Az RP2040 mikrovezérlő programozása Arduino IDE környezetben – 2. rész

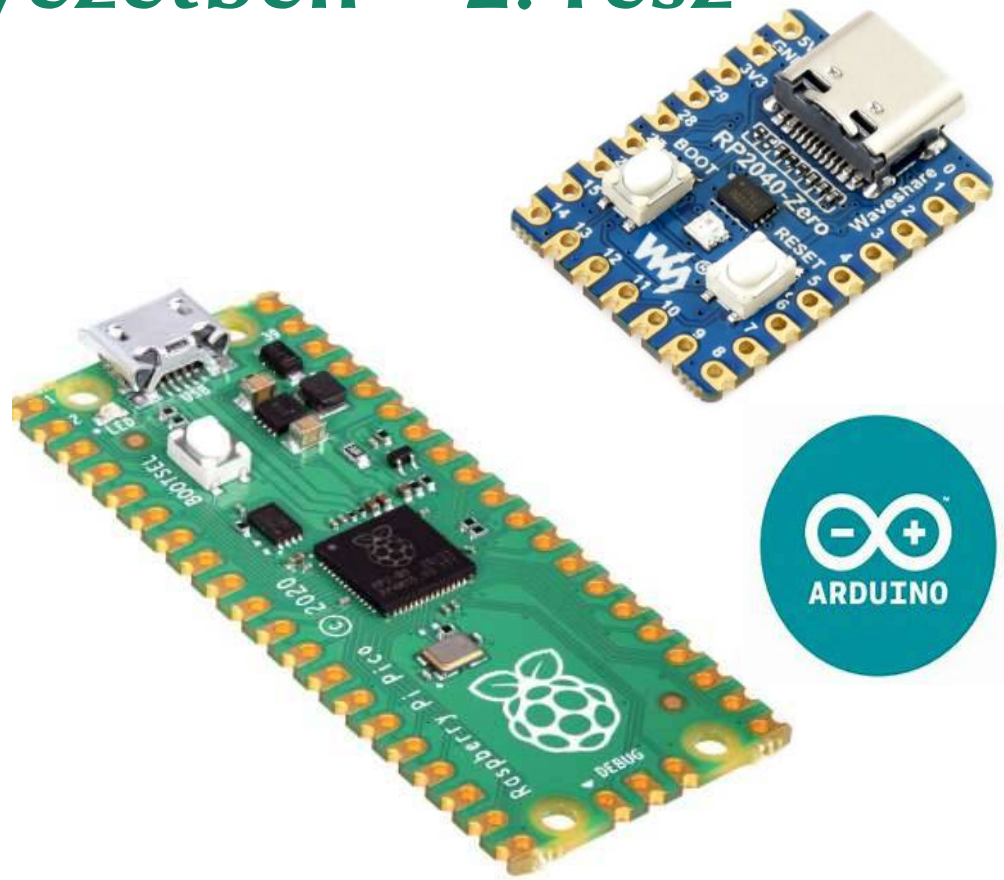
Bink2 | Arduino 1.8.19 (Windows Store 1.8.57.0)

File Edit Sketch Tools Help

Bink2 §

```
#include <mbed.h>

mbed::DigitalOut led(LED1);

void setup() {
}

void loop() {
  led.write(1);                        // Switch the LED on
  rtos::ThisThread::sleep_for(1000); // Wait for a second
  led.write(0);                        // Switch the LED off
  rtos::ThisThread::sleep_for(1000); // Wait for a second
}
```

15 Raspberry Pi Pico on COM14



Felhasznált és ajánlott irodalom

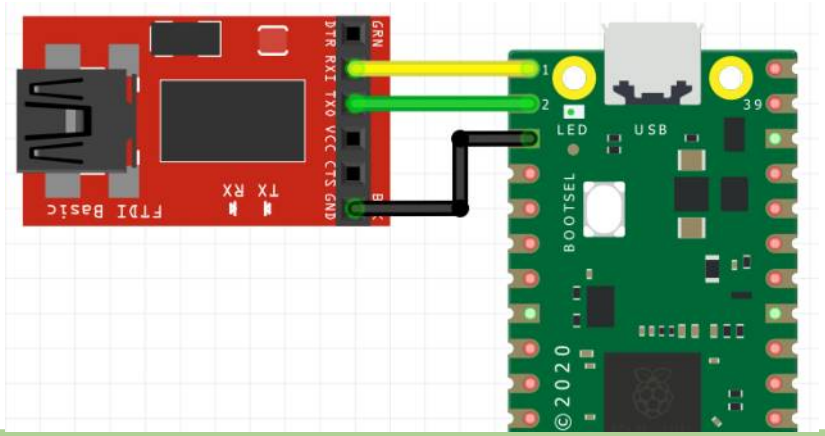
- Raspberry Pi: Pico-series Microcontrollers
- Raspberry Pi: RP2040 adatlap (PDF)
- Raspberry Pi: Raspberry Pi Pico Datasheet
- Raspberry Pi: Raspberry Pi Pico-series C/C++ SDK
- Raspberry Pi: Getting started with Raspberry Pi Pico-series Microcontrollers
- Waveshare: RP2040-Zero, a Pico-like MCU Board

- 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
- Alan Yorinks: NeoPixelConnect

Soros portok kezelése – Arduino programokban

- **Serial (USB):** Az alapértelmezett **Serial** kommunikáció a Raspberry Pi Pico-n az USB porton keresztül történik

```
void setup() {
  Serial.begin(9600); // Inicializálás 9600 bps
}

void loop() {
  Serial.println("Hello, Arduino!");
  delay(1000); // 1 másodperces késleltetés
}
```



- **Serial1 (UART):** a GPIO pineken keresztül használhatjuk a **Serial1**-et is, amely a **UART0**-n keresztül érhető el. Alapértelmezetten:
  - **GPIO0 – TX** (UART0 kimenete)
  - **GPIO1 – RX** (UART0 bemenete)

```
void setup() {
  Serial1.begin(9600); // Inicializálás 9600 bps
}

void loop() {
  Serial1.println("Hello, Arduino!");
  delay(1000); // 1 másodperces késleltetés
}
```

Soros portok kezelése – Arduino mbed programokban

- **Serial (USB):** Az USB porton kommunikáló **Serial** porton keresztül csak **Arduino** parancsokkal tudunk kiíratást végezni (**Serial.print(), Serial.write()**) formázott kiíratáshoz **sprintf()**-fel kombinálhatjuk…
- **Serial1 (UART):** az mbed standard output (printf()) alapértelmezetten **Serial1**-re van irányítva
  - **GPIO0 – TX** (UART0 kimenete)
  - **GPIO1 – RX** (UART0 bemenete)
- Megjegyzés: az **mbedOS printf()** függvénye *thread-safe*, többszálú kiíratásnál beépített mutex kezelést használ

```
#include "mbed.h"
using namespace rtos;
char buffer[50];

void setup() { Serial.begin(115200); }

void loop() {
  sprintf(buffer, "Temperature: %d C\n", 25);
  Serial.write(buffer, strlen(buffer));
  ThisThread::sleep_for(1s);
}
```

```
#include "mbed.h"
using namespace rtos;

void setup() {
  //Serial1.begin(115200);
}

void loop() {
  printf("Temperature: %d C\n", 25); //to Serial1
  ThisThread::sleep_for(1000);
}
```

printf() átirányítása Arduino mbed programokban

- A **printf()** formázási lehetőségei és az USB virtuális soros port kényelme egyidejű kihasználásához a **stdio** kimenetet át kell irányítani a **Serial** kimenetre

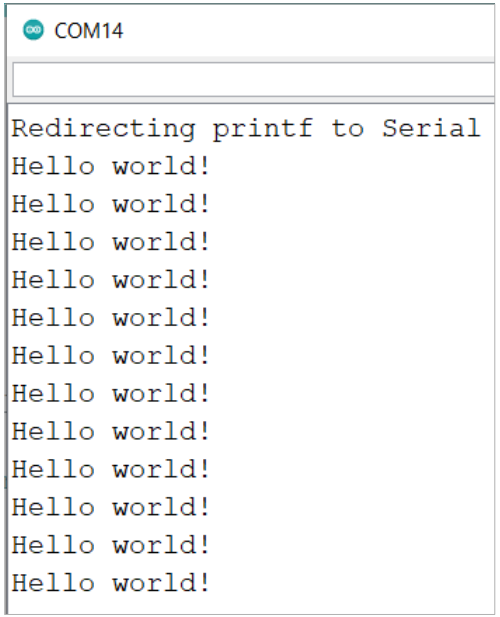
- Egy új virtuális soros eszközosztályt definiálunk, melynek **write()** és **read()** metódusai a **Serial.write()** és **Serial.readBytes()** függvényt hívják meg
- A példányosítás után az **mbed_override_console()** függvény az objektumpéldány címét fogja átadni konzol gyanánt (ez az átirányítás)
- Fentieket a **RedirectedSerial** nevű könyvtárban is közzé tettük a kényelmes használathoz

```
#include <Arduino.h>
#include "mbed.h"
class RedirectedSerial : public mbed::FileHandle {
public:
    virtual ssize_t write(const void *buffer, size_t size) {
        return Serial.write((const uint8_t*)buffer, size);
    }
    virtual ssize_t read(void *buffer, size_t size) {
        return Serial.readBytes((char*)buffer, size);
    }
  . . . .
}

RedirectedSerial mySerial;

namespace mbed {
    FileHandle *mbed_override_console(int fd) {
        return &mySerial;
    }
}
```

A RedirectedSerial könyvtár használata

- A **RedirectedSerial.h** fejléc állomány becsatolása lényegében az előző oldalon bemutatott átirányítást építi be a programba, ezután már csak a felhasználói programot (**setup()** és **loop()** függvény) kell hozzáadni a programhoz
- Ha a **printf()** függvényt használjuk, az most már a **Serial** virtuális soros portra ír (USB-n keresztül)

```
#include <RedirectedSerial.h>

void setup() {
    Serial.begin(115200);        // Initialize Serial (USB)
    delay(5000);
    printf("Redirecting printf to Serial\n"); // startup message
}

void loop() {
    printf("Hello world!\n");  // Print a message every second
    delay(1000);
}
```



Mbed programszálak kezelése

- A korábbi mbed tananyagban és tanfolyamon az „étkező filozófusok” probléma kapcsán találkoztunk már olyan esettel, hogy több programszál törzsét ugyanaz a függvény alkotta (itt a philosopher fv.). A programszálak és a paraméter-átadás kezelése azonban az újabb kiadású MbedOS API-ban megváltozott:
  - A programszál példányosítása és indítása nem vonható össze
  - Ha a programszálhoz rendelt függvénynek paramétert kell átadni, az csak közvetetten, a **callback()** függvénnyel adhatjuk meg

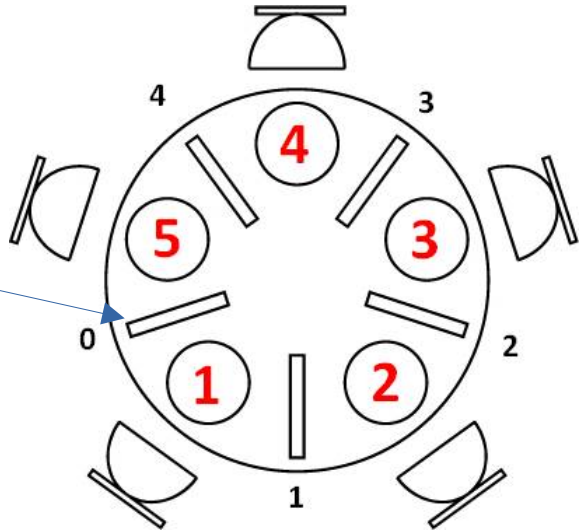
Az eredeti, **mbed v2.0**-ban írt programunk kritikus része:

```
int main() {
    Thread t2(philosopher, (void *)2U);
    Thread t3(philosopher, (void *)3U);
    Thread t4(philosopher, (void *)4U);
    Thread t5(philosopher, (void *)5U);
    philosopher((void *)1U);
}
```

A következő oldalon megmutatjuk, hogy ezeket a sorokat hogyan írhatjuk át az **MbedOS v6.1x**-nek megfelelően

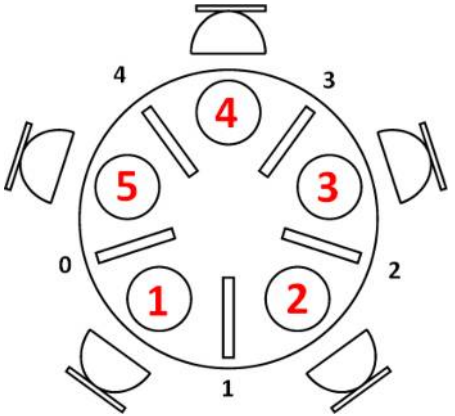
dining_philosophers.ino – 2/1.

```
#include "mbed.h"
using namespace mbed;
using namespace rtos;

Mutex chopstick[5];  // mutexes as chopsticks

int main() {
  int task_ids[5] = {1, 2, 3, 4, 5};
  Thread threads[4];
  // Start four threads
  for (int i = 0; i < 4; i++) {
    threads[i].start(callback(philosopher, &task_ids[i]));
  }

  // Run the fifth philosopher in the main thread
  philosopher(&task_ids[4]);
}
```



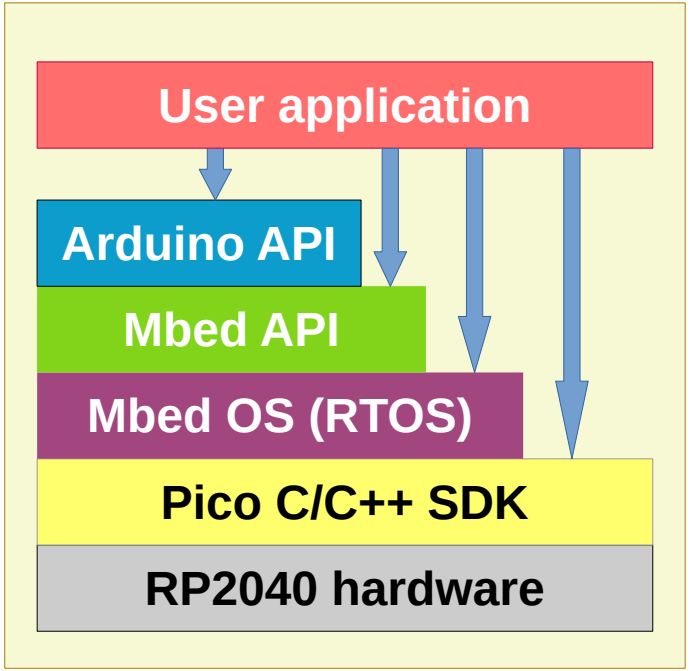
dining_philosophers.ino – 2/2.

```
void philosopher(const void *args) {
  int num = *(static_cast<const int*>(args)); // Typecast of args
  while (true) {
    if (chopstick[num - 1].trylock()) {
      if (chopstick[num % 5].trylock()) {
        printf("Philosopher %d is EATING\n", num); // Start EATING
        ThisThread::sleep_for(1000 + rand() % 1000);
        chopstick[num % 5].unlock();    // Release chopsticks
        chopstick[num - 1].unlock();
        printf("Philosopher %d is THINKING\n", num);  // Start THINKING
        ThisThread::sleep_for(2000 + rand() % 2000);  // Gets hungry after this time
      } else {
        chopstick[num - 1].unlock();
        ThisThread::sleep_for(100 + rand() % 100);    // Wait for random time if failed
      }
    } else {
      ThisThread::sleep_for(100 + rand() % 100);      // Wait for random time if failed
    }
  }
}
```



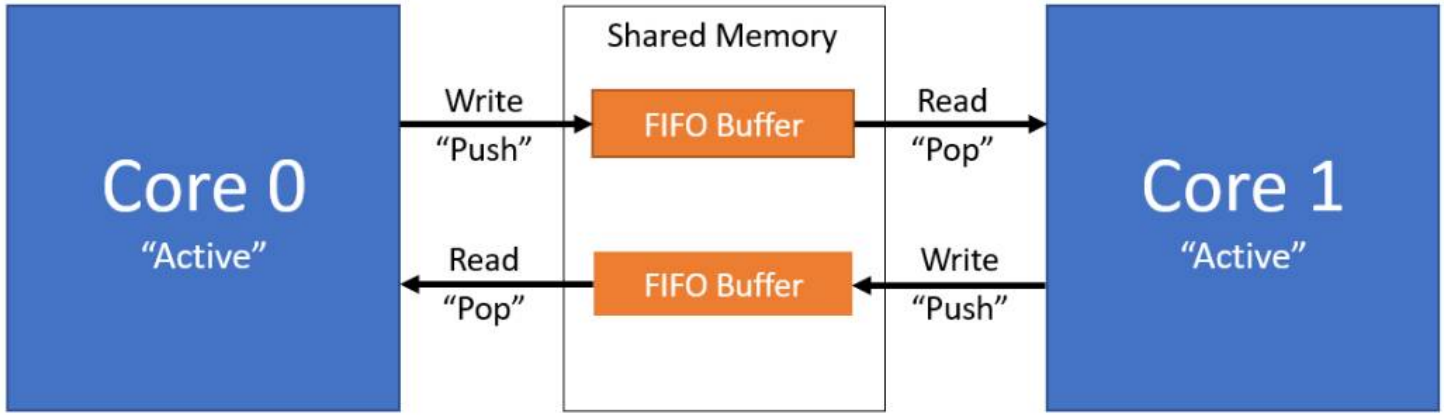
Arduino MbedOS for RP2040 Boards

- Az általunk használt **Arduino MbedOS RP2040 Boards** szoftver különböző rétegekből áll, melyek ismerete nélkülözhetetlen a hatékony programfejlesztéshez
- **Hardver:** a hardver (CPU-k, perifériák, memória)
- **SDK:** közvetlen(ebb) hozzáférést biztosít a hardver erőforrásaihoz
- **Mbed OS:** valós idejű operációs rendszer (CMSIS-RTOS RTX alapokon), ami biztosítja a multitasking és időzítési szolgáltatásokat, valamint az alapvető RTOS funkciókat
- **Mbed API:** objektumorientált programozási interfész a perifériák és a hardveres kommunikáció kezeléséhez
- **Arduino API:** egyszerűsített programozási interfész az az **RP2040** kártyákhoz, **Mbed OS**-alapokon
- **Felhasználói program:** hozzáférhet az Arduino API-hoz, az Mbed API-hoz , az RTOS szolgáltatásaihoz és az SDK API-hoz is, ha speciális funkcionalitásra van szükség



Kétmagos programok szinkronizálása

- Az **RP2040** két M0+ maggal rendelkezik, amelyeket külön-külön lehet programozni
- A program a fő magon (**core 0**) indul el, majd a másodlagos mag (**core 1**) futását a Pico C/C++ SDK `multicore_launch_core1()` függvényével indíthatjuk el
- A két magot mindkét irányban egy-egy 8x32 bites FIFO köti össze, ami adatcserére és szinkronizálásra használható, a **Pico C/C++ SDK** függvényeinek segítségével
- Az adatküldő/fogadó függvényhívások blokkoló típusúak (várakozás)
- A FIFO állapot lekérdezése (van-e beérkezett adat, vagy szabad hely a FIFO tárban) pedig nem blokkoló 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

Egy és kétmagos programfutás összehasonlítása

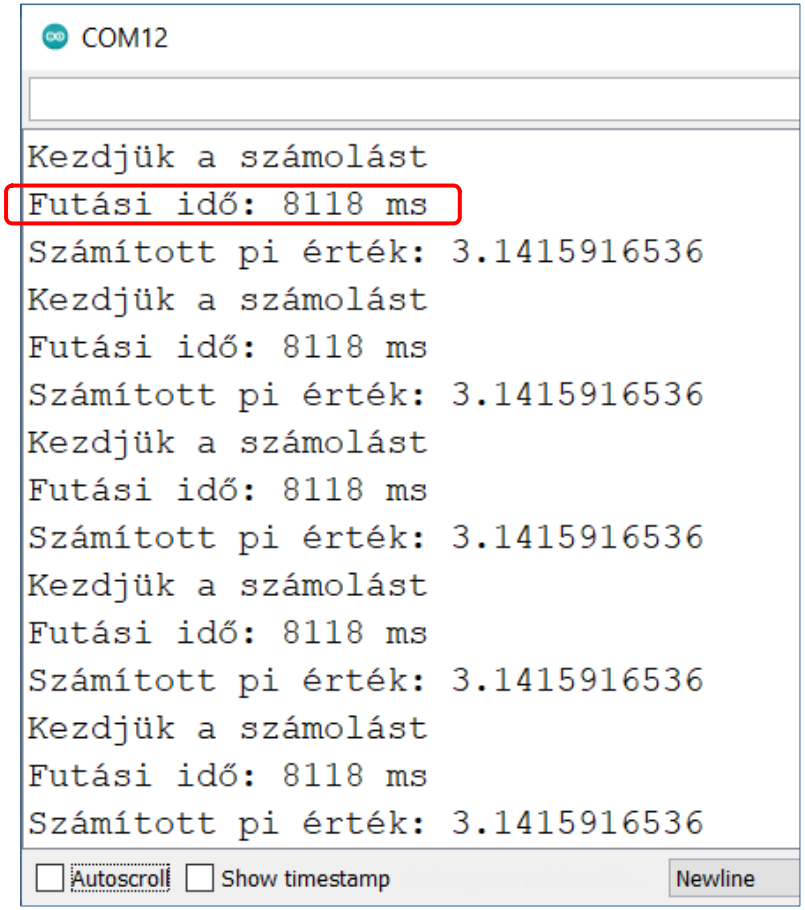
- Bemutatjuk, hogy az **RP2040** mikrovezérlőn a többmagos futtatás jelentősen csökkentheti a számítási időt a CPU intenzív feladatoknál
- A Leibnitz-sorozat segítségével számítjuk ki **π** közelítő értékét, és az egy- és kétmagos futtatás esetén mért számítási időket összehasonlítjuk

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

- Az alternáló Leibnitz sorozat lassan konvergál, így nagyszámú (n = 1 000 000) iterációt végezve, jól mérhető a végrehajtási idő
- **single_core.ino:** az egymagos számolás programja, számolási idő = 8118 ms
- **dual_core.ino:** a kétmagos programnál a feladatot megosztottuk, **core1** a sorozat első felét, **core0** pedig a második felét számolta ki, így a számolási idő = 4458 lett
- A kétmagos futtatás során a 100%-os teljesítménynövekedés helyett körülbelül 91%-ot értünk el a szinkronizációs és memória-hozzáférési időveszteségek miatt

single_core.ino

```ino
#include <Arduino.h>

void setup() { Serial.begin(115200); }

void loop() {
  Serial.println("Kezdjük a számolást");
  unsigned long start_time = millis();
  volatile double pi = 0.0;
  unsigned long n = 1000000; // Iterációk száma
  for (unsigned long i = 0; i < n; ++i) {
    if (i % 2 == 0) { pi += 1.0 / (2 * i + 1); }
      else { pi -= 1.0 / (2 * i + 1); }
  }
  pi *= 4; // A Gregory-Leibniz sorozat pi/4-et számolja
  unsigned long end_time = millis();
  unsigned long execution_time = end_time - start_time;
  Serial.print("Futási idő: ");   // Kiírjuk a futási időt
  Serial.print(execution_time);
  Serial.println(" ms");
  Serial.print("Számított pi érték: ");
  Serial.println(pi, 10); // 10 tizedesjegyig íratjuk ki
  delay(2000);            // 2 másodperc szünet
}
```



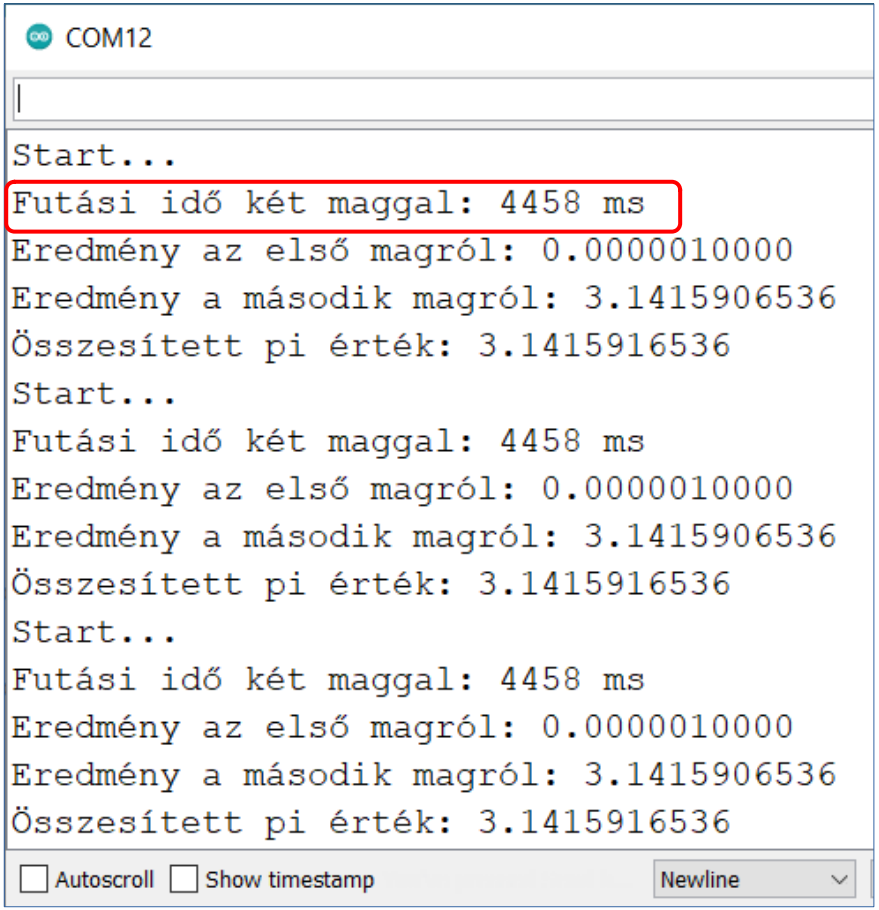
dual_core.ino – 2/1.

```ino
#include <Arduino.h>
#include "pico/multicore.h"

volatile double result0 = 0.0;
volatile double result1 = 0.0;
volatile int ledstate = LOW;
volatile unsigned long n = 1000000; // Iterations
volatile unsigned long n_2 = n / 2;

void core1_task() {
  while (true) {
    int flip = multicore_fifo_pop_blocking();
    digitalWrite(14, flip);     // GPIO14 flip
    volatile double pi = 0.0;
    for (unsigned long i = 0; i < n_2; ++i) {
      if (i % 2 == 0) {
        pi += 1.0 / (2 * i + 1);
      } else {
        pi -= 1.0 / (2 * i + 1);
      }
    }
    result1 = pi * 4;
    multicore_fifo_push_blocking(flip);
  }
}
```

```ino
void core0_task() {
  // Hosszú számolás (Gregory-Leibniz sorozat)
  volatile double pi = 0.0;
  for (unsigned long i = n_2; i < n; ++i) {
    if (i % 2 == 0) {
      pi += 1.0 / (2 * i + 1);
    } else {
      pi -= 1.0 / (2 * i + 1);
    }
  }
  result0 = pi * 4;
}

void setup() {
  Serial.begin(115200);
  pinMode(14, OUTPUT);
  delay(1000);
  // Elindítjuk a második magon futó feladatot
  multicore_launch_core1(core1_task);
}
```

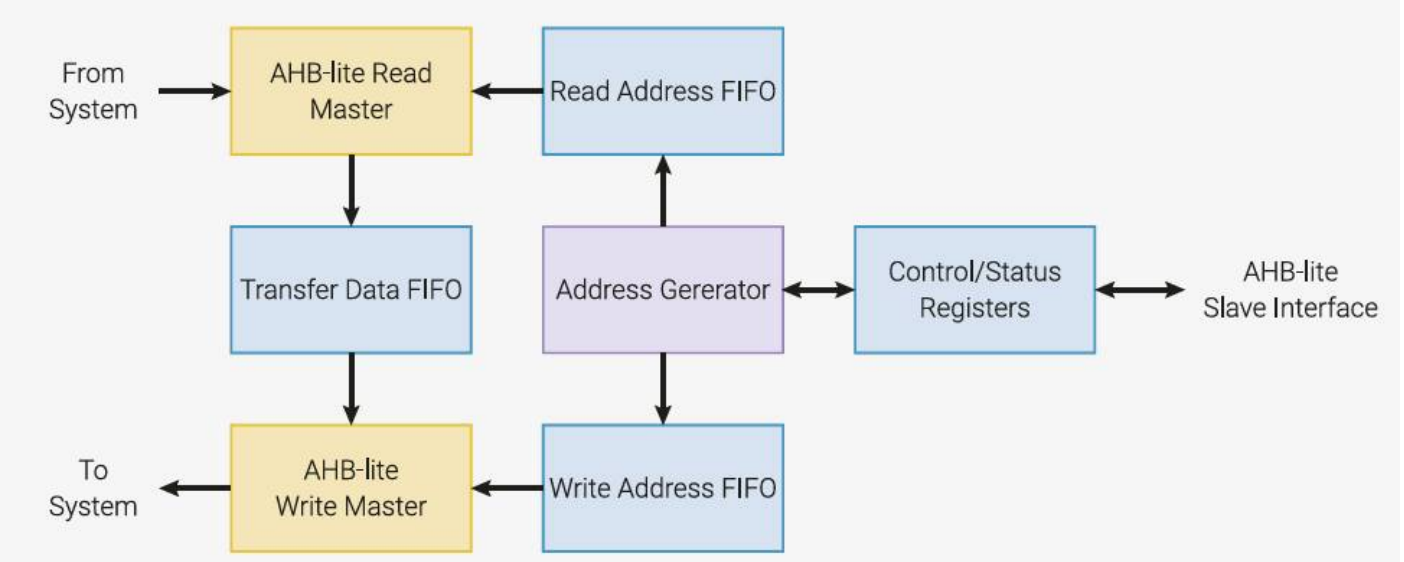
dual_core.ino – 2/2.

```
void loop() {
  Serial.println("Start...");
  unsigned long start_time = millis();
  multicore_fifo_push_blocking(ledstate);
  core0_task();
  int flop = multicore_fifo_pop_blocking();
  unsigned long end_time = millis();
  unsigned long execution_time = end_time - start_time;
  ledstate = !ledstate;
  // Kiírjuk a futási időt
  Serial.print("Futási idő két maggal: ");
  Serial.print(execution_time);
  Serial.println(" ms");
  Serial.print("Eredmény az első magról: ");
  Serial.println(result0, 10); // 10 tizedesjegyig pontos
  Serial.print("Eredmény a második magról: ");
  Serial.println(result1, 10); // 10 tizedesjegyig pontos

  // Összesített eredmény
  double total_pi = result0 + result1;
  Serial.print("Összesített pi érték: ");
  Serial.println(total_pi, 10); // 10 tizedesjegyig pontos
  delay(2000);
}
```



Az RP2040 DMA vezérlője

- A **DMA** vezérlő közvetlen és autonóm adatátvitelt végez a perifériák és/vagy memóriaterületek között. Külön olvasási és írási vezérlővel csatlakozik a buszhoz



- Az olvasási vezérlő minden órajelnél képes adatot olvasni egy címről, míg az írási vezérlő egy másik címre tud írni. A címgenerátor összehangolt olvasási és írási címeket állít elő, amelyeket a vezérlők a cím FIFO-kon keresztül használnak fel. Egyszerre akár 12 átviteli csatorna is aktív lehet. A szoftver vezérlő- és állapotregisztereken keresztül irányítja

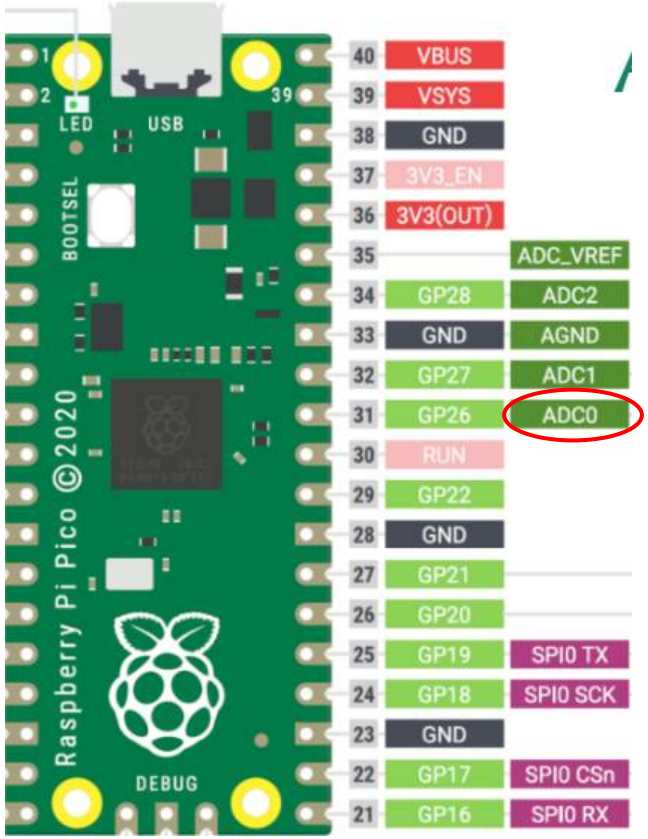
A DMA csatorna beállításai

A DMA csatorna beállításához a Pico C/C++ SDK függvényeit használhatjuk:

1) **DMA csatorna kiválasztása:** a **dma_claim_unused_channel()** függvénnyel válasszunk egy szabad DMA csatornát!
2) **DMA csatorna konfigurálása: dma_channel_get_default_config()** – az alapértelmezett konfiguráció megszerzése, majd a paraméterek beállítása, például az adatméretet (**channel_config_set_transfer_data_size()**), az olvasási cím növekményét (**channel_config_set_read_increment()**), és a **DREQ** forrást (**channel_config_set_dreq()**).
3) **DMA csatorna beállítása:** a **dma_channel_configure()** függvénnyel, megadjuk a célcímet, az olvasási címet és az adatátviteli méretet
4) **IRQ engedélyezése:** a DMA megszakítást a **dma_channel_set_irq0_enabled()** függvénnyel engedélyezzük, és az **irq_set_exclusive_handler()** függvénnyel állítjuk be
5) **Átvitel indítása:** a **dma_channel_set_read_addr()** függvénnyel, ha utolsó paramétere *true*, vagy a **dma_channel_start()** függvénnyel

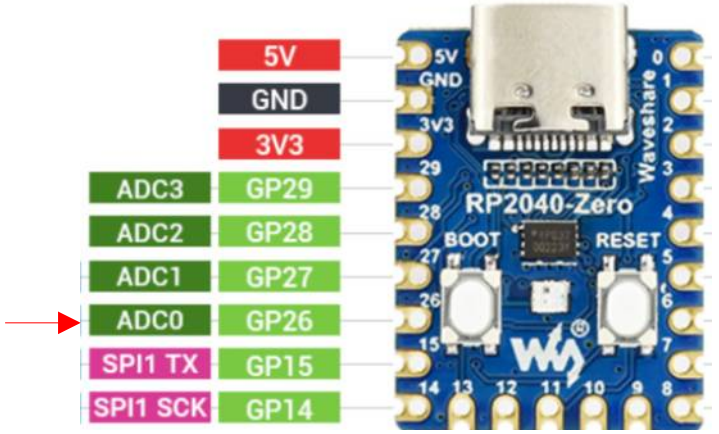
adc_dma.ino – 3/1.

```
#include <mbed.h>
#include <hardware/adc.h>
#include <hardware/dma.h>
#include <hardware/irq.h>

#define BUFFER_SIZE 512
#define ADC_CHANNEL 0
#define SAMPLE_RATE 1000

uint16_t adcBuffer[BUFFER_SIZE];
int dma_channel;
volatile bool bufferFull = false;

void dma_handler() {
  bufferFull = true;
  dma_hw->ints0 = 1u << dma_channel;  // IRQ jelzőbit törlés
}

void setup() {
  Serial.begin(115200);

  // ADC inicializálása
  adc_init();
  adc_gpio_init(26 + ADC_CHANNEL);  // GPIO 26, ha ADC_CHANNEL 0
  adc_select_input(ADC_CHANNEL);
```



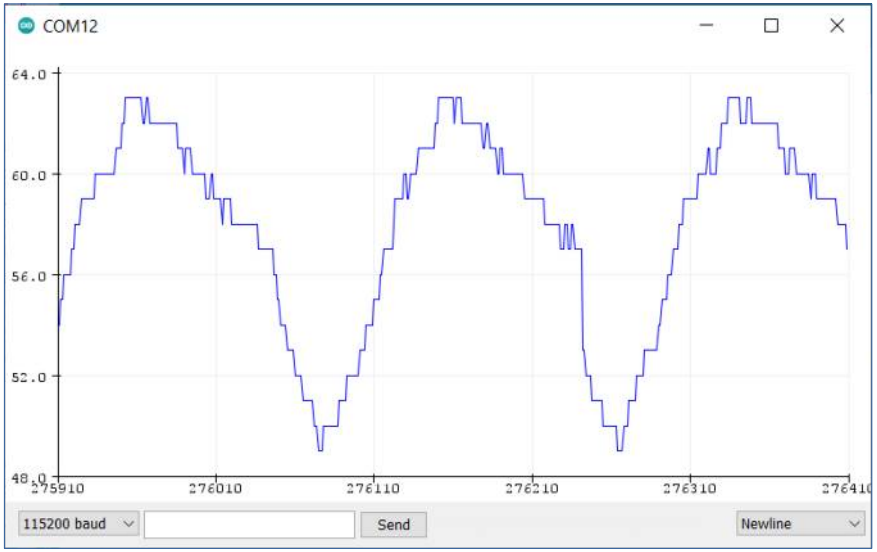
adc_dma.ino – 3/2.

```
// ADC FIFO beállítása
adc_fifo_setup(
  true,     // FIFO engedélyezése
  true,     // DMA adat kimenet engedélyezése
  1,        // FIFO küszöbérték (1 elem esetén DMA kérés)
  false,    // Error flag 15. bitben
  true      // Adat jobbra shiftelés
);

// DMA inicializálása
dma_channel = dma_claim_unused_channel(true);
dma_channel_config cfg = dma_channel_get_default_config(dma_channel);
channel_config_set_transfer_data_size(&cfg, DMA_SIZE_16);
channel_config_set_read_increment(&cfg, false);
channel_config_set_write_increment(&cfg, true);
channel_config_set_dreq(&cfg, DREQ_ADC);
dma_channel_configure(
  dma_channel,
  &cfg,
  adcBuffer,           // Cél cím (adcBuffer)
  &adc_hw->fifo,       // Forrás cím (ADC FIFO)
  BUFFER_SIZE,         // Átmásolandó adatok száma
  true                 // Azonnal indítsa a DMA-t
);
```

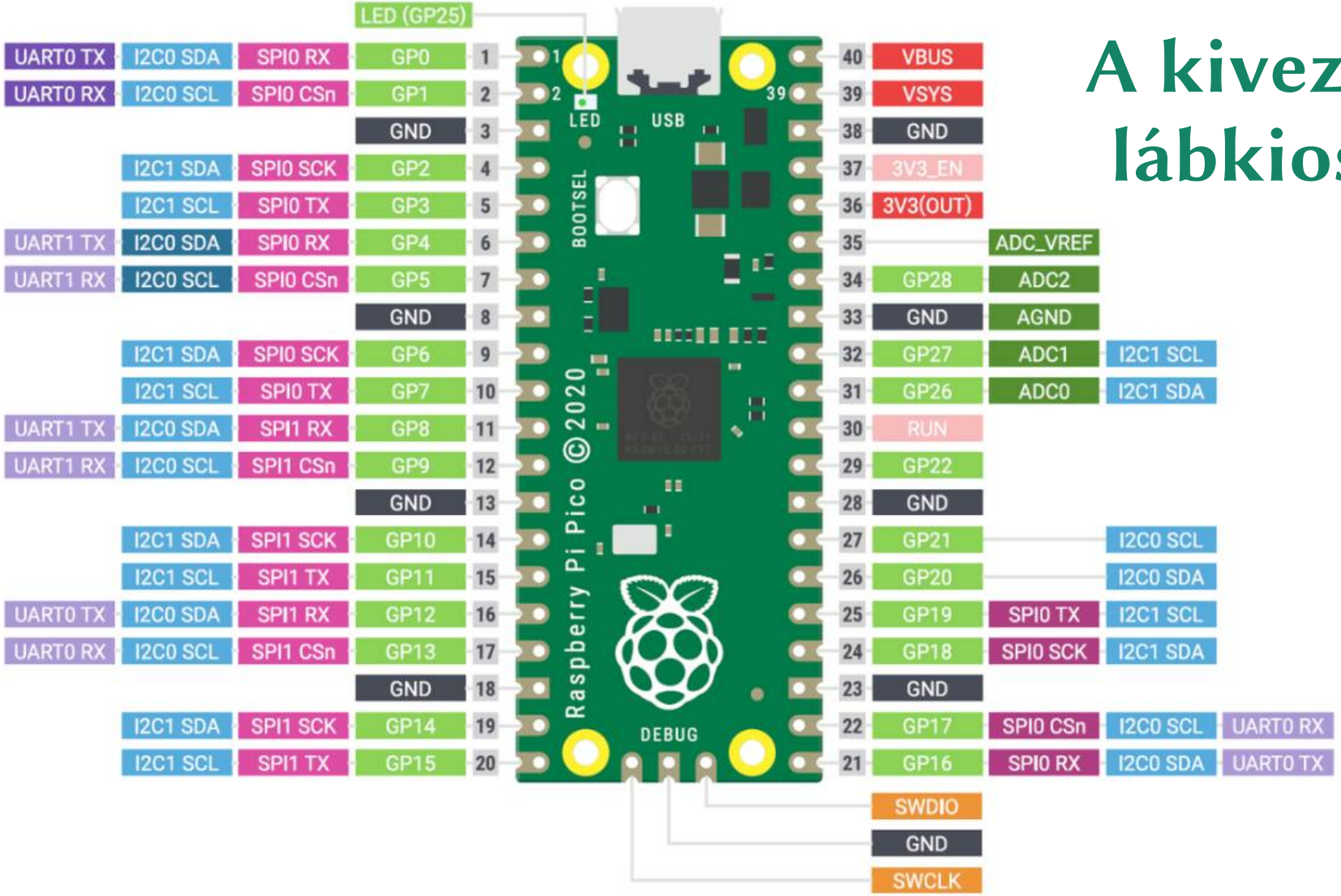


adc_dma.ino – 3/3.

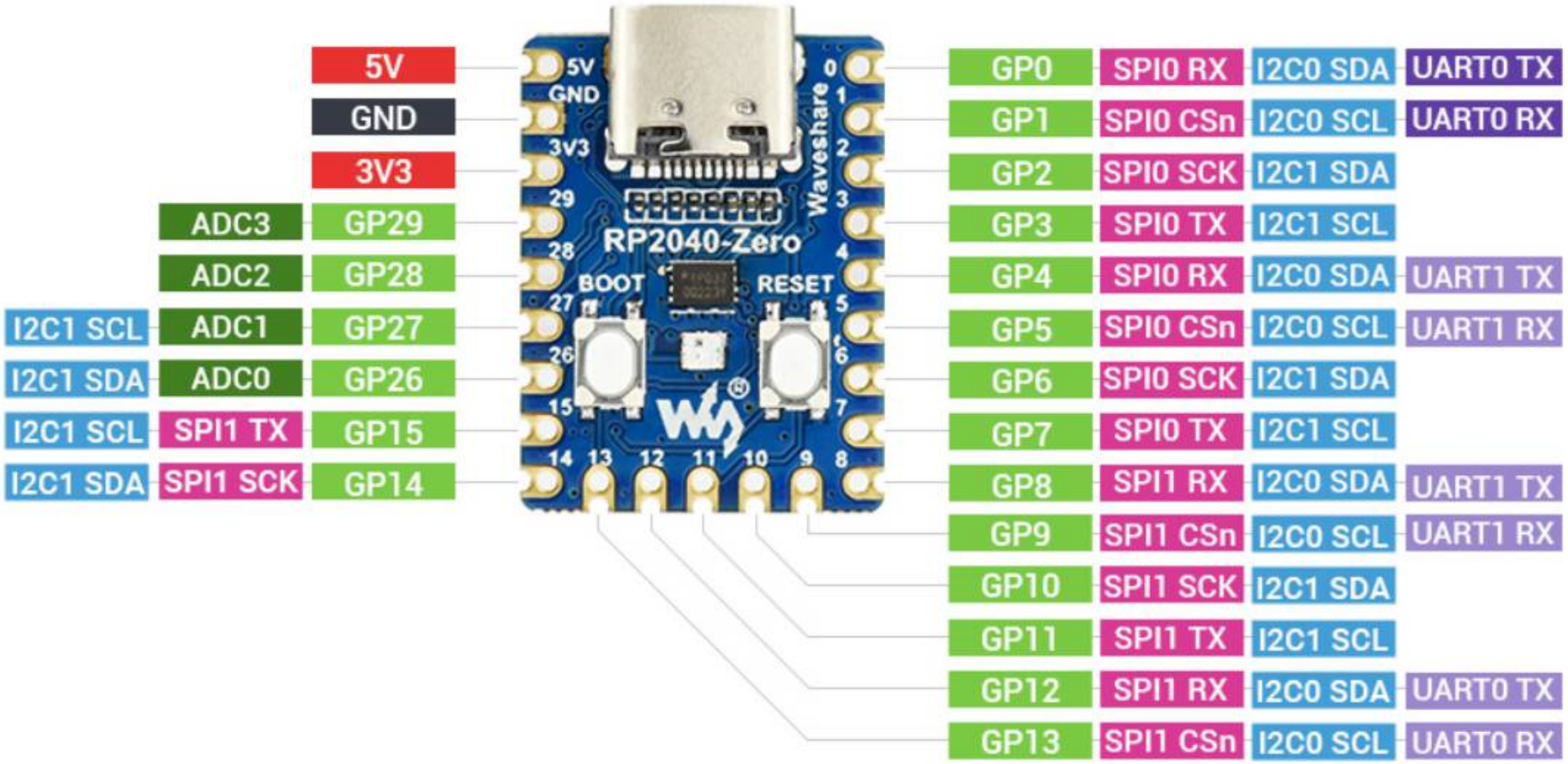
```
  // DMA megszakítás beállítása
  dma_channel_set_irq0_enabled(dma_channel, true);
  irq_set_exclusive_handler(DMA_IRQ_0, dma_handler);
  irq_set_enabled(DMA_IRQ_0, true);

  // ADC mintavételezési sebesség beállítása
  adc_set_clkdiv(48000000 / (SAMPLE_RATE * 96) - 1);
  adc_run(true);  // ADC indítása
}

void loop() {
  if (bufferFull) {
    bufferFull = false;
    // Adatok kilistázása
    for (int i = 0; i < BUFFER_SIZE; i++) {
      Serial.println(adcBuffer[i]);
    }
    // DMA újraindítása
    dma_channel_set_write_addr(dma_channel, adcBuffer, false);
    dma_channel_set_trans_count(dma_channel, BUFFER_SIZE, true);
  }
}
```



A kivezetések lábkiosztása



A kivezetések lábkiosztása



További kivezetések

- Néhány kivezetés a kártya hátoldalán van kivezetve