

Nyári kalandok, mikrovezérlő nyaggyatások

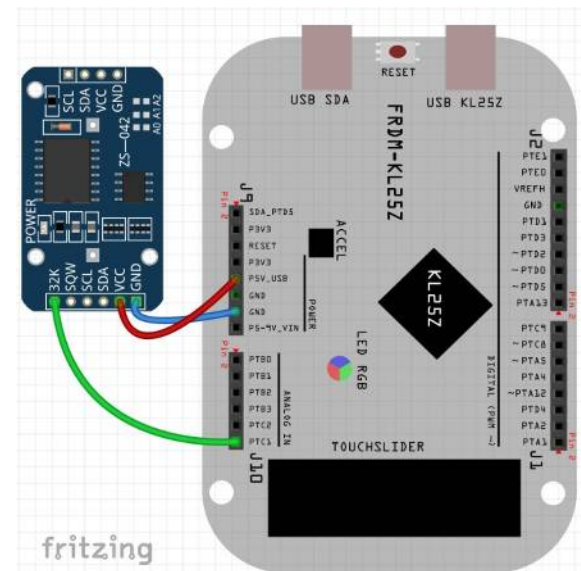


Főbb témák

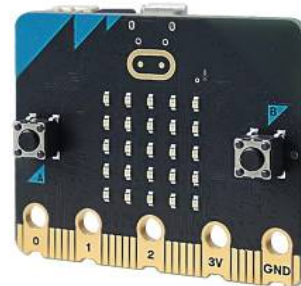
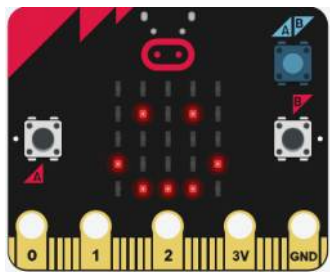
- ❖ A CLion IDE felfedezése



- ❖ A FRDM-KL25Z kártya programozása
mbed környezetben című tananyag
átdolgozása gcc4mbed-hez



- ❖ Kísérletek **micro:bit** kártyákkal: MakeCode, MicroPython,
~~gcc4mbed~~



CLion – C/C++ integrált fejlesztői környezet

- ❖ A **Clion IDE** egy komplett C/C++ fejlesztői környezet, a JetBrains terméke, 2025. májusától ingyenes licencet biztosít a nem kereskedelmi célú használatra
- ❖ **Főbb jellemzők:**
 - **CMake**-alapú projektek
 - Automatikus toolchain-felismerés: GCC, Clang, **MinGW**, **GCC cross-compile**
 - Beépített GDB/LLDB debugger: memória, változók, veremtar nézetek, töréspontok
 - Valós idejű kódelemzés: hibaelemzés, figyelmeztetések, refaktorálás
- ❖ **Bár nem erre találták ki, de jól használható:**
 - **Rust** projektekhez
 - **HTML5**, illetve **Bootstrap** projektekhez
 - **Python** projektekhez virtuális környezet használatával
 - **Vite / React / Vue / Nuxt / Node.js / React Native** projektekhez



Mit jelent a refaktorálás?

- ❖ Olyan **automatikus kódátalakítás**, amely nem változtatja meg a program működését, csak a szerkezetét teszi tisztábbá, következetesebbé, biztonságosabbá
- ❖ **Konkrét példák (C/C++)**:
 - **Rename symbol** — átnevez egy változót, függvényt, osztályt mindenhol
 - **Extract function** — egy kódrészletből új függvényt csinál, paraméterezéssel együtt
 - **Change signature** — függvény paramétereit módosítja, és minden hívást automatikusan javít
 - **Move/inline function** — áthelyez vagy beágyaz függvényeket, header–source szinten is
 - **Safe delete** — megmondja, hogy egy függvény tényleg törölhető-e, vagy nem, mert használja valami
 - **Namespace/class átszervezés** — átmozgatja a kódot úgy, hogy a build ”nem törik el”

Milyen projekteket próbáltam ki eddig?

- ❖ **MingW C++** - ez az alapértelmezett és legjobban támogatott projekttypus
- ❖ **Python 3.12** – itt első alkalommal sok kínlódással jár a projekt kialakítása a Pythonnal és a virtuális környezetettel (pláne, ha az ember nem ismeri a **Clion** lelkevilágát), de az újabb projekteknél már elég létrehozni egy üres projektet (Empty projekt) és a **Run** → **Edit Configuration**-ban hivatkozni a korábban már beállított interpreterre
- ❖ **Rust** – támogatja a párhuzamos szálak futtatását és mindenféle ellenőrzéssel biztonságosabbá tehetjük a programfejlesztést (sok hibát már a fordítás fázisában ki tud szűrni)
- ❖ **RPi Pico SDK C projektek** – a toolchain konfigurálását bonyolítja, hogy build rendszer „hosted” (Cmake, ninja), a fordító pedig a targethez igazodó keresztfordító
- ❖ **HTML5 weblapok** – itt túl sok támogatást nem ad a Clion, csak intelligens editorként használjuk

MingW C++ projekt

- ❖ Két fájl tartalmát kell felügyelnünk: **main.cpp** és **CmakeLists.txt**

main.cpp

```
#include <iostream>

int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

CmakeLists.txt

```
cmake_minimum_required(VERSION 4.2)
project(pc_test)

set(CMAKE_CXX_STANDARD 17)

add_executable(pc_test main.cpp)
```

A projekt lefordítható, illetve a Clion IDE-ben debugolható is

Független futtatáshoz a MingW/bin mappa legyen benne a PATH-ban! (.dll-ek!)

Ez az alapértelmezett tartalom, a Clion hozta létre

Python projektek

- ❖ A **Python** projektek esetében elmarad a build folyamat, így nincs szükség **CmakeLists.txt** fájlra. Python-fejlesztéshez a CLion csak fájl szintű támogatást nyújt, nem teljes projektmodell
- ❖ Létrehozunk egy üres projektet: **File** → **New** → **Empty Project**
- ❖ Létrehozzuk a program fájlt: **File** → **New Project** → **main.py**
- ❖ A projektkönyvtárba lépve hozzuk létre a Python virtuális környezetet:
`python -m venv venv`
- ❖ A virtuális környezetbe az alábbi paranccsal tölthetjük fel a szükséges kiegészítő könyvtárakat: `venv\Scripts\python.exe -m pip install <csomag>`
- ❖ **Clion**-ban a **Run** → **Edit Configuration** menüben ugyanezt a `venv\Scripts\python.exe`-t kell betallózni/megadni fordítónak
- ❖ A projekt futtatható és debugolható

A python_debug projekt

❖ Kipróbáltuk, működik: a Python projektek a Clion-ban debug-olhatók!

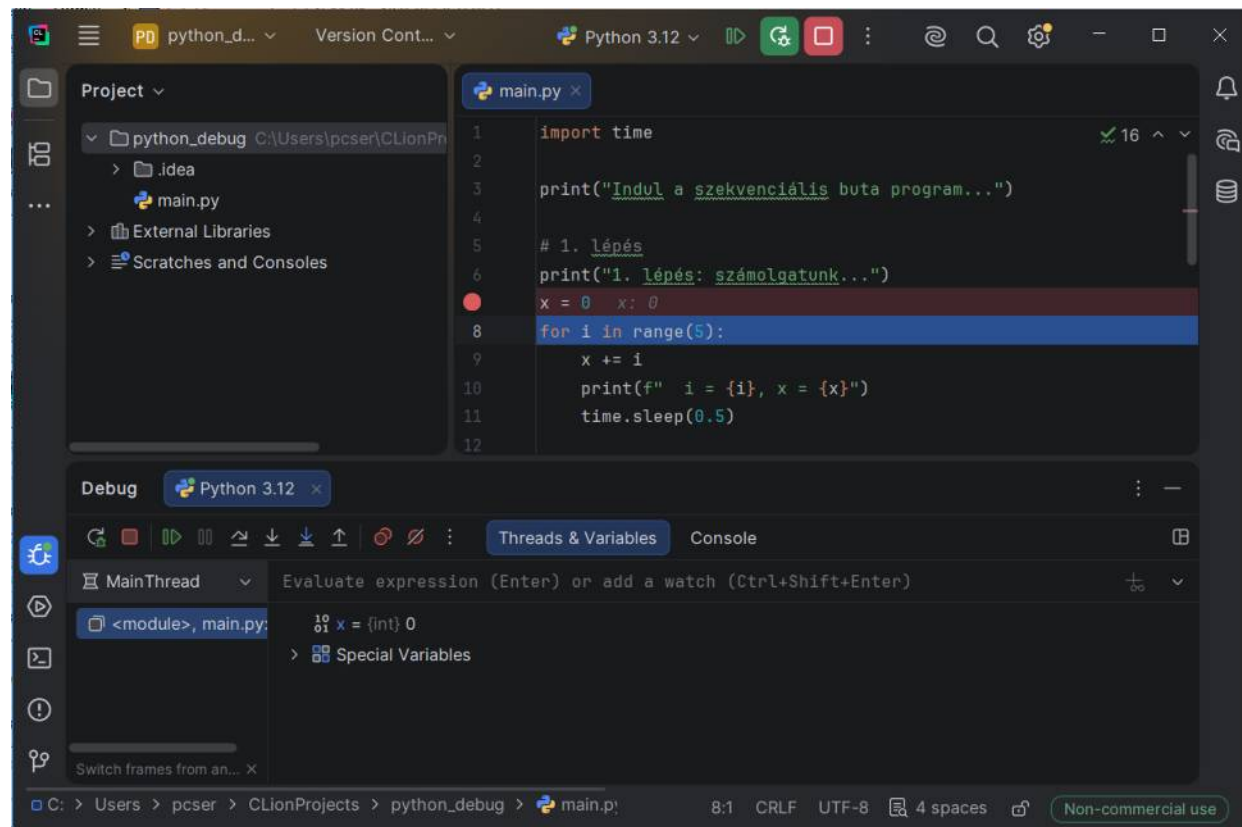
main.py

```
import time

# 1. lépés
print("1. lépés: számolgatunk...")
x = 0
for i in range(5):
    x += i
    print(f" i = {i}, x = {x}")
    time.sleep(0.5)

# 2. lépés
print("\n2. lépés: várunk egy kicsit...")
time.sleep(1)

. . . stb.
```



Rust projekt

- ❖ A CLion Rust-támogatása stabil és kezdőbarát, mert a Cargo-alapú projekteket automatikusan felismeri
- ❖ Nem kell CMake, a CLion a **Cargo.toml** alapján maga hozza létre a futtatási és debug konfigurációkat
- ❖ A szerkesztő jól kezeli a Rustot: hibajelzés, automatikus kódkiegészítés, ugrás definícióra
- ❖ A debugger működik, láthatjuk a változókat, stack frame-eket, szálakat
- ❖ A Rust projektekhez tehát a CLion teljes értékű fejlesztőkörnyezet, kezdőknek is vállalható

main.rs

```
fn main() {  
    println!("Hello, world!");  
}
```

Többszálú futtatás Rustban

```
use std::{env, thread, time::Instant};
```

pi_calc.rs

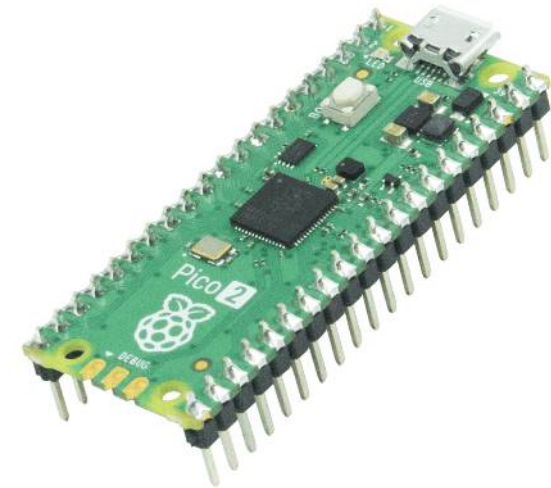
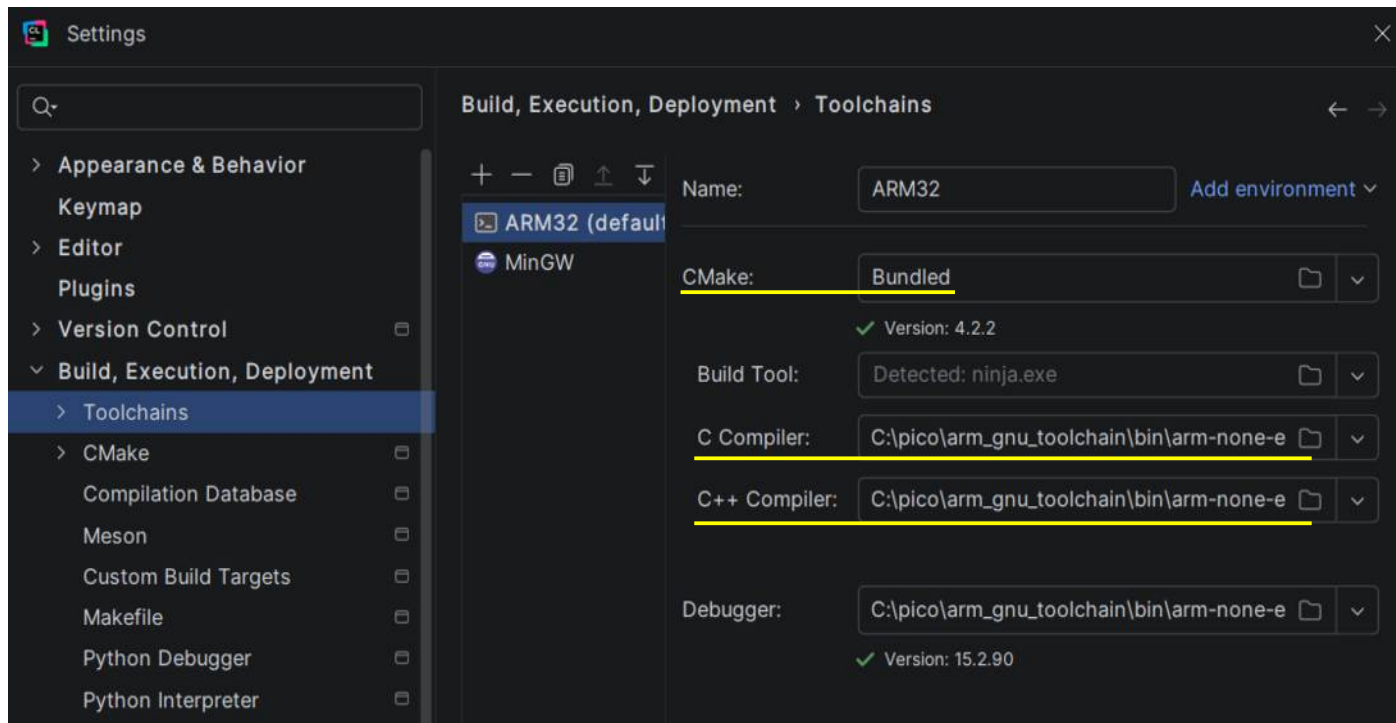
```
fn main() {
    let args: Vec<String> = env::args().collect();
    let k: u64 = args.get(1).and_then(|s| s.parse().ok()).unwrap_or(1_000_000_000);
    let n: usize = args.get(2).and_then(|s| s.parse().ok()).unwrap_or(4);
    let t0 = Instant::now();
    let chunk = (k + n as u64 - 1) / n as u64;
    let sum: f64 = thread::scope(|s| {
        let mut handles = Vec::with_capacity(n);
        for t in 0..n {
            handles.push(s.spawn(move || {
                let start = t as u64 * chunk;
                let end = ((t as u64 + 1) * chunk).min(k);
                (start..end).fold(0.0, |acc, i| {
                    acc + if i % 2 == 0 { 1.0 } else { -1.0 } / (2 * i + 1) as f64
                })
            }));
        }
        handles.into_iter().map(|h| h.join().unwrap()).sum()
    });
    println!("k = {k}, N = {n} | Pi: {:.12} | Idő: {:?}" , 4.0 * sum, t0.elapsed());
}
```

π közelítő értékének kiszámítása egy rossz hatékonyságú módszerrel (Leibnitz)
 $k = 1\,000\,000\,000$

n	idő
1	12,4 s
2	6,5 s
4	4,1 s
8	2,5 s
16	1,9 s

RP2040 projekt toolchain

- ❖ Az **RP2040** projektnél a toolchain-t úgy kell konfigurálni, hogy a **Cmake** és a **Build Tool** (ninja) a **Clion**-nal települt (Bundled) legyen, a C és C++ compiler megadása viszont az ARM GCC fordítóra mutasson



RP2040 projektek – CMake konfigurálás

- ❖ Az alábbi CMake kapcsolókkal a CLiont ARM-cross-compile módba kényszerítjük, letiltjuk a futtatható try-compile teszteket, és explicit módon megadjuk az RP2040-hez szükséges fordítókat és SDK-útvonalakat

```
-DCMAKE_SYSTEM_NAME=Generic  
-DCMAKE_TRY_COMPILE_TARGET_TYPE="STATIC_LIBRARY"  
-DCMAKE_C_COMPILER="c:/pico/arm_gnu_toolchain/bin/arm-none-eabi-gcc.exe"  
-DCMAKE_CXX_COMPILER="c:/pico/arm_gnu_toolchain/bin/arm-none-eabi-g++.exe"  
-Dpicotool_DIR="C:/pico/picotool"
```

- ❖ Az **RPi Pico C/C++ SDK**, az **ARM GCC** fordító és a **picotool** nálam együtt, a **C:\\pico** mappába van telepítve

RP2040 projekt: blink_simple

- ❖ Két fájl tartalmát kell felügyelnünk: **main.c** és **CmakeLists.txt**

main.c

```
#include "pico/stdlib.h"
#define LED_PIN 29

int main() {
    gpio_init(LED_PIN);
    gpio_set_dir(LED_PIN, GPIO_OUT);
    while (true) {
        gpio_put(LED_PIN, true);
        sleep_ms(250);
        gpio_put(LED_PIN, false);
        sleep_ms(250);
    }
}
```

CmakeLists.txt

```
cmake_minimum_required(VERSION 3.13)
include($ENV{PICO_SDK_PATH}/external/pico_sdk_import.cmake)

project(blink_simple C CXX ASM)
pico_sdk_init()

add_executable(${PROJECT_NAME}
    main.c
)

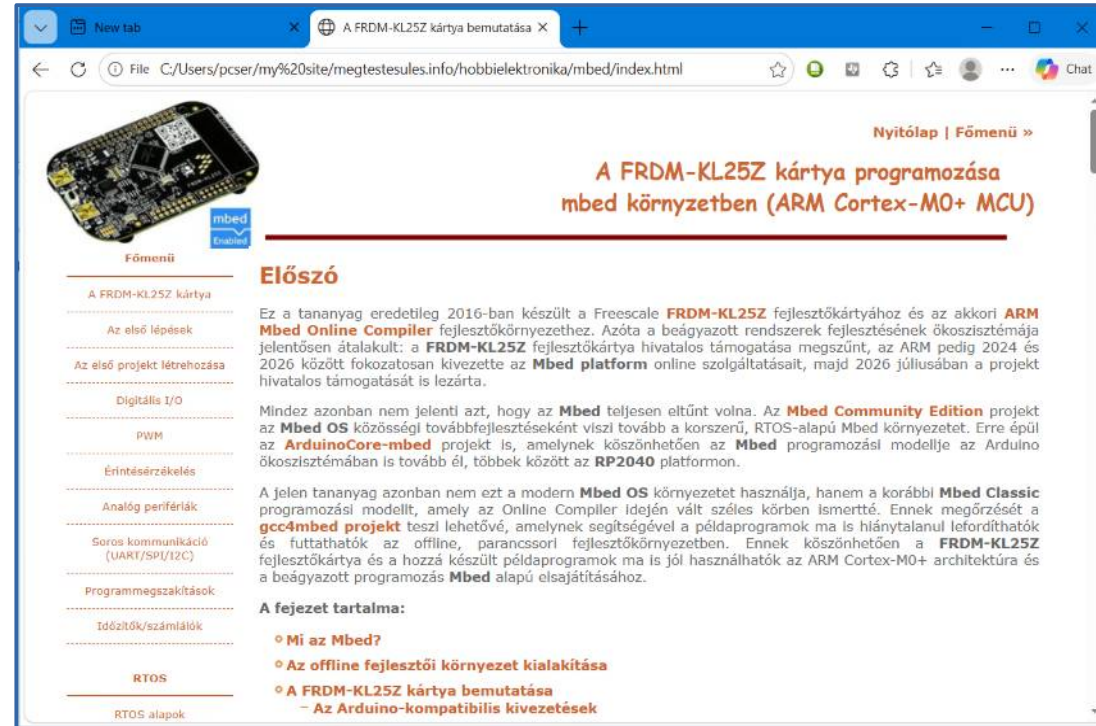
target_link_libraries(${PROJECT_NAME} pico_stdlib)

# create map/bin/hex/uf2 file etc.
pico_add_extra_outputs(${PROJECT_NAME})
```

- ❖ Mit nyerünk a Clion IDE-vel?
 - Le tudjuk fordítani a Pico SDK mintapéldáit
 - Saját projekteket készíthetünk a mintapéldák alapján
- ❖ Alternatívák: Arduino Mbed OS for RP2040, vagy Visual Studio Code az SDK-val

Az mbed webes tananyag átdolgozása

- ❖ 2026. júliusában véglegesen megszűnt az **ARM mbed** projekt támogatása és az **mbed.org** korábbi gyűjteményei (dokumentumok, programkönyvtárak) is eltűntek
- ❖ Emiatt a 2016-ban készült **mbed tananyag** (cspista.hu/mbed) elavulttá, használhatatlanná vált, teljesen át kellett dolgozni
- ❖ **Az átdolgozás főbb lépései:**
 - Munkamenet átírása **gcc4mbed**-re
 - Firmware frissítés **J-Link OB**-re
 - Korábbi projektek átnézése és kiegészítése **Makefile**-al
 - Törött linkek pótlása (archive.org, softwareheritage.org)
 - Startup kód átírása belső RTC órajel generálásához



BBC micro:bit kártyák nyaggatása

- ❖ A fő cél a 2,5 éve használt fejlesztői környezetek ellenőrzése
 - makecode.microbit.org (Microsoft MakeCode)
 - python.microbit.org (Micro:bit Educational Foundation)
- ❖ **Szöveg kiíratás** kipróbálása az USB virtuális soros porton
Tapasztalat: a kiírandók összeállításának módjától függően karaktervesztés lehetséges
- ❖ classroom.microbit.org kipróbálása
Tapasztalat: érdekes és hasznos eszköz a tanulók munkájának ellenőrzésére és segítésére
- ❖ Kísérlet a **gcc4mbed** fejlesztői környezet használatára
Tapasztalat: csak a **micro:bit v1** MCU-hoz van beépített támogatás (a v2 kártyához az Mbed OS 6/7 nyújthat támogatást) de a használat módja még nem világos (nincs elég információ)

