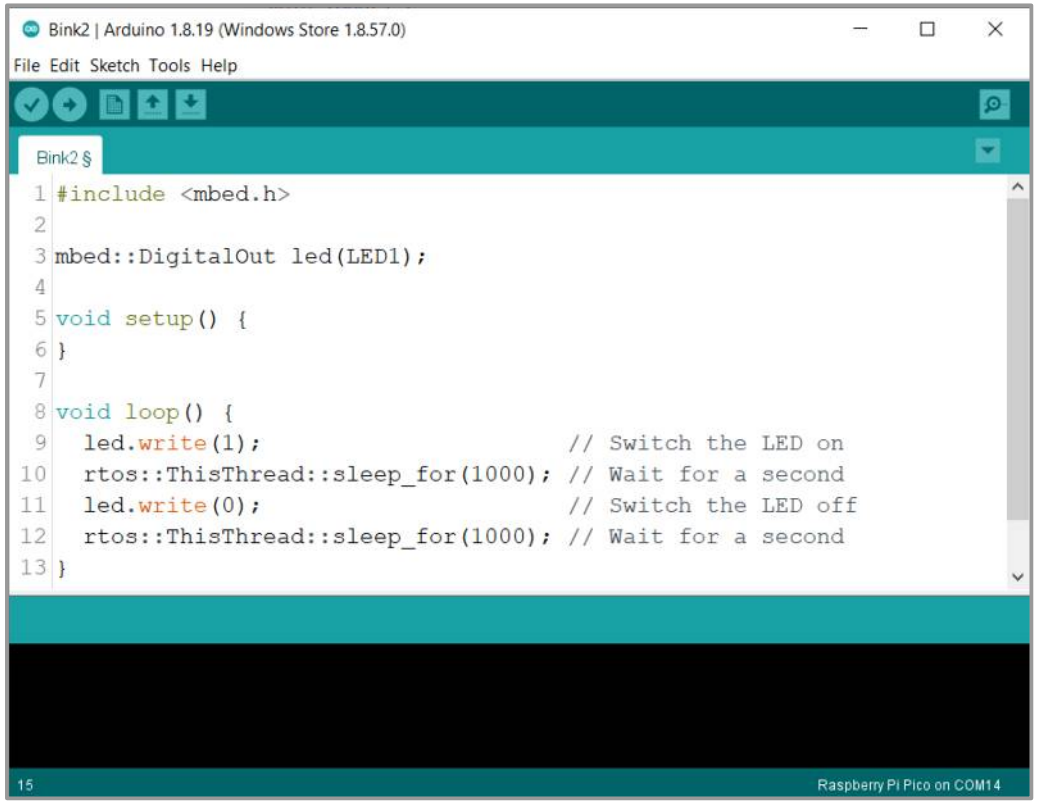
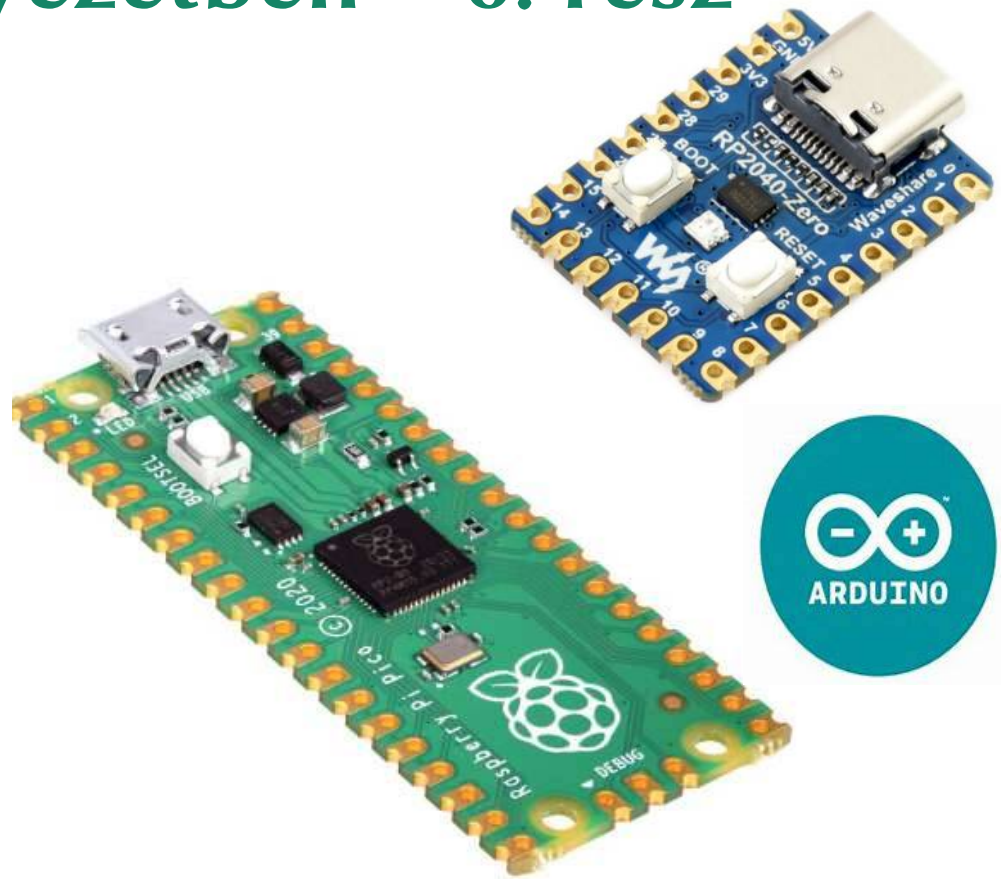
Az RP2040 mikrovezérlő programozása Arduino IDE környezetben – 6. rész





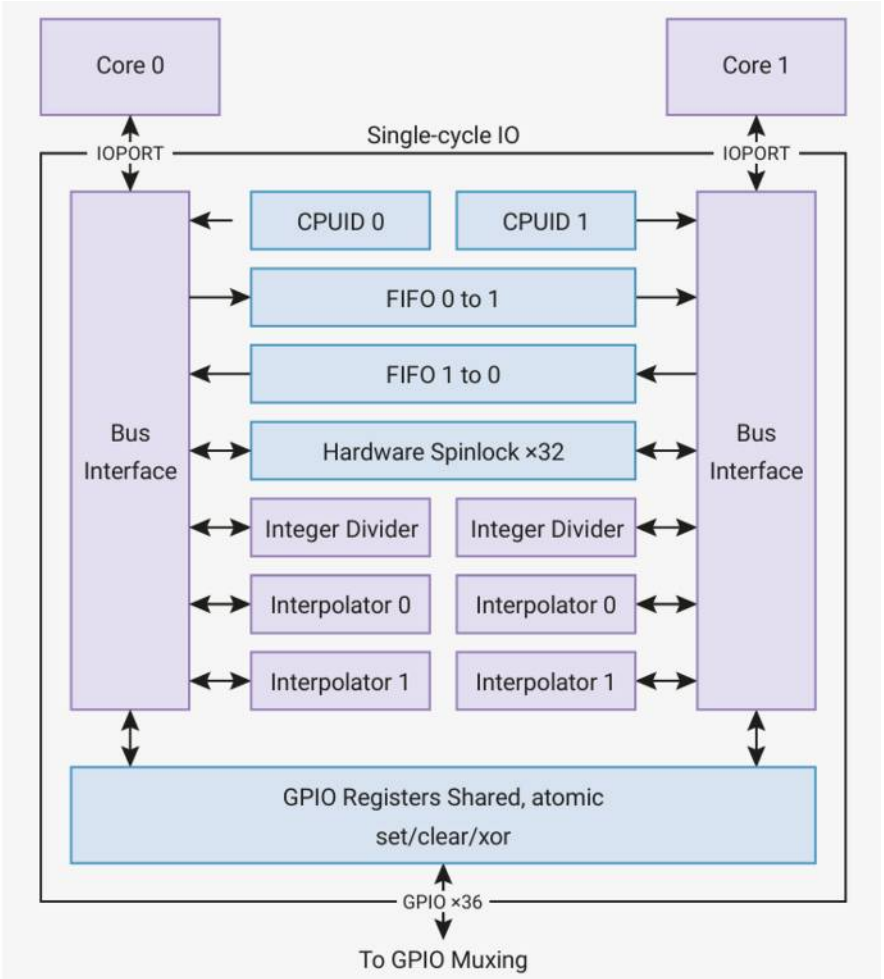
Felhasznált és ajánlott irodalom

- Raspberry Pi: Pico-series Microcontrollers
- Raspberry Pi: RP2040 adatlap (PDF)
- Raspberry Pi: Raspberry Pi Pico Datasheet
- Raspberry Pi: Raspberry Pi Pico-series C/C++ SDK
- Raspberry Pi: Getting started with Raspberry Pi Pico-series Microcontrollers
- Waveshare: RP2040-Zero, a Pico-like MCU Board

- Ralphjy: Program RPi Pico using Mbed library with Arduino IDE
- Learn Embedded Systems: Basic Multicore Pico Project
- BigG: Four Multicore C programs for Raspberry Pi Pico using Arduino IDE
- Valentin Milea: DHT sensor library for the Raspberry Pi Pico
- Alan Yorinks: NeoPixelConnect

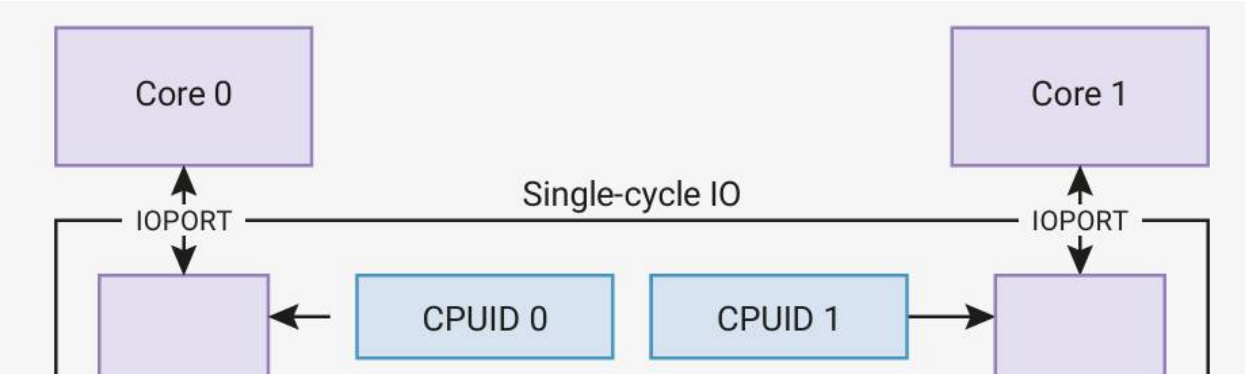
SIO – Single-cycle IO block

- A **Single-cycle IO** blokk speciális perifériái gyors és determinisztikus hozzáférést nyújtanak a processzoroknak
- A processzorok a SIO-t közvetlenül érik el, a memória-címzésű IOPORT címtartományban (0xd0 000 000 – 0xffffffff)
- Az IOPORT egy dedikált busz, amelyen keresztül az írás/olvasás egy utasításciklus, szemben az AHB-lite busszal, amelyen keresztül 2 vagy több ciklus az elérés
- A SIO nem csatlakozik a fő rendszerbuszhoz, mivel szigorú időzítési követelményekkel rendelkezik. Ezért csak a processzorok, illetve a hardveres nyomkövető érheti el a processzor debug portjain keresztül, tehát DMA-val sem lehet elérni vagy kezelni



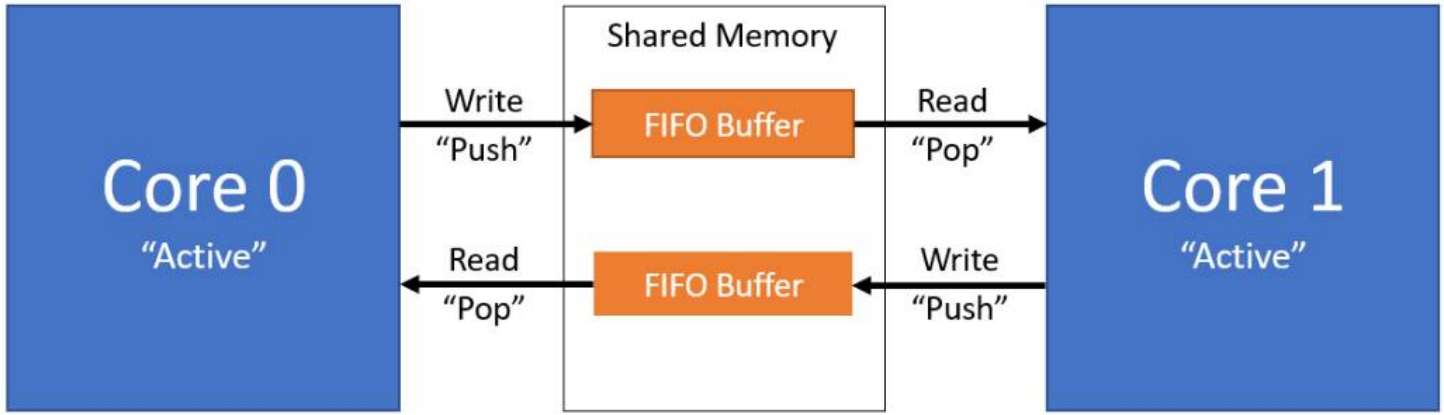
A CPUID regiszter

- A **CPUID** regiszter az első regiszter az **IOPORT** címtartományban (0xd0 000 000) **Core 0** esetén az olvasott érték **0, Core 1** esetén pedig **1.** Ez egy egyszerű módszer annak meghatározására, hogy melyik processzoron fut az adott program.
- A bootolás során mindkét mag egyszerre indul el, de **Core 1** mély alvás állapotba kerül, míg **Core 0** folytatja a rendszerindítási folyamatot. A második mag (**Core 1**) futását a Pico C/C++ SDK `multicore_launch_core1()` függvényével indíthatjuk el
- **Megjegyzés:** ne tévesszük össze ezt a regisztert a Cortex-M0+ belső **CPUID** regiszterével, amely a processzor típusát és verzióját tartalmazza



Kétmagos programok szinkronizálása

- A két processzormagot mindkét irányban egy-egy 8x32 bites **FIFO** köti össze, ami adatcserére és szinkronizálásra használható, a **Pico C/C++ SDK** függvényeinek segítségével (a szinkronizálást a blokkoló várakozások biztosítják)
- Az adatküldő/fogadó Pico C/C++ SDK függvényhívások blokkoló típusúak (várakozás)
- A **FIFO** állapot lekérdezése (hogy van-e beérkezett adat, vagy szabad hely a FIFO tárban) pedig nemblokkoló függvényekkel történik



Az RP2040 C/C++ SDK multicore függvényei

- **multicore_reset_core1**(void): Alaphelyzetbe állítja a második magot.
- **multicore_launch_core1**(void (*entry)(void)): Elindítja a második magot egy megadott belépési ponttal
- **multicore_fifo_rvalid**(void): Ellenőrzi, hogy van-e érvényes adat a FIFO-ban
- **multicore_fifo_wready**(void): Ellenőrzi, hogy a FIFO készen áll-e az írásra.
- **multicore_fifo_push_blocking**(uint32_t data): Adatot küld a másik magba *(blokkol)*
- **multicore_fifo_pop_blocking**(void): Adatot fogad a másik magból *(blokkol)*

Lockout kezelése (a másik mag futásának felfüggesztése):

- **multicore_lockout_victim_init**(void): Előkészíti a lockout folyamatot
- **multicore_lockout_start_blocking**(void): Blokkoló módon kezdi a lockout-ot
- **multicore_lockout_start_timeout_us**(uint64_t timeout_us): Lockout időtúllépéssel
- **multicore_lockout_end_blocking**(void): Blokkoló módon befejezi a lockout-ot.
- **multicore_lockout_end_timeout_us**(uint64_t timeout_us): Lockout befejezése időtúllépéssel

Egyszerű példa kétmagos programfutásra

- A 2024. február 27-i előadásban bemutattuk, hogy az **RP2040** mikrovezérlőn a többmagos futtatás jelentősen csökkentheti a számítási időt a CPU intenzív feladatoknál. Abban az előadásban a Leibnitz-sorozat segítségével számítottuk ki a **π** közelítő értékét:

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots = \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1},$$

- Az alternáló Leibnitz sorozat lassan konvergál, így nagyszámú (n = 1 000 000) iterációt végezve, jól mérhető a végrehajtási idő
- Az akkori **dual_core.ino** programot most úgy módosítjuk, hogy:
  - Ugyanazt a **core_task** függvényt indítjuk el mindkét CPU-n, s a **CPUID** kiolvasásával döntjük el, hogy melyik processzormagon fut a program
  - A **Core1** magot a **FIFO** segítségével szinkronizáljuk a **Core0** maghoz

Ilyen módon a kód egyszerűbb és áttekinthetőbb lett

picalc_dual_core.ino – 2/1.

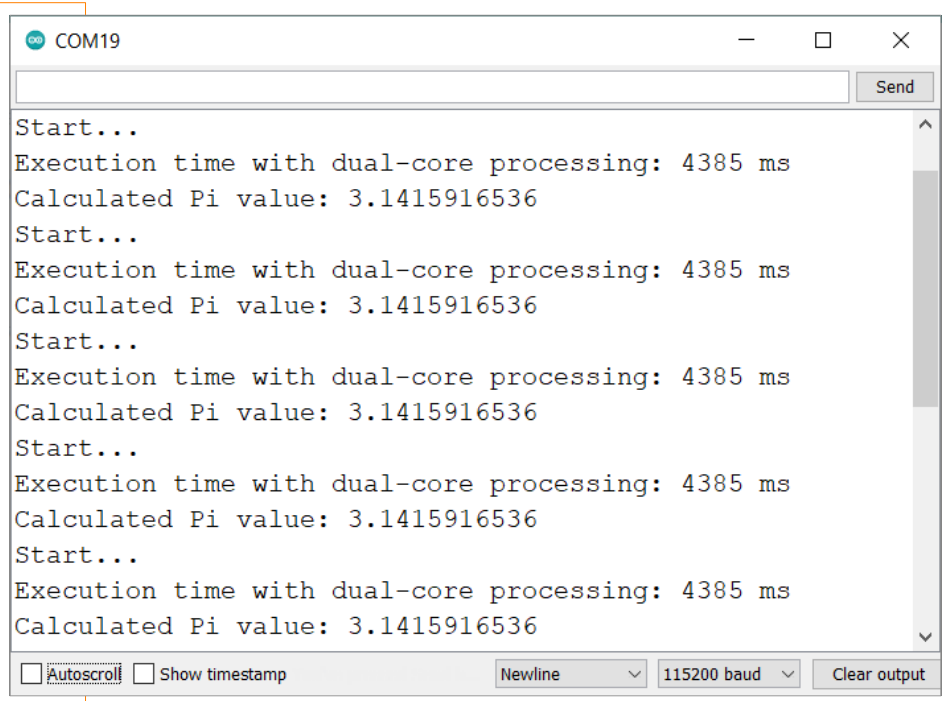
```
#include "pico/multicore.h"
double result0 = 0.0;
double result1 = 0.0;
uint32_t n = 1000000; // Number of iterations

double pi_calc(uint32_t start, uint32_t end) {
  double pi = 0.0;
  for (uint32_t i = start; i < end; ++i) {
    pi += (i % 2 == 0) ? 1.0 / (2 * i + 1) : -1.0 / (2 * i + 1);
  }
  return pi * 4; // Multiply by 4 to get the final result
}

void core_task() {
  uint32_t core_id = *(volatile uint32_t *)(0xD0000000); // Read CPUID register
  if (core_id == 0) {
    result0 = pi_calc(n / 2, n);          // Core 0: Runs once, no infinite loop
  }
  if (core_id == 1) {
    while (true) { // Core 1: Runs in an infinite loop
      uint32_t received_n = multicore_fifo_pop_blocking();
      result1 = pi_calc(0, received_n / 2);
      multicore_fifo_push_blocking(received_n); // Send back some data via FIFO
    }
  }
}
```

Pi közelítő számolása a Leibnitz sorozattal

A CPU-kon futó feledat (Core1 feladata végtelen ciklusban fut!)

picalc_dual_core.ino – 2/2.

```ino
void setup() {
  Serial.begin(115200);
  delay(1000);
  // Elindítjuk a második magon futó feladatot
  multicore_launch_core1(core_task);
}

void loop() {
  Serial.println("Start...");
  uint32_t start_time = millis();
  // Send the iteration count to Core 1 via FIFO
  multicore_fifo_push_blocking(n);
  // Core 0 performs the calculation
  core_task();
  // Wait for Core 1
  uint32_t dummy = multicore_fifo_pop_blocking();
  double final_result = result1 + result0; // Sum
  // Measure execution time
  uint32_t end_time = millis();
  uint32_t execution_time = end_time - start_time;
  Serial.print("Execution time with dual-core processing: ");
  Serial.print(execution_time);   Serial.println(" ms");
  Serial.print("Calculated Pi value: ");
  Serial.println(final_result, 10);
  delay(2000);
}
```



Közvetlen GPIO kezelés

- Az **RP2040** processzorok a **SIO** modulon keresztül közvetlen hozzáférést kapnak a **GPIO** regiszterekhez, lehetővé téve a gyors bit-szintű manipulációt, amellett, hogy az **AHB-Lite** buszon keresztül is vezérelhetők, rugalmasabb hozzáférést biztosítva
- **Regiszterek és funkcióik:**
  - **GPIO_OUT** – Kimeneti szint beállítása (1: magas, 0: alacsony)
  - **GPIO_OE** – Kimenet engedélyezése (0: nagy impedancia, 1: aktív meghajtás)
  - **GPIO_IN** – GPIO állapotok kiolvasása, egyszerre akár mind a 30 lábról
- A **GPIO_OUT** és és **GPIO_OE** regiszterek atomi **SET, CLR** és **XOR** műveletekkel kezelhetők, lehetővé téve a **GPIO** lábak frissítését egyetlen művelettel. Ez nemcsak a párhuzamos kétmagos hozzáférést tesz biztonságossá, hanem megszakításkezelés és előtérben futó kód esetén is garantálja a stabil GPIO-működést
- A **read-modify-write** műveleteknél a regiszter először beolvasásra kerül, majd módosítás után visszaírásra. Ha közben egy másik folyamat is módosítja ugyanazt a regisztert, annak változásai elveszhetnek, mert a visszaírás felülírja a köztes módosítást.
- Az **atomi műveletek** egyetlen lépésben oldják a regiszter módosítását, biztosítva a biztonságos és versenyhelyzet-mentes frissítést

Hardware spinlock rendszer

- Az **RP2040** mikrovezérlő 32 hardveres **spinlockot** biztosít, amelyekkel a programok kizárólagos hozzáférést biztosíthatnak megosztott erőforrásokhoz
- **Spinlock felépítése:** Minden **spinlock** egy egybitnyi állapotjelző, amely egyedi címen található (SPINLOCK0–SPINLOCK31)
- **Használati műveletek:**
  - **Olvasás:** Próbálkozás a lock megszerzésére. Ha a visszakapott érték nem nulla, a lock sikeresen lefoglalásra került, ha nulla, akkor már más folyamat használja
  - **Írás (bármilyen érték):** A lock felszabadítása. Ezután a következő próbálkozás sikeres lesz.
  - **Versenyhelyzet kezelése:** Ha mindkét mag egyszerre próbálja megszerezni ugyanazt a lockot, akkor **Core 0** élvez elsőbbséget
- **Hatékonyság:** A spinlockot úgy szerzi meg a program, hogy folyamatosan figyeli az állapotjelzőt („pörög” a lockon), amíg az elérhetővé nem válik. Hosszú ideig tartó zárolás emiatt nem hatékony, ezért a spinlockokat általában rövid, kritikus szakaszok védelmére használják
- Ez a **spinlock-rendszer** segíti a többmagos programok szinkronizációját az **RP2040**-en, lehetővé téve a megosztott erőforrások biztonságos elérését

Spinlockok kiosztása az RP2040 Pico SDK-ban

- **Spinlock 0–13** – Az SDK és egyéb könyvtárak kizárólagos használatára fenntartott. Minden egyes lefoglalt spinlock saját PICO_SPINLOCK_ID azonosítóval rendelkezik
- **Spinlock 14–15** – Az operációs rendszerek vagy rendszer szintű szoftverek számára vannak fenntartva (PICO_SPINLOCK_ID_OS1 és PICO_SPINLOCK_ID_OS2).
- **Spinlock 16–23** – Megosztott használatú **striped spinlockok,** amelyeket körkörös kiosztással oszt ki a **next_striped_spin_lock_num() függvény.** A szétosztás csökkenti annak esélyét, hogy magasabb szintű zárolási mechanizmusok ugyanazt használják
- **Spinlock 24–31** – Felhasználói programokban exkluzív használatú **spinlockok,** amelyek a **spin_lock_claim_unused()** függvényhívással *első igénylő* alapon kerülnek kiosztásra
- Bővebb információ és az SDK függvények dokumentációja itt található: www.raspberrypi.com/documentation/pico-sdk/hardware.html#group_hardware_sync
- **Mutex és spinlock összehasonlítása:**
  - **Mutex:** Ha egy folyamat lefoglalja, a többiek blokkolásra kerülnek és várakoznak
  - **Spinlock:** a kizárásra került folyamat folytonosan próbálkozik, ami rövid várakozásnál hatékony megoldás, de hosszabb várakozás esetén gazdaságtalan

Pédaprogram: spinlock és SIO→GPIO használata

- A következő program is kétmagos működést valósít meg:
  - **Core 0** (az alapértelmezett mag) egy villogó LED-et vezérel 500 ms-os intervallummal.
  - **Core 1** pulzáló fényhatást kelt a másik LED-en, a fényerőt fokozatosan növelve és csökkentve
- Mivel a két LED-et ugyanaz a **GPIO** port vezérli, a két mag között versenyhelyzet alakul ki a közös erőforrás eléréséért, amelyet a **SIO SPINLOCK** segítségével kontrollálunk. A **spinlock** használata biztosítja, hogy a LED-ek állapotának módosítása szinkronizált legyen, elkerülve a versenyhelyzetből adódó nem kívánt mellékhatásokat
- A SIO GPIO elérést használjuk, a GPIO portot ennek megfelelően kell inicializálni!
- Részletezve a **spinlock** működését:
  - A **spinlock_claim**(lock_id) és **spin_lock_init**(lock_id) inicializálja a spinlockot
  - A **spin_lock_blocking**(lock) függvény segítségével a program blokkolva vár a zárolásra, így elkerülhető a két mag egyidejű írása az azonos erőforrásra
  - **A spin_unlock**(lock, save) feloldja a zárolást, így másik mag is hozzáférhet
- A ***spinlock_id*** esetünkben 24 lesz (ez az első szabadon használható ID), ***lock*** pedig egy **spin_lock_t*** típusú spinlock objektum

multicore_spinlock.ino – 4/1.

```
#include <Arduino.h>
#include "pico/multicore.h"
#include "hardware/sync.h"
#include "hardware/gpio.h"

#define CORE0_LED 8u   // LED connected to Core 0
#define CORE1_LED 5u   // LED connected to Core 1

int brightness = 0;     // LED brightness level
int fadevalue = 1;      // Increment/decrement value for brightness transition
uint spinlock_id = 24;  // Spinlock identifier
spin_lock_t* lock;      // Spinlock object
uint32_t save = 0;      // Spinlock state saving variable

// Macros for atomic GPIO manipulation using SIO registers
// GPIO_SET(pin) atomically sets the specified pin high
// GPIO_CLR(pin) atomically clears the specified pin (sets it low)
#define GPIO_SET(pin) (sio_hw->gpio_set = (1u << pin))
#define GPIO_CLR(pin) (sio_hw->gpio_clr = (1u << pin))
```

multicore_spinlock.ino – 4/4.

```
//--- Core 1 execution loop -------------------
// Controls the brightness of the CORE1_LED by gradually increasing and decreasing it.
// Uses spinlocks to ensure exclusive access to the GPIO registers
void core1_loop() {
  While (1) {
    brightness += fadevalue;
    if (brightness <= 0 || brightness >= 256) {
      fadevalue = -fadevalue;  // Reverse direction when reaching bounds
    }

    save = spin_lock_blocking(lock);
    GPIO_SET(CORE1_LED);
    spin_unlock(lock, save);
    sleep_us(brightness);      // Adjust delay based on brightness

    save = spin_lock_blocking(lock);
    GPIO_CLR(CORE1_LED);
    spin_unlock(lock, save);
    sleep_us(2000);            // Fixed delay
  }
}
```

multicore_spinlock.ino – 4/3.

```
/**
 * Setup function
 * Initializes GPIO pins, sets up the spinlock, and launches Core 1 execution.
 */
void setup() {
  // Initialize GPIO pins and configure them for SIO control
  // The _gpio_init() function enables I/O and sets the GPIO multiplexer
  // to select the SIO GPIO function, allowing direct manipulation via SIO registers.
  _gpio_init(CORE1_LED);
  gpio_set_dir(CORE1_LED, GPIO_OUT);  // Set CORE1_LED as an output pin
  _gpio_init(CORE0_LED);
  gpio_set_dir(CORE0_LED, GPIO_OUT);  // Set CORE0_LED as an output pin

  // Initialize spinlock for safe multi-core access
  spin_lock_claim(spinlock_id);
  lock = spin_lock_init(spinlock_id);

  // Launch Core 1 after a delay to prevent initialization issues
  delay(1000);
  multicore_launch_core1(core1_loop);
}
```

multicore_spinlock.ino – 4/4.

```
/**
 * Core 0 main loop
 * Blinks the CORE0_LED at a fixed interval using spinlocks for safe access.
 */

void loop() {
  save = spin_lock_blocking(lock);  // Acquire spinlock for exclusive access to GPIO
  GPIO_SET(CORE0_LED);              // Atomically set CORE0_LED high
  spin_unlock(lock, save);          // Release spinlock after GPIO modification
  delay(500);                       // Keep LED on for 500 milliseconds

  save = spin_lock_blocking(lock);  // Acquire spinlock before modifying GPIO again
  GPIO_CLR(CORE0_LED);              // Atomically clear CORE0_LED (set it low)
  spin_unlock(lock, save);          // Release spinlock after operation
  delay(500);                       // Keep LED off for 500 milliseconds
}
```

Megjegyzés: esetünkben az állapotmentés és visszaállítás valójában nem lett volna szükséges, mert nem olvasunk vagy módosítunk megosztott adatot a **spinlock** alatt, csak egyetlen **GPIO** műveletet végzünk

Hardveres osztóáramkör

- Az **RP2040** SIO modul processzoronként egy-egy 8 ciklusos osztó/modulo egységet is tartalmaz, ami lehetővé teszi, hogy azok előjeles, vagy előjel nélküli egészosztási műveleteket végezzenek párhuzamosan, miközben más műveleteket is futtatnak
- Az eredmények a 9. ciklusban olvashatók ki, és egy ***'ready'*** bit segítségével lekérdezhető a számítás állapota
- Az új számítás azonnal elindul az operandus regiszter írásakor, és egy új írás felülírja a folyamatban lévő számítást
- Ha több számot kell ugyanazzal az osztóval osztani, elegendő csak egyszer **xDIVISOR**-t beírni, az előjelességet **SDIVIDEND** vagy **UDIVIDEND** használata határozza meg
- Az **AEABI** (ARM Embedded Application Binary Interface) támogatás a C szintű / és % műveleteket közvetlenül a hardveres osztóra képezi le, így a fordító nem használ szoftveres osztási rutinokat
- A következő oldalon bemutatott egyszerű példában mi **közvetlen regiszterelérést** használunk, hogy a közös osztóval végzett műveletek egyszerűségét szemléltethessük

sio_hw_divider.ino

```
#include <Arduino.h>
#include "RedirectedSerial.h"
#include "hardware/regs/sio.h"
// Előre definiált regiszternevek előjeles osztáshoz
#define SIO_DIVIDEND_REG  (*(volatile int32_t *)(SIO_BASE + SIO_DIV_SDIVIDEND_OFFSET))
#define SIO_DIVISOR_REG   (*(volatile int32_t *)(SIO_BASE + SIO_DIV_SDIVISOR_OFFSET))
#define SIO_QUOTIENT_REG  (*(volatile int32_t *)(SIO_BASE + SIO_DIV_QUOTIENT_OFFSET))
#define SIO_CSR_REG       (*(volatile int32_t *)(SIO_BASE + SIO_DIV_CSR_OFFSET))

void setup() {
    Serial.begin(115200);
    printf("Hardware Divider közvetlen regiszterkezeléssel (előjeles osztás)\n");
    int32_t dividends[ARRAY_SIZE] = {987654, 456789, -654321, 123456, -999999};
    int32_t divisor = 123;                  // Közös osztó
    int32_t quotients[5];
    SIO_DIVISOR_REG = divisor;              // Beállítjuk az osztót egyszer!
    for (int i = 0; i < 5; i++) {           // Tömb feldolgozása egy ciklusban
        SIO_DIVIDEND_REG = dividends[i];    // Csak az osztandót írjuk be
        while (!(SIO_CSR_REG & 1)) {}       // Várjuk az eredményt
        quotients[i] = SIO_QUOTIENT_REG;    // Kiolvassuk az eredményt
        Printf("% d / %d = % d\n", dividends[i], divisor, quotients[i]);
    }
}

void loop() { }
```

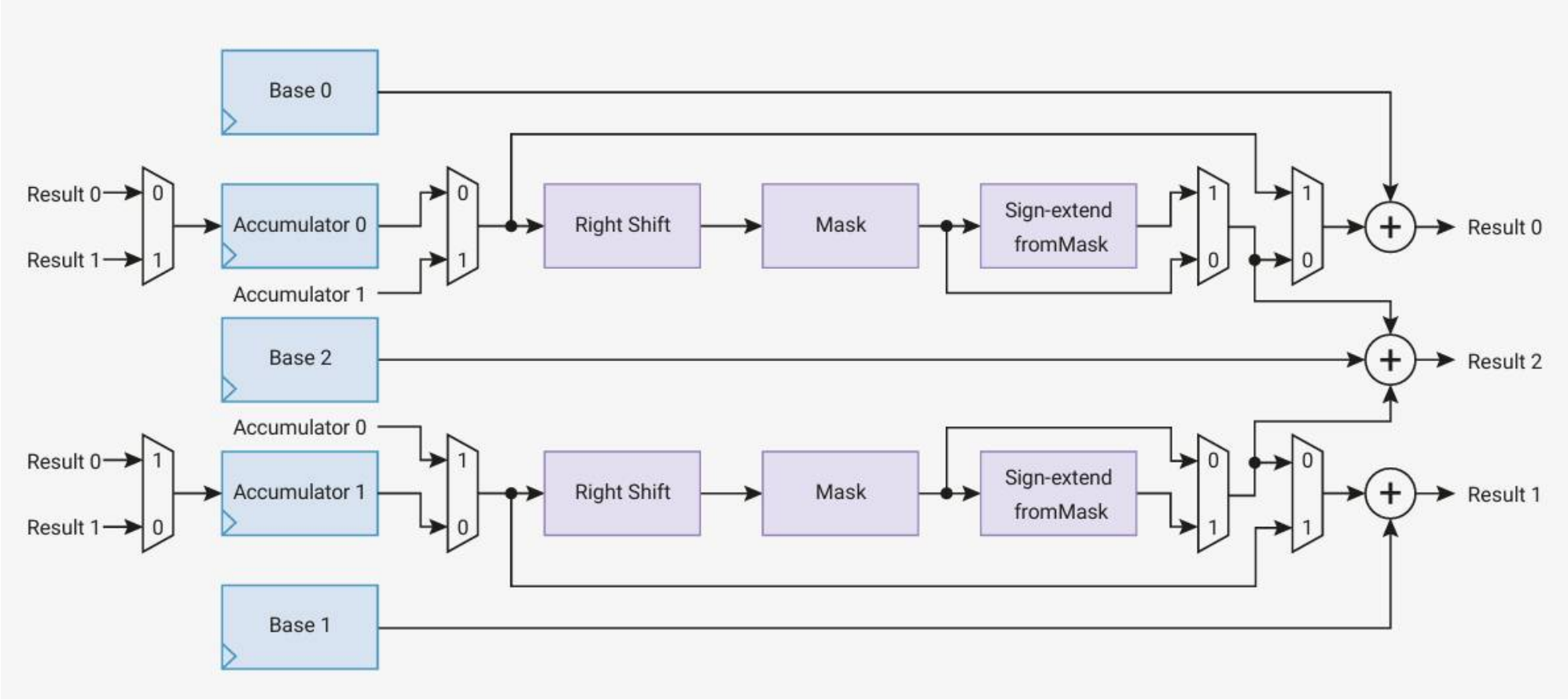
Az eredmény:

```
 987654 / 123 =  8029
 456789 / 123 =  3713
-654321 / 123 = -5319
 123456 / 123 =  1003
-999999 / 123 = -8130
```

SIO interpolátorok

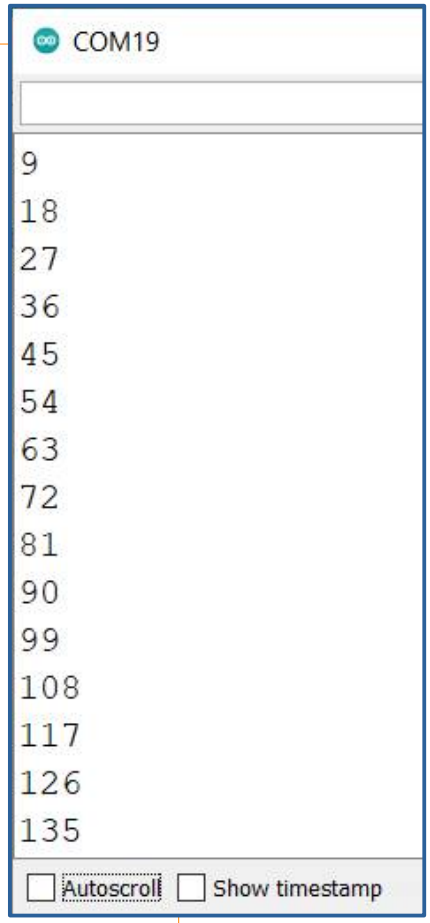
- Az **RP2040** mindkét processzora két interpolátorral (INTERP0 és INTERP1) rendelkezik, amelyek előre konfigurált műveleteket egyetlen CPU ciklusban hajtanak végre
- Előnyök és alkalmazások:
- Gyorsítják az ismétlődő számításokat, csökkentve a CPU terhelését.
- Kevesebb CPU regisztert igényelnek, így optimalizálják a kód kritikus szakaszait.
- Főként audio feldolgozásra használják az SDK-ban, de alkalmasak pl. kvantálásra, ditheringre, cím-generálásra, textúra leképezésre, tömörítésre és visszacsatolásra.
- Ezek rugalmasan konfigurálhatók, így sokféle optimalizálást lehetővé tesznek:
  - Alapérték és léptetés beállítása - Az interpolált értékek növekedési sebessége változtatható.
  - Akkumulátor használata - Az előző értékek dinamikusan befolyásolják a következő számításokat
  - Kimeneti módok - Az eredmény lehet közvetlen, módosított vagy kombinált érték.
  - Hardveres címképezés - Gyors memóriahozzáférést tesz lehetővé, például táblázatkezelésnél

Az interpolátorok felépítése

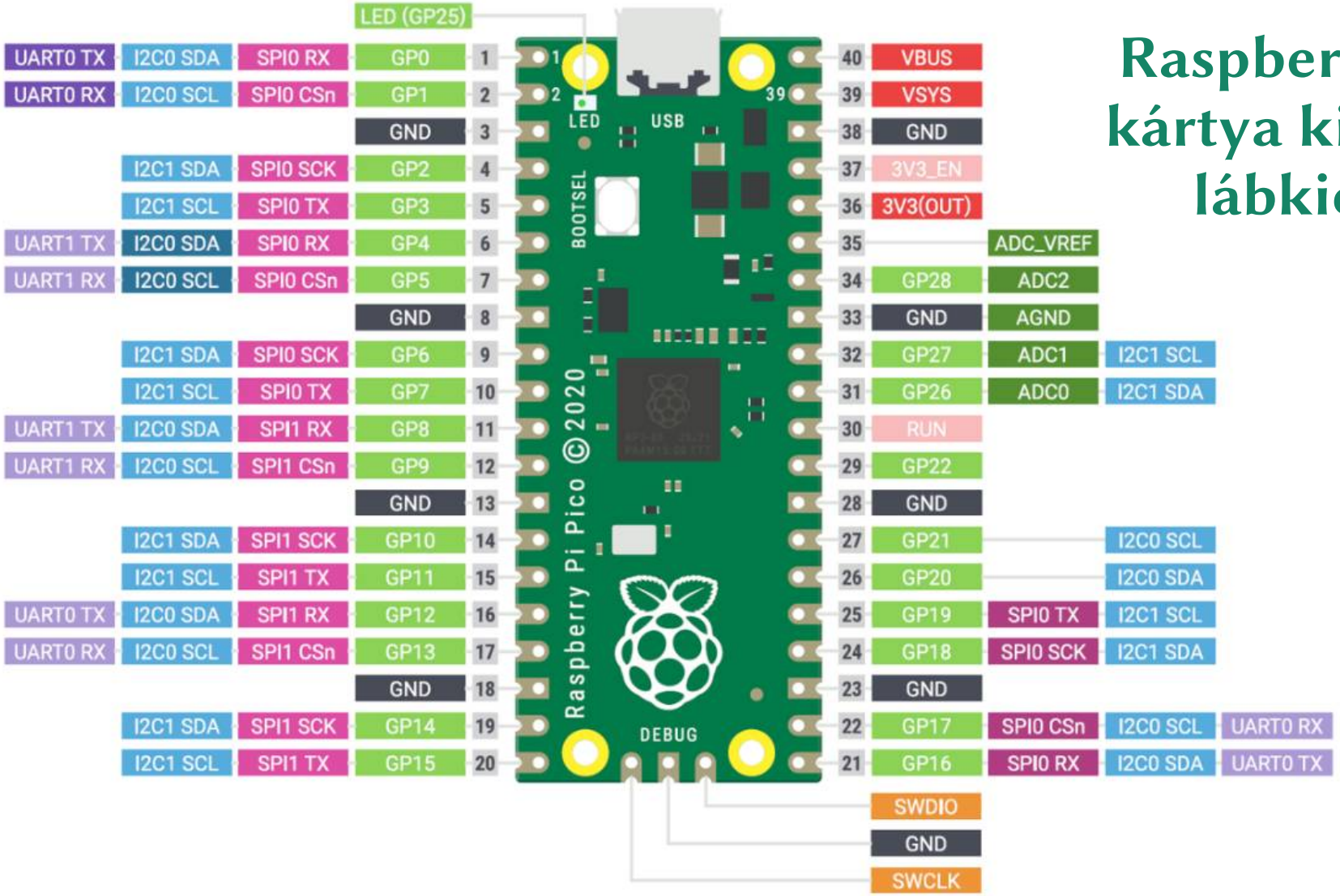


Egyszerű példa: 9-es szorzótábla

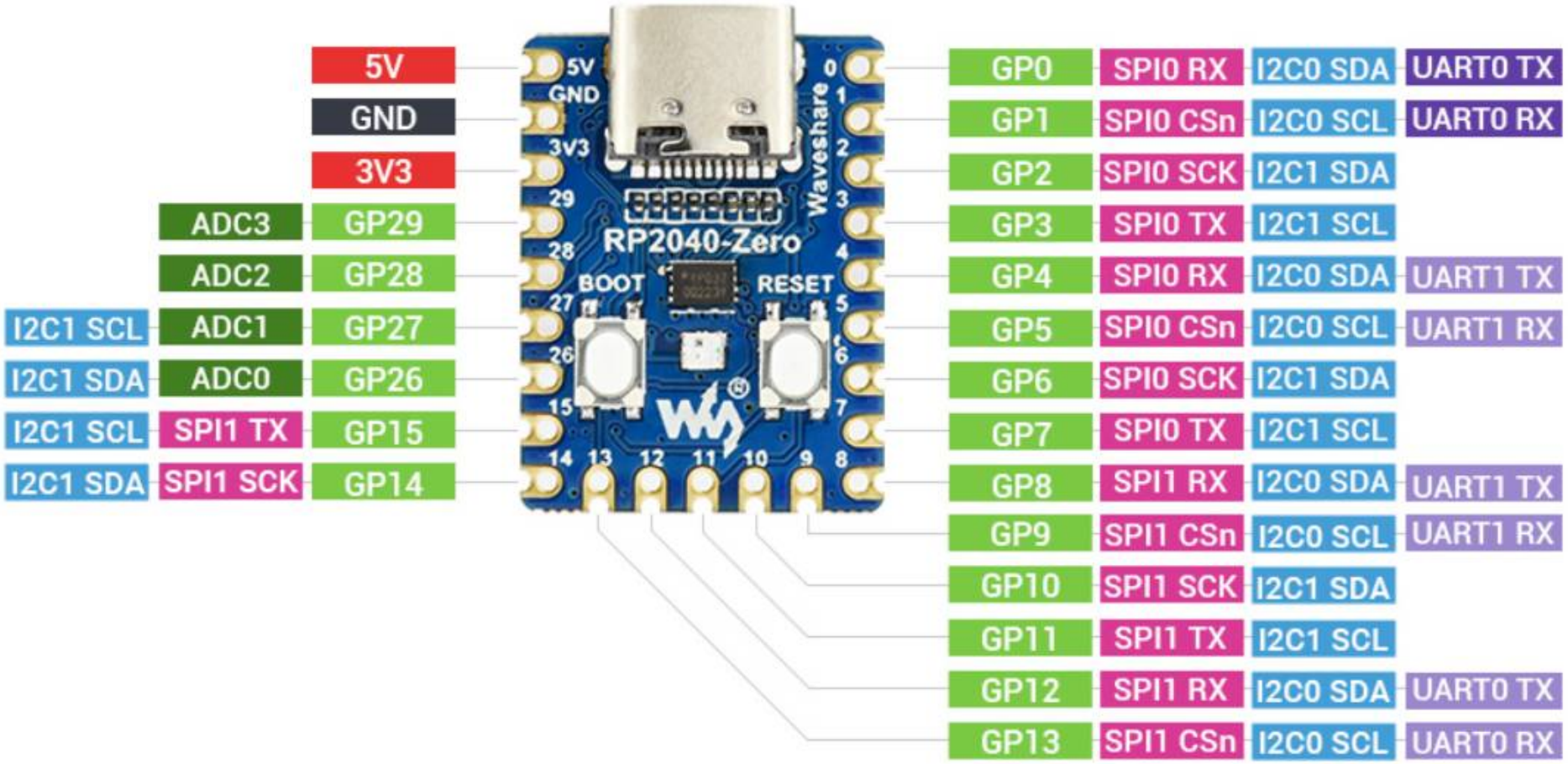
```
#include "hardware/interp.h"  // RP2040 SDK for interpolator access

void setup() {
    Serial.begin(115200);  // Initialize serial communication

    // Configure interpolator lane 0
    interp_config cfg = interp_default_config();
    interp_set_config(interp0, 0, &cfg);

    // Set initial values for interpolation
    interp0->accum[0] = 0;    // Start value
    interp0->base[0] = 9;     // Multiplication factor
}

void loop() {
    // Read interpolated value and print it
    Serial.println(interp0->pop[0]);
    delay(500);  // Wait half a second before next output
}
```



Raspberry Pi Pico kártya kivezetések lábkiosztása



Az RP2040-Zero kártya kivezetéseinek lábkiosztása



További kivezetések

- Néhány kivezetés az RP2040-Zero kártya hátoldalán van kivezetve