiche per comunicazione seriale sincrona)*
- *2 periferiche I2C (periferiche per comunicazione seriale sincrona)*
- *2 periferiche I2S (trasferimento audio digitale)*
- *10 pin per ingresso touch*
- *34 GPIO, General Purpose Input Output, pin di uso generico.*
- *8 canali **RMT** indipendenti (acronimo di **Remote Control Transceiver**, o semplicemente modulo di controllo remoto). E' una delle periferiche hardware più originali e potenti presenti all'interno dell'ESP32, in pratica è un blocco hardware dedicato alla generazione e alla ricezione di sequenze di impulsi temporizzati con estrema precisione (gestione di forme d'onda digitali). Nato storicamente per decodificare o trasmettere i segnali a infrarossi (IR) dei telecomandi (come quelli dei televisori), nel mondo dello sviluppo e dell'IoT è diventato famosissimo per un altro motivo: pilotare le strisce LED indirizzabili (es. WS2812 / NeoPixels) e gestire protocolli di comunicazione personalizzati a singolo filo (single-wire). L'RMT invece possiede una sua memoria interna (RAM dedicata ai blocchi di impulsi) e lavora in autonomia tramite DMA o interrupt hardware, lasciando la CPU completamente libera. Permette di definire la durata degli stati logici alti e bassi con una precisione di clock elevatissima, impossibile da ottenere via software in modo stabile su un normale pin GPIO. Avere 8 canali significa che si possono, ad esempio, gestire contemporaneamente 8 diverse strisce LED con animazioni differenti, oppure pilotare più periferiche basate su impulsi nello stesso momento, senza che un canale interferisca con l'altro.*
- *2 unità **MCPWM (Motor Control Pulse Width Modulator)** ovvero una periferica hardware avanzata progettata nativamente per il controllo di motori (motori DC con spazzole, motori brushless BLDC o servomotori) e sistemi di conversione di potenza. A differenza del PWM classico, l'MCPWM gestisce in autonomia funzioni complesse come le zone di sicurezza (dead-time), i segnali di sincronismo, la modulazione a portante e la gestione dei segnali di guasto (fault handling) per bloccare il motore in caso di sovracorrente. L'ESP32 ha 2 unità con 3 coppie di segnali PWM con le quali si possono pilotare due motori brushless (trifase) o 6 motori DC.*
- ***LED PWM Control (LEDC)** 16 canali indipendenti (8 per il canale ad alta velocità e 8 a bassa velocità) per regolare la luminosità dei LED o generare semplici onde quadre.*
- ***Sensore di Hall:** È un sensore magnetico integrato direttamente nel silicio del SoC ESP32 in grado di rilevare variazioni di un campo magnetico esterno vicino al chip.*

- **Ethernet.** Un controller MAC Ethernet integrato nel chip (compatibile con i chip fisici PHY esterni tramite interfaccia RMII) che permette di collegare l'ESP32 via cavo RJ45 anziché affidarsi solo al Wi-Fi.
- **TWAI** (Two-Wire Automotive Interface). È il nome con cui Espressif identifica il controller compatibile con il protocollo CAN Bus (Controller Area Network), uno standard industriale robusto utilizzato in ambito automotive e nell'automazione industriale.
- **JTAG.** È un'interfaccia hardware standard internazionale utilizzata per il debugging in-circuit e la programmazione a basso livello. Sull'ESP32, i pin JTAG permettono agli sviluppatori di collegare un debugger esterno (come un JTAG programmer) per mettere in pausa l'esecuzione del codice riga per riga, esaminare i registri della CPU e monitorare lo stato della memoria in tempo reale.
- **Camera DVP.** L'ESP32 supporta l'interfacciamento diretto con una telecamera tramite bus DVP (Digital Video Port). Il DVP è un'interfaccia parallela standard a 8 o 16 bit comunemente usata da moduli di videocamere economiche ma diffuse per l'IoT (come la famosissima OV2640 o OV7670 presenti ad esempio nelle schede ESP32-CAM), consentendo al microcontrollore di acquisire flussi di immagini o video.

Tutte queste caratteristiche sono implementate dentro a questo chip di pochi millimetri.



All'interno del Modulo **ESP-WROOM-32** insieme al SoC **ESP32-D0WD** troviamo quanto segue:

- **Memoria Flash SPI esterna.** Solitamente da **4 MB** (o 8 MB a seconda della versione), indispensabile perché il SoC nudo e crudo non ha memoria Flash interna per contenere il firmware e il programma dell'utente.
- **Cristallo Oscillatore.** Un componente a **40 MHz** che fornisce il segnale di clock di precisione necessario al funzionamento del processore, internamente al SoC tramite un PLL il Clock viene portato fino a 240MHz
- **Componenti Passivi di Supporto.** Una fitta rete di condensatori, resistenze e induttanze di disaccoppiamento e filtraggio, necessari per stabilizzare le tensioni di alimentazione del silicio.
- **Circuito di Adattamento d'Impedenza.** Un filtro RF che ottimizza il trasferimento di potenza tra il transceiver radio del SoC e l'antenna.
- **Antenna integrata.** Nel caso del modulo WROOM-32 standard, si tratta di un'antenna a **PWB (stampata)** direttamente sul circuito stampato del modulo stesso (altre varianti usano un connettore U.FL per un'antenna esterna).
- **Schermatura Metallica.** La tipica scatola d'argento saldata sul PCB, che serve a proteggere i circuiti da interferenze elettromagnetiche esterne e a evitare che il modulo disturbi altri dispositivi (fondamentale per ottenere le certificazioni di conformità CE e FCC).

Tutto questo è integrato nel modulo in figura che misura circa 5cm².



All'interno della scheda **ESP32 DEVKIT V1** oltre al modulo **ESP-WROOM-32** troviamo quanto segue:

- **Chip di conversione USB-UART (es. CP2102 o CH340).** Poiché l'ESP32 non ha una porta USB nativa per la comunicazione seriale standard, questo chip fa da "traduttore" permettendo al computer di inviare il codice e comunicare con il microcontrollore tramite un cavo USB.
- **Regolatore di tensione (es. AMS1117).** Prende i 5V erogati dalla porta USB del PC (o da un alimentatore esterno) e li converte stabilmente a **3.3V**, che è la tensione operativa richiesta dal modulo ESP32.
- **Pulsanti fisici di controllo:**
 - **EN / RST (Reset):** Per riavviare manualmente la scheda.
 - **BOOT (o IO0):** Da tenere premuto durante la fase di caricamento del codice nei casi in cui la scheda non entri automaticamente in modalità programmazione.
- **Circuito di Auto-Reset.** Un piccolo circuito a transistor che permette all'IDE di Arduino di mettere l'ESP32 in modalità boot e resettarlo in autonomia quando premi "Carica", senza farti premere i pulsanti a mano ogni volta.
- **LED di stato.** Un LED di alimentazione (Power) sempre acceso quando la scheda è collegata, e spesso un LED collegato a un pin specifico (spesso il GPIO 2) utilizzabile per test rapidi.
- **Pin Header (file di pin metallici).** Le doppie file di pin a passo 2.54mm che portano all'esterno tutti i GPIO, i pin di alimentazione (3.3V, 5V, GND) e i pin di reset, consentendo di infilare la scheda su una breadboard.

La scheda ha una dimensione di 5x3cm e si trova in commercio ad un costo al massimo di 9 euro.



PANORAMICA COMMERCIALE

Come anticipato sopra esistono diverse tipologie di SoC ESP32, e di conseguenza si trovano anche differenti moduli e differenti schede con risorse differenti.

Sempre nel sito della Espressif c'è la possibilità di vedere una panoramica dei moduli esistenti con la possibilità di effettuare delle comparazioni tra di loro.

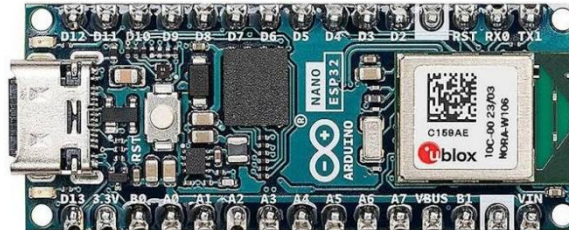
<https://products.espressif.com/#/product-comparison?names=&type=Module>

Ci sono anche schede prodotte da Espressif, ma in commercio si trovano molti cloni prodotti da terze parti come ad esempio la scheda sopra citata ESP32 DEVKIT V1, clone della ESP32-DevKitC prodotta da Espressif.



Non tutti i moduli sono prodotti da Espressif, ad esempio la scheda **Arduino nano ESP32**, monta un modulo prodotto da terze parti **u-blox NORA-W106-10B** che al suo interno ha quanto segue:

- **SoC.** Un **ESP32-S3** con architettura Dual-Core Xtensa 32-bit LX7.
- **Memoria.** 16 MB di Flash esterna e 8 MB di PSRAM aggiuntiva.
- **Connettività.** Wi-Fi a 2.4 GHz e Bluetooth 5 (con supporto BLE).
- **Antenna.** Un'antenna integrata direttamente sul modulo u-blox.



Rispetto alla precedente scheda questa ha sicuramente caratteristiche migliori, di seguito una sintetica tabella comparativa, ovviamente con costi superiori intorno ai 20 euro.

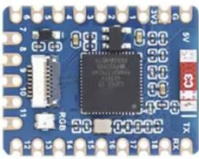
	ESP32 DEVKIT V1	ARDUINO NANO ESP32
Modulo	<i>ESP-WROOM-32</i>	<i>u-blox NORA-W106-10B</i>
SoC	<i>ESP32-D0WD</i>	<i>ESP32-S3</i>
Architettura CPU	<i>Xtensa Dual-Core a 32-bit (LX6)</i>	<i>Xtensa Dual-Core LX7 a 32-bit (con istruzioni vettoriali per AI)</i>
Frequenza di Clock	<i>Fino a 240MHz</i>	<i>Fino a 240MHz</i>
Memoria Flash (programma)	<i>4 o 8MB</i>	<i>16MB</i>
PSRAM <i>(Pseudo-Static Random Access Memory)</i> memoria RAM esterna aggiuntiva.	<i>Assente</i>	<i>8MB</i>
Connettività Wireless	<i>Wi-Fi 4 (802.11 b/g/n) + Bluetooth 4.2</i>	<i>Wi-Fi 4 (802.11 b/g/n) + Bluetooth 5 (BLE)</i>
Interfaccia USB-Seriale	<i>Chip seriale dedicato (CP2102 / CH340) con micro-USB o Type-C</i>	<i>USB-C nativa gestita direttamente dall'ESP32-S3</i>

Esistono inoltre altre schede di formato ancora più piccolo (Tiny o Supermini) come ad esempio le seguenti:

- **Seeed Studio XIAO ESP32-C5** *SoC architettura RISC-V a 32 bit con un core principale ed uno secondario a basso consumo, RAM+PSRAM e connettività avanzata, USB nativa gestita dal SoC.*

- **ESP32-C5-Zero.** *SoC architettura RISC-V a 32 bit con un core principale ed uno secondario a basso consumo, RAM e connettività avanzata, USB nativa gestita dal SoC.*

- **Seeed Studio XIAO ESP32-S3** *SoC Xtensa Dual-Core LX7, RAM+PSRAM, connettività avanzata e USB nativa gestita dal SoC.*

- **ESP32-S3-Tiny** *SoC Xtensa Dual-Core LX7, RAM+PSRAM, connettività avanzata e USB nativa gestita dal SoC.*


Sono tutte schede in formato ridotto, circa 2x2cm con prestazioni davvero eccellenti, ovviamente avendo una dimensione ridotta sono ridotti anche i relativi GPIO a disposizione.

Spesso si tende a generalizzare dicendo che la serie **ESP32-C** utilizzi un'architettura RISC-V a singolo core (spesso affiancata da un coprocessore a bassissimo consumo), mentre la serie **ESP32-S** punti a configurazioni Dual-Core. Sebbene questa sia una tendenza comune nei modelli più diffusi (come l'ESP32-C3 o l'ESP32-S3), non è una regola assoluta: ad esempio, l'**ESP32-S2** è un chip basato su architettura Xtensa ma **Single-Core**, così come esistono evoluzioni nella famiglia C. Più in generale, la lettera "**C**" identifica la linea orientata all'efficienza e ai costi contenuti basata su architettura aperta RISC-V, mentre la lettera "**S**" identifica la linea votata alle prestazioni avanzate, alla multimedialità, all'interfacciamento con display/fotocamere e alle istruzioni vettoriali per l'Intelligenza Artificiale.

Le caratteristiche riportate sopra sono estremamente sintetiche, le schede hanno diverse dotazioni. basta comunque una ricerca in rete per verificare tutte le caratteristiche.

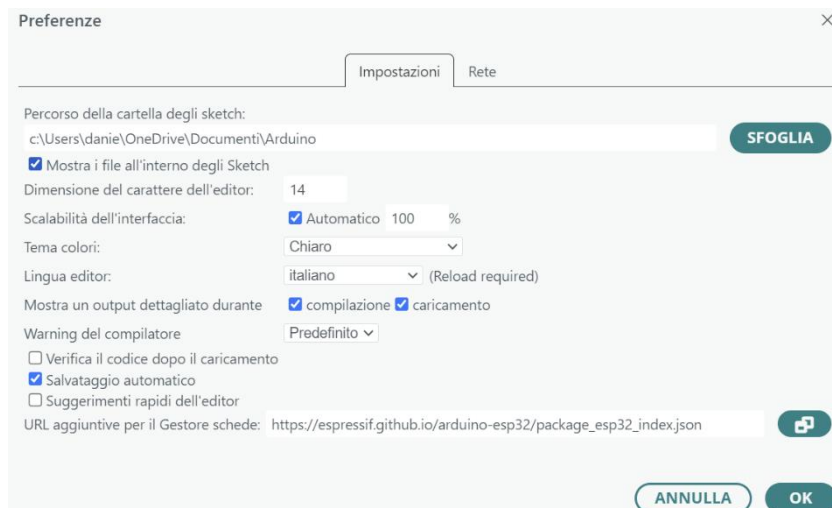
Anche in questo caso il costo di queste schede è relativamente basso tra i 10 ed i 20 euro in base al modello.

SOFTWARE E PROGRAMMAZIONE.

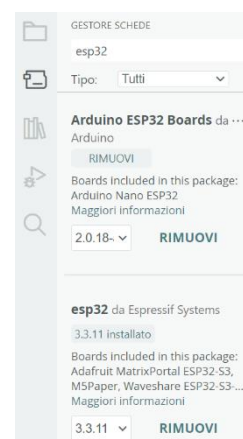
Ovviamente utilizzando schede del mondo Arduino (come la Nano ESP32), c'è il pieno supporto delle librerie ufficiali di Arduino, mentre per le altre schede occorre installare il Core Arduino per ESP32, un pacchetto che integra al suo interno l'**ESP-IDF**, ovvero il framework di sviluppo software ufficiale creato e rilasciato da Espressif Systems per programmare l'intera famiglia di chip ESP32.

Per installarlo nell'IDE di Arduino occorre aprire la sezione **File-Preferenze** ed inserire nel campo **"URL aggiuntive per il gestore di schede"** il seguente link:

https://espressif.github.io/arduino-esp32/package_esp32_index.json



Successivamente tramite il "Gestore di Schede" occorre cercare ed installare il pacchetto ESP32 sviluppato da Espressif.



Volendo invece uscire dall'IDE di Arduino, si possono utilizzare altri ambienti per programmare queste schede, ne cito alcuni di seguito:

- **MicroPython.** Serve installare un firmware sulla scheda ESP32 con l'interprete Python e si può usare Thonny IDE per scrivere programmi e caricarli sulla scheda.
- **PlatformIO.** Si installa Visual Studio Code e si installa l'estensione PlatformIO IDE.
- **JavaScript / TypeScript.** Come per Python occorre installare un firmware con l'interprete Espruino a bordo della scheda e si utilizza l'applicazione Espruino IDE.
- **Rust.** Occorre installare RUSTUP e CARGO e si utilizza poi Visual Studio Code per la scrittura ed il caricamento dei programmi.