Az RP2040 mikrovezérlő programozása Arduino IDE környezetben – 3. rész

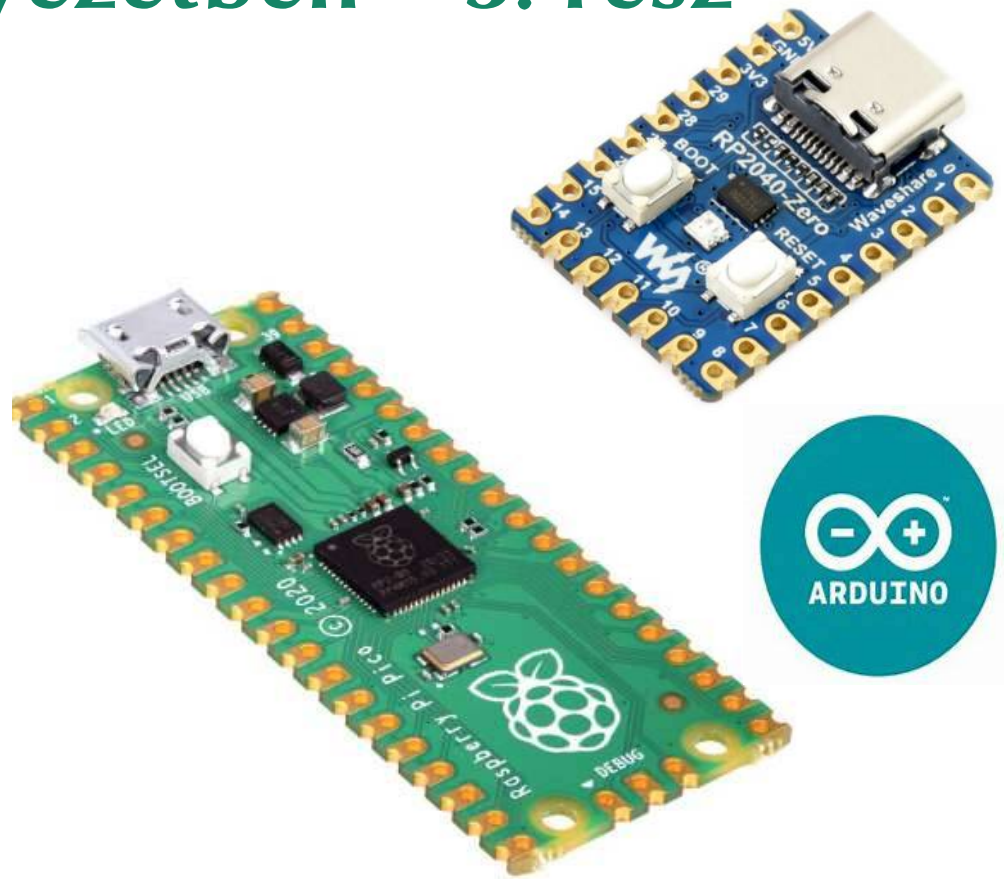
Bink2 | Arduino 1.8.19 (Windows Store 1.8.57.0)

File Edit Sketch Tools Help

Bink2 §

```
1  #include <mbed.h>
2
3  mbed::DigitalOut led(LED1);
4
5  void setup() {
6  }
7
8  void loop() {
9    led.write(1);                        // Switch the LED on
10   rtos::ThisThread::sleep_for(1000); // Wait for a second
11   led.write(0);                        // Switch the LED off
12   rtos::ThisThread::sleep_for(1000); // Wait for a second
13 }
```

15 Raspberry Pi Pico on COM14



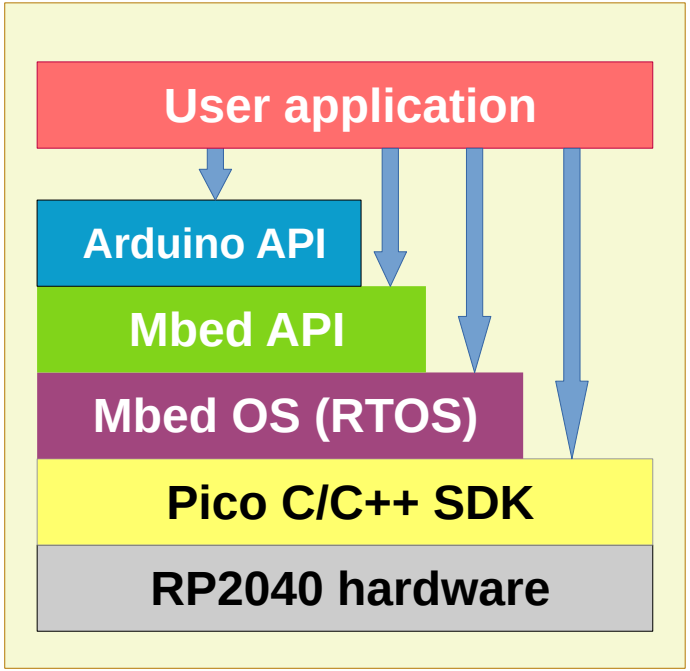
Felhasznált és ajánlott irodalom

- 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
- Valentin Milea: DHT sensor library for the Raspberry Pi Pico
- Alan Yorinks: NeoPixelConnect

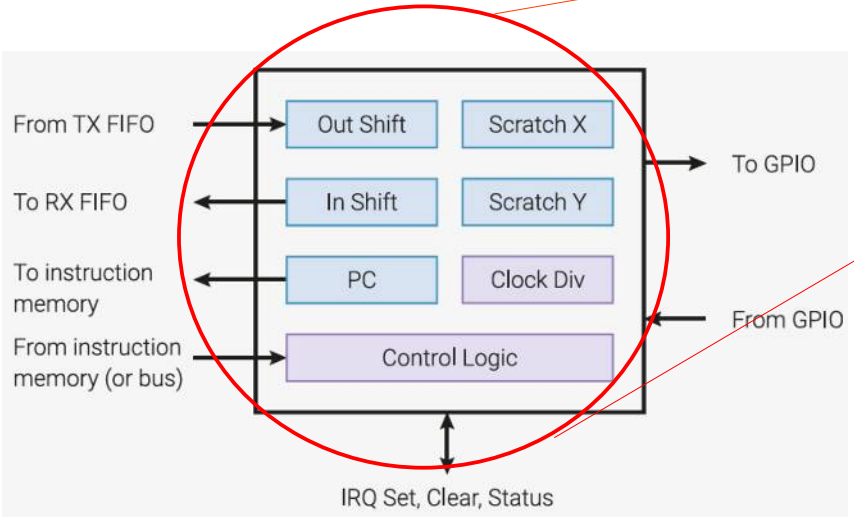
Emlékeztető: Arduino MbedOS for RP2040 Boards

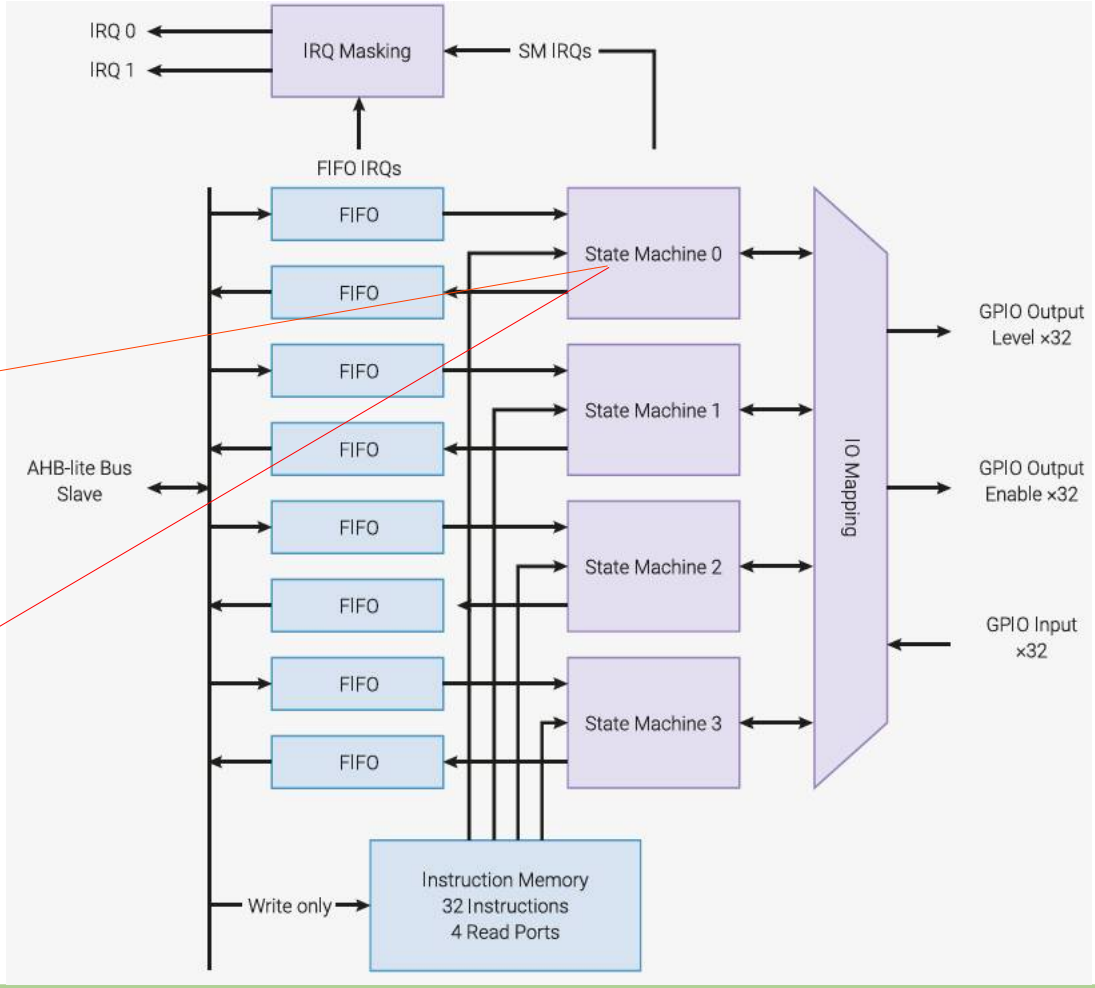
- Az általunk használt **Arduino MbedOS RP2040 Boards** szoftver különböző rétegekből áll, melyek ismerete nélkülözhetetlen a hatékony programfejlesztéshez
- **Hardver:** a hardver (CPU-k, perifériák, memória)
- **SDK:** közvetlen(ebb) hozzáférést biztosít a hardver erőforrásaihoz
- **Mbed OS:** valós 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



PIO – a programozható I/O blokk

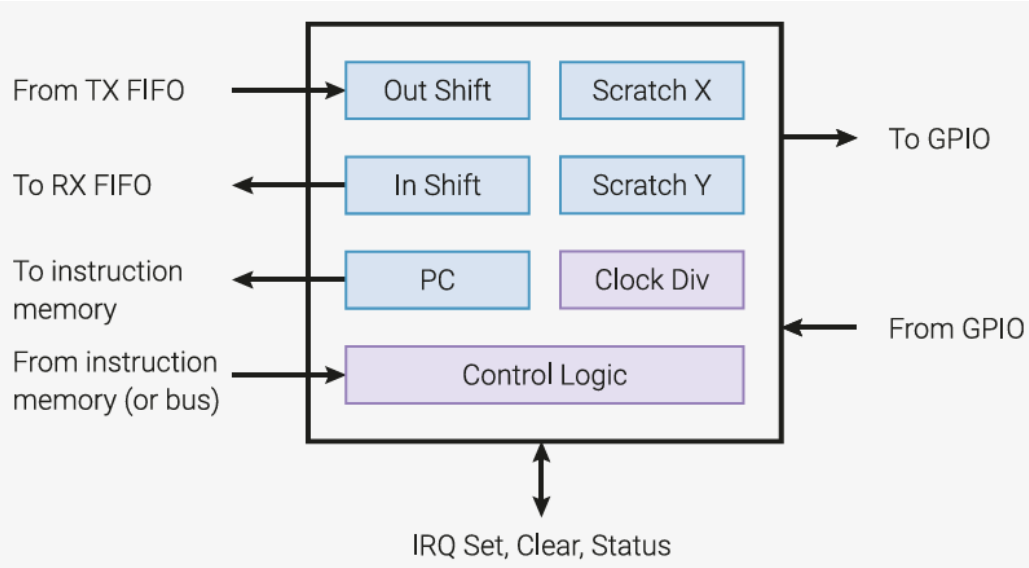
Két PIO blokk van, mindegyikben négy állapotgép található, amelyek egymástól függetlenül hajthatnak végre szekvenciális programokat a GPIO-k kezelésére és adatok átvitelére. A PIO állapotgépek erősen specializáltak az I/O műveletekre, a pontos időzítésre és a fix funkciójú hardverekkel való szoros integrációra helyezve a hangsúlyt





A PIO állapotgépei

- Mindegyik állapotgép tartalmazza az alábbiakat:
  - Két 32-bites shift regiszter (ISR és OSR) – mindkét irányban tetszőleges lépésszám
  - Két 32-bites adattároló regiszter (X és Y)
  - 4× 32-bites FIFO mindkét irányba (TX/RX), átkonfigurálható 8× 32 egyirányú FIFO-vá
  - Tört értékű órajel osztó (16 integer, 8 fractional bits)
  - Rugalmas GPIO hozzárendelés
  - DMA interfész
  - IRQ jelző
- Az állapotgépek programot hajtanak végre, s az állapotgépeket szinkronizáltan indíthatjuk és futtathatjuk, ha szükséges
- A PIO mindössze 9 utasítással rendelkezik:
  JMP, WAIT, IN, OUT, PUSH, PULL, MOV, IRQ, SET
- A PIO utasításokat kényelmi okokból általában a **pioasm.exe** assemblerrel fordítjuk le, ami C típusú fejléc állományt készít



PIO utasításkészlet

- A PIO utasítások 16 bitesek, az alábbi kódolás szerint:

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JMP	0	0	0	Delay/side-set					Condition			Address				
WAIT	0	0	1	Delay/side-set					Pol	Source		Index				
IN	0	1	0	Delay/side-set					Source			Bit count				
OUT	0	1	1	Delay/side-set					Destination			Bit count				
PUSH	1	0	0	Delay/side-set					0	IfF	Blk	0	0	0	0	0
PULL	1	0	0	Delay/side-set					1	IfE	Blk	0	0	0	0	0
MOV	1	0	1	Delay/side-set					Destination			Op		Source		
IRQ	1	1	0	Delay/side-set					0	Clr	Wait	Index				
SET	1	1	1	Delay/side-set					Destination			Data				

JMP utasítás

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JMP	0	0	0	Delay/side-set					Condition			Address				

- A **JMP** utasítás a PC programszámlálót **Adress**-re állítja, ha a **Condition** feltétel teljesül
- **Feltétel:**
  - **000: (feltétlen)** – mindig
  - **001: !X** – X regiszter tartalma nulla
  - **010: X--** – csökkentés előtt X nem nulla
  - **011: !Y:** – regiszter tartalma nulla
  - **100: Y--** – csökkentés előtt Y nem nulla
  - **101: X!=Y** – X és Y regiszterek tartalma különböző
  - **110: PIN** – ugrás, ha a bemenet magas szinten áll
  - **111: !OSRE** – az OSR regiszter tartalma nem nulla

- A **Delay/side-set** bitekkel 0 - 31 órajel késleltetést írhatunk elő, vagy mellékhatásként előre definiált GPIO kimenet(ek)et vezérelhetünk (side-set) – ez a többi utasításra is vonatkozik!

WAIT utasítás

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
WAIT	0	0	1	Delay/side-set					Pol	Source		Index				

- Felfüggeszti a futást, amíg az előírt feltétel nem teljesül
- **Pol** – polaritás (**1:** magas szintre vár, **0:** alacsony szintre vár)
- **Source** – jelforrás, amire várunk
  00 – GPIO (**Index** értéke szabja meg hogy melyik kimenetre vár)
  01 – PIN (Index értéke szabja meg, hogy az állapotgéphez rendelt bemenetek közül hanyadik)
  10 – ISR (Index értéke szabja meg, hogy melyik megszakításkérő bitre várunk)
  11 – fenntartott
- Az „zászló indexet” ugyanúgy dekódolják, mint az IRQ index mezőt: ha a legmagasabb helyiértékű bit (MSB) be van állítva, akkor a végrehajtó állapotgép azonosítóját (0–3) hozzáadják az IRQ indexhez moduló-4 művelettel az alsó két bit figyelembevételével. Példa: a 2-es állapotgép „0x11” zászlóértéknél a 3-as zászlóra várakozik, míg „0x13” értéknél az 1-es zászlóra. Ez lehetővé teszi, hogy azonos programot futtató több állapotgép egymással szinkronizáljon.

IN utasítás

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IN	0	1	0	Delay/side-set					Source			Bit count				

- Beléptet **Bit count** számú bitet a **Source** által megadott forrásból a Bemeneti Shift Regiszterbe (**ISR**). A léptetés irányát a **SHIFTCTRL_IN_SHIFTDIR** regiszterben lehet beállítani. Továbbá a beléptetési bitszámlálót megnöveli **Bit count**-tal (32-nél telítésbe megy)
- **Source** (forrás):
  - 000: PINS (GPIO bemenetek)
  - 001: X (scratch register X)
  - 010: Y (scratch register Y)
  - 011: NULL (csak arra szolgál, hogy ISR tartalmát léptethessük)
  - 100: Reserved
  - 101: Reserved
  - 110: ISR
  - 111: OSR
- Az **IN** mindig a forrás legkisebb helyiértékű bitjeit használja. Például, ha a **PINCTRL_IN_BASE** értéke 5, akkor az **IN PINS, 3** utasítás az 5., 6. és 7. láb értékeit olvassa be, és ezeket a Bemeneti Shift Regiszterbe (ISR) helyezi. Először az ISR bitjei balra vagy jobbra lépnek, hogy helyet csináljanak az új bemeneti adatoknak, majd a bemeneti adatokat bemásolják az így hagyott résbe. A bemeneti adatok bitsorrendje független a léptetés irányától.

OUT utasítás

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OUT	0	1	1	Delay/side-set					Destination			Bit count				

- Kiléptet **Bit count** számú bitet a Kimeneti Shift Regiszterből (**OSR**) és kiírja a **Destination** paraméter által megadott helyre. Továbbá a beléptetési bitszámlálót megnöveli **Bit count**-tal (32-nél telítésbe megy)
- **Destination** (cél):
- 000: PINS (GPIO kimenetek)
- 001: X (scratch register X)
- 010: Y (scratch register Y)
- 011: NULL (eldobja a kiléptetett adatbiteket)
- 100: PINDIRS (GPIO adatáramlási irány beállítása)
- 101: PC (feltétel nélküli ugrás a megadott címre)
- 110: ISR
- 111: EXEC (végrehajtja az OSR-ből kiforgatott adatot, mint utasítást)

PUSH utasítás

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PUSH	1	0	0	Delay/side-set					0	IfF	Blk	0	0	0	0	0

- Belerakja az **ISR** regiszter tartalmát az **RX FIFO**-ba, egy 32-bit szóként. Törli az **ISR** regisztert
- **IfFull:** Ha 1, akkor nem hajtódik végra PUSH utasítás, amíg az előírt beléptetési számot nem értük el. A **SHIFTCTRL_PUSH_THRESH** regiszter határozza meg, hogy az **ISR** regiszter tartalmának mennyi bitjét kell feltölteni, mielőtt egy PUSH művelet végrehajtható.
- **Block:** Ha 1 (ez az alapértelmezett érték), akkor a programvégrehajtás leáll és várakozik, amíg az **RX FIFO** betelt állapotban van. Ha a **Blk** bit 0, akkor betelt **RX FIFO** esetén nem áll meg a program, de ISR törlődik és az adat elvész
- **Megjegyzés:** A **PUSH** utasítás végrehajtása során az **ISR** teljes tartalma törlődik, ezért ha további feldolgozásra lenne szükség az **ISR**-ben tárolt adatokkal, azt a **PUSH** előtt kell elvégezni
- A **PUSH** utasítás az RP2040 PIO logikai analizátor példájában kulcsszerepet játszik az adatok begyűjtésében és továbbításában. Ebben a példában a PIO állapotgép a GPIO-k állapotát az **ISR** regiszterbe lépteti be, majd a **PUSH** segítségével az **RX FIFO**-ba mozgatja az adatokat.

PULL utasítás

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PULL	1	0	0	Delay/side-set					1	IfE	Blk	0	0	0	0	0

- Betölt egy 32 bites adatot a **TX FIFO**-ból az **OSR** regiszterbe
- **IfEmpty:** Ha 1, akkor nem hajtódik végra PULL utasítás, amíg az előírt kiléptetési számot nem értük el, azaz amíg **OSR** előző tartalmát nem ürítettük ki (lásd: **SHIFTCTRL_PULL_THRESH**)
- **Block:** Ha 1 (ez az alapértelmezett érték), akkor a programvégrehajtás leáll és várakozik, amíg a **TX FIFO** üres állapotban van. Ha a **Blk** bit 0, akkor üres **TX FIFO** esetén nem áll meg a program, hanem az **X** regiszter tartalma íródik az **OSR** regiszterbe. Más szavakkal: ha **TX FIFO** üres, akkor a **PULL NOBLOCK** utasítás a **MOV OSR, X** hatását idézi elő. Ez lehetővé teszi a program számára, hogy folyamatosan működjön, még akkor is, ha nem érkeztek új adatok a **TX FIFO**-ból

```
start:
pull noblock     ; FIFO-ból kiolvasás az OSR-be
mov x, osr       ; OSR tartalmának mentése X-be
 . . .
jmp start
```

A pull noblock utasítás és az adatok X-be mentése biztosítja a folyamatos működést, amelyben mindig az utoljára kapott adattal (pl. PWM esetén a kitöltés) dolgozunk

MOV utasítás

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOV	1	0	1	Delay/side-set					Destination			Op		Source		

- A **Source** bitekkel megadott forráshelyről a **Destination** bitekkel megadott célhelyre másolja az adatot, s közben az **Op** bitekkel specifikált műveletet is elvégzi az adatokon

- **Source:**
  - 000: PINS (pin mapping as IN)
  - 001: X
  - 010: Y
  - 011: NULL
  - 100: Reserved
  - 101: STATUS
  - 110: ISR
  - 111: OSR

- **Operation:**
  - 00: None
  - 01: Bitwise complement
  - 10: Bit-reverse
  - 11: Reserved

- **Destination:**
  - 000: PINS (pin mapping as OUT)
  - 001: X
  - 010: Y
  - 011: Reserved
  - 100: EXEC
  - 101: PC
  - 110: ISR (count ⟵ 0)
  - 111: OSR (count ⟵ 0)

IRQ utasítás

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IRQ	1	1	0	Delay/side-set					0	Clr	Wait	Index				

- Beállítja vagy törli az **Index** bitekkel kijelölt IRQ jelzőbitet
- **Clear:** Ha a **Clr** bit 1, akkor a megadott IRQ jelzőbit törlődik, ahelyett hogy beállításra kerülne (Ha a **Clr** bit 1, akkor a **Wait** bit nem játszik szerepet)
- Wait: Ha a **Wait** bit 1, akkor az állapotgép megállítja a végrehajtást, amíg a megadott IRQ megszakításjelző törlésre nem kerül (például egy megszakításkezelő által)
- **Index:**
  - **LSB:** Az Index bitcsoport legalsó 3 bitje választja ki az IRQ jelzőbitek közül, hogy melyiket állítsa be vagy törölje az utasítás (0 és 7 között)
  - **MSB:** Ha az Index bitcsoport legfelső bitje 1-be van állítva, akkor a **state machine ID** (állapotgép-azonosító, 0–3) hozzáadódik a kiválasztott indexhez modulo-4 összeadás segítségével. Ez lehetővé teszi, hogy a különböző állapotgépek ugyanazon a programon belül eltérő IRQ jelzőbiteket használjanak
- A 0–3 IRQ jelzőbitek a rendszer szintű interruptokhoz csatlakoztathatók (PIO IRQ0/IRQ1 vonalak)
- A 4–7 IRQ jelzőbitek csak a **PIO** állapotgépei között használhatók.

SET utasítás

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SET	1	1	1	Delay/side-set					Destination			Data				

- Az utasításban megadott **Data** értéket a **Destination** bitekkel adott helyre írja
- **Data** – 5 bites közvetlen érték a kivezetések vagy a regiszterek beállításához
- **Destination:**
  - 000: PINS
  - 001: X register (5 LSBs are set to Data, all others cleared to 0)
  - 010: Y register (5 LSBs are set to Data, all others cleared to 0)
  - 011: Reserved
  - 100: PINDIRS
  - 101: Reserved
  - 110: Reserved
  - 111: Reserved

Side-set

- A **side-set** funkció lehetővé teszi, hogy az **RP2040 PIO** állapotgépei akár 5 GPIO szintjét vagy irányát módosítsák a PIO állapotgép utasítások végrehajtásával szinkronban. Ez különösen hasznos gyors interfészek, például SPI esetében, ahol az órajel (pl. 1⟶0 vagy 0⟶1 átmenet) és az adatátmenet (OSR-ből egy új bit kiléptetése a GPIO-ra) szinkronban kell, hogy történjen
- A **side-set** használata csökkenti a program méretét, javítja az időzítési pontosságot, és lehetővé teszi a magasabb frekvenciájú SPI órajeleket, ahogy ezt az alábbi programocska is szemlélteti
- A **side-set** adat a **Delay/side-set** mezőbe van kódolva minden utasításban és a .side_set n direktívával mondjuk meg, hogy az 5 bites mezőből hány bitet foglal le a **side-set** (annyi lábat tud vezérelni)
- A **side-set** bármelyik utasítással kombinálható
- A **side-set** lábkiosztása független az **OUT** és **SET** által használt lábkiosztástól, bár átfedhet velük
- Ha a **side-set** és az **OUT** vagy **SET** ugyanarra a PIN-re ír egyszerre, akkor a **side-set** adat kerül alkalmazásra.

```
1 .program spi_tx_fast
2 .side_set 1
3
4 loop:
5 out pins, 1 side 0
6 jmp loop side 1
```

Itt az adatküldés 1bit/2órajel ciklus sebességgel zajlik

PIO .wrap_target / .wrap direktívák

- A PIO **.wrap_target** és **.wrap** direktívák lehetővé teszik, hogy a program automatikusan visszatérjen egy adott ciklus elejére, megtakarítva a különálló **JMP** utasítás használatát. Ez nem csak a kód tömörségét növeli, hanem értékes órajel-ciklusokat takarít meg
- Az alábbi példában megmutatjuk, hogy 50 %-os kitöltésű négyszögjel keltéséhez a **JMP** utasítás használatakor legalább 4 órajel-ciklus kell, míg a **.wrap** használatakor 2 is elég

4 ciklus

```
.program squarewave_jmp
loop:
    set pins, 1  ; GPIO magasra állítása
    nop          ; kompenzálás 50 %-hoz
    set pins, 0  ; GPIO alacsonyra állítása
    jmp loop     ; Ciklus elejére ugrás
```

2 ciklus

```
.program squarewave_wrap
.wrap_target
    set pins, 1  ; GPIO magasra állítása
    set pins, 0  ; GPIO alacsonyra állítása
.wrap
```

- Ahogy a fenti példa is szemlélteti, a **.wrap** direktívák használata különösen ott értékes, ahol nagyon magas frekvenciájú jelek generálására vagy szoros időzítésre van szükség, mint pl. SPI vagy UART kommunikáció, quadratura encoderek, LED mátrixok vezérlése

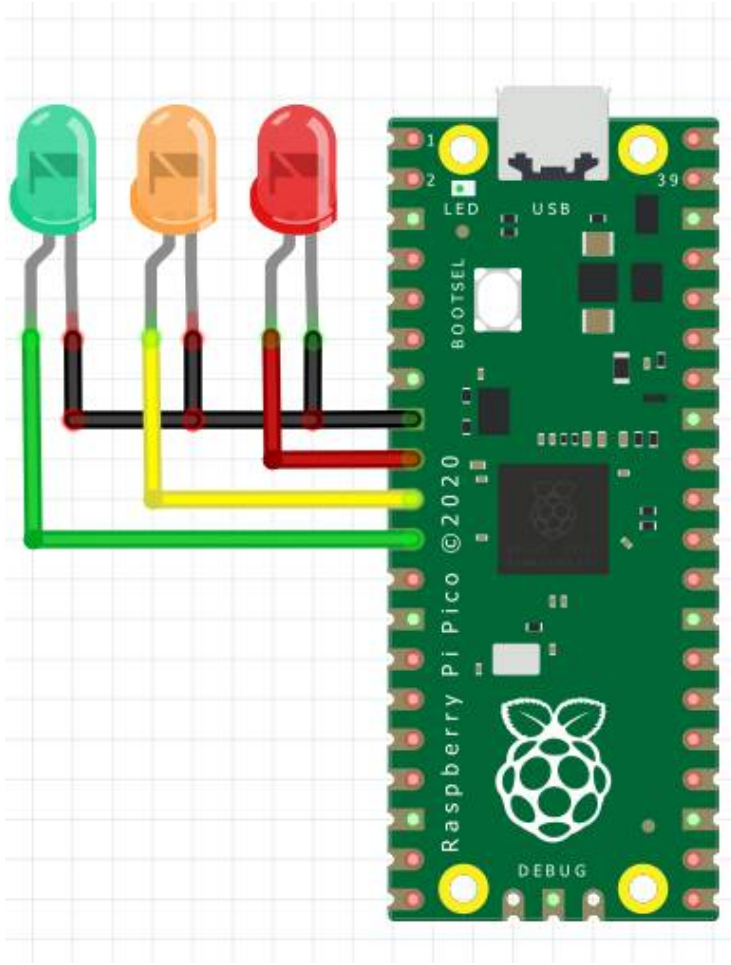
Raspberry Pi Pico C/C++ SDK PIO mintapéldák

A Raspberry Pi Pico SDK Examples gyűjtemény a PIO modulra az alábbi példaprogramokat tartalmazza:

- **addition** – Két szám egyszerű összeadása a PIO segítségével
- **apa102** – APA102 típusú LED szalagok vezérlése
- **clocked_input** – Órajellel szinkronizált adatbevitel példája
- **differential_manchester** – Differenciális Manchester kódolás megvalósítása
- **hello_pio** – Alapvető PIO program bemutatása
- **hub75** – Hub75 alapú LED mátrixok vezérlése
- **i2c** – I2C protokoll használata adatátvitelhez
- **ir_nec** – NEC infravörös kommunikáció implementációja
- **logic_analyser** – Logikai analizátor megvalósítása a PIO-val
- **manchester_encoding** – Manchester kódolási technika példája
- **onewire** – Egyvezetékes kommunikációs protokoll használata
- **pio_blink** – LED villogtatása PIO-val
- **pwm** – Impulzusszélesség modulációs (PWM) kimenet generálása
- **quadrature_encoder** – Forgásszög kódolók jeleinek kezelése
- **spi** – SPI protokoll használata kommunikációhoz
- **squarewave** – Négyszögjel generálása PIO használatával
- **st7789_lcd** – ST7789 típusú LCD képernyő vezérlése
- **uart_rx** – UART adatfogadás implementációja
- **uart_tx** – UART adatküldés megvalósítása
- **ws2812** – WS2812 típusú LED-ek vezérlése

LED-ek villogtatása PIO állapotgépekkel

- Az alábbi program a **Raspberry Pi RP2040** mikrovezérlő **PIO** perifériájával autonóm módon vezérli a LED-eket, különböző frekvenciákon villogtatva (3 Hz, 4 Hz, 1 Hz).
- Az állapotgépek mind ugyanazt a **PIO** programot futtatják, azonban különböző paraméterekkel inicializálva, így eltérő frekvenciájú működést érnek el
- A LED-ek anódjait itt a **GPIO6, GPIO7** és **GPIO8** kimenetekre kötöttük, természetesen bármelyik másik lábra is köthetnénk, ezt az állapotgépek inicializálásánál adhatjuk meg, a villogtatási frekvenciával együtt



blink.pio

```
.program blink
    pull block      ; Read period data from FIFO
    out y, 32       ; Store data in Y register
.wrap_target        ; Cycle begins here
    mov x, y        ; Get saved delay parameter
    set pins, 1     ; Turn LED on
lp1:
    jmp x-- lp1     ; Delay for (x+1) cycles
    mov x, y        ; Get saved delay parameter
    set pins, 0     ; Turn LED off
lp2:
    jmp x-- lp2     ; The same delay cycles again
.wrap               ; Blink forever!
```

Az adat beolvasása és eltárolása csak egyszer, az állapotgép inicializálásakor fut le. Itt vesszük át a késleltetéshez szükséges ciklusok számát

A **`.wrap_target`** és a **`.wrap`** direktívák közé zárt ciklustörzs automatikusan ismétlődik (végtelen ciklus)

A **`jmp x-- label`** utasítás csak akkor ugrik (vissza) a megadott címre, ha x értéke nem nulla

```
% c-sdk {
void blink_program_init(PIO pio, uint sm, uint offset, uint pin) {
   pio_gpio_init(pio, pin);
   pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);
   pio_sm_config c = blink_program_get_default_config(offset);
   sm_config_set_set_pins(&c, pin, 1);
   pio_sm_init(pio, sm, offset, &c);
}
%}
```

A **`blink_program_init()`** segítő függvény nem az assembly program része, csupán azért itt definiáljuk, hogy a **blink.pio.h** állományba átmásolódva megkönnyítse az állapotgép inicializálását

pio_blink.ino

```
#include "hardware/clocks.h"
#include "hardware/pio.h"
#include "blink.pio.h"                          // Generált PIO fejlécfájl

void blink_pin_forever(PIO pio,uint sm,uint offset,uint pin,uint freq)
{
    blink_program_init(pio, sm, offset, pin);
    pio_sm_set_enabled(pio, sm, true);
    uint32_t delay_cycles = (clock_get_hz(clk_sys)/(2 * freq)) - 3;
    pio_sm_put(pio, sm, delay_cycles);          // Send period to FIFO
}

void setup() {
    PIO pio = pio0;
    uint offset = pio_add_program(pio, &blink_program);
    blink_pin_forever(pio, 0, offset, 6, 3); // GPIO6 3 Hz
    blink_pin_forever(pio, 1, offset, 7, 4); // GPIO7 4 Hz
    blink_pin_forever(pio, 2, offset, 8, 1); // GPIO8 1 Hz
}

void loop() {
  // Do nothing
}
```

Ez a függvény elindít egy állapotgépet, melynek sorszámát (sm), az általa kezelt GPIO láb sorszámát (pin) és a villogtatási frekvenciát (freq) paraméterként adjuk meg

A PIO objektumosztály példányosítása és programjának betöltése után elindítunk három állapotgépet, más-más paramé-terekkel (sm, pin, freq)

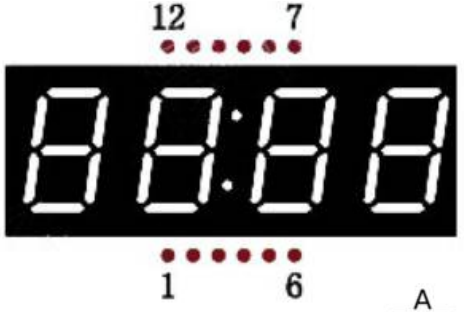
Van másik! – pio_ledblink.ino

- Van másik út: nem muszáj az assemblerrer bajlódni, a PIO utasítások az SDK API hívásokkal is megadhatók
- A mellékelt listán bemutatjuk, mivel kell megtoldani az előző oldalon bemutatott programot az `#include "blink.pio.h"` sor helyett
- A `.wrap_target` és a `.wrap` direktíváknak megfelelő beállítást a pirossal kiemelt sor biztosítja (ez az előzőekben a **blink_pio.h** fejléc állományban volt elrejtve, onnan emeltük át

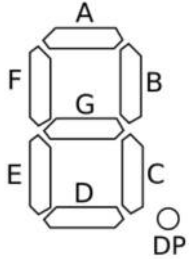
```
static const uint16_t blink_instructions[] = {
    pio_encode_pull(false, true),      // 0: pull   block
    pio_encode_out(pio_y, 32),         // 1: out    y, 32
    pio_encode_mov(pio_x, pio_y),      // 2: mov    x, y
    pio_encode_set(pio_pins, 1),       // 3: set    pins, 1
    pio_encode_jmp_x_dec(4),           // 4: jmp    x--, 4
    pio_encode_mov(pio_x, pio_y),      // 5: mov    x, y
    pio_encode_set(pio_pins, 0),       // 6: set    pins, 0
    pio_encode_jmp_x_dec(7)            // 7: jmp    x--, 7
};
const pio_program_t blink_program = {
  .instructions = blink_instructions,
  .length = 8,                              // Az utasítások száma
  .origin = -1
};

void blink_program_init(PIO pio, uint sm, uint offset, uint pin) {
  pio_gpio_init(pio, pin);
  pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);
  pio_sm_config c = pio_get_default_sm_config();
  sm_config_set_wrap(&c, offset+2, offset+7);  // Wrap címek
  sm_config_set_set_pins(&c, pin, 1);
  pio_sm_init(pio, sm, offset, &c);
}
```

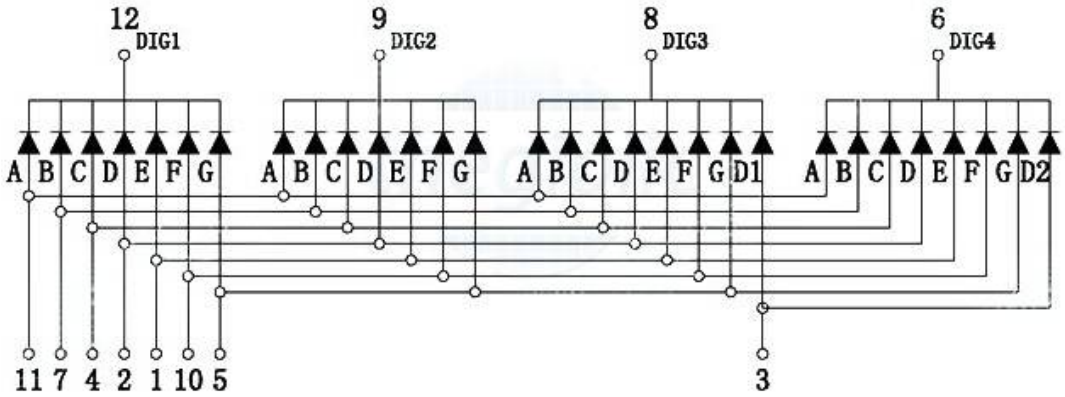
Hétszegmens kijelző, multiplex vezérléssel

- A hétszegmenses, 4 digites, közös katódú kijelzőnél az egyes számjegyek közös katódjai egy-egy vezérlő lábra vannak kötve, míg a szegmensek anódjait szegmensenként közösítjük
- A multiplex kijelzés során a számjegyeket felváltva, egymás után aktiváljuk, s ezzel szinkronban vezéreljük a szegmens bemeneteket
- Ha a fenti folyamatot gyorsan és folyamatosan ismételgetjük, akkor az emberi szem számára folyamatosan világítónak tűnnek a számjegyek
- Hogy a CPU-t ne terheljük és a programszervezést ne bonyolítsuk, a multiplex kijelzést most a PIO modulok egyik állapotgépével fogjuk megvalósítani.
- **Ennek lépései:**
  1. FIFO-ból adatbeolvasás (ha van) és mentés (32 bit)
  2. Utolsó számjegy kiírás és DIG4 aktiválás
  3. Harmadik számjegy kiírás és DIG3 aktiválás
  4. Második számjegy kiírás és DIG2 aktiválás
  5. Első számjegy kiírás és DIG1 aktiválás
  6. Ugrás a ciklus elejére



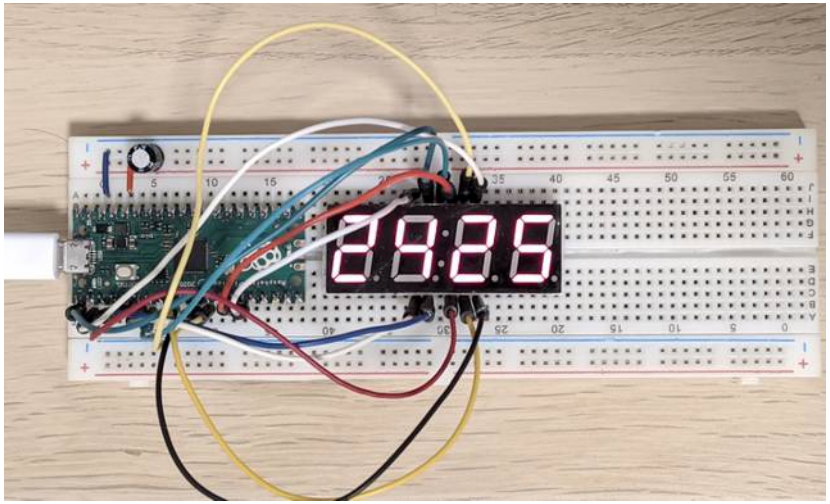
5463AS





Hétszegmens kijelző, multiplex vezérléssel

- A bekötési lista a mellékelt táblázatban látható
- Az egyes szegmenseket 100 – 220 Ω értékű áramkorlátozó ellenállásokon keresztül illik bekötni
- A mellékelt programban a DP vezérlő jelet (ami a kijelzőn esetében a középső kettőspontot gyújtja ki) nem használjuk, így a GPIO7 és a kijelző 3-as lábának összekötése fölösleges



GPIO	Funkció	Kijelző	
0	A	11	szegmensek
1	B	7	
2	C	4	
3	D	2	
4	E	1	
5	F	10	
6	G	5	
7	DP	3	
8	DIG4	6	számjegyek
9	DIG3	8	
10	DIG3	9	
11	DIG1	12	

side-set vezérléssel

pio_7seg.pio

```
.program pio_7seg
.side_set 4 opt          ; 4 bites side-set a digitvezérléshez (GPIO 8-11)
.wrap_target             ; A ciklustörzs elejének jelölése
start:
pull noblock             ; 1. FIFO-ból kiolvasás az OSR-be, ha van új adat
mov x, osr               ; OSR tartalmának másolása az X-be (mentés)
out pins, 8 side 0b1110 ; 2. Első bájt kiküldése a szegmensekre, GPIO8 LOW (aktiválás)
nop                      ; Késleltetés
nop                      ; Késleltetés
nop side 0b1111          ; GPIO8 HIGH (deaktiválás)
out pins, 8 side 0b1101 ; 3, Második bájt kiküldése a szegmensekre, GPIO9 LOW (aktiválás)
nop                      ; Késleltetés
nop                      ; Késleltetés
nop side 0b1111          ; GPIO9 HIGH (deaktiválás)
out pins, 8 side 0b1011 ; 4 Harmadik bájt kiküldése a szegmensekre, GPIO10 LOW (aktiválás)
nop                      ; Késleltetés
nop                      ; Késleltetés
nop side 0b1111          ; GPIO10 HIGH (deaktiválás)
out pins, 8 side 0b0111 ; 5. Negyedik bájt kiküldése a szegmensekre, GPIO11 LOW (aktiválás)
nop                      ; Késleltetés
nop                      ; Késleltetés
nop side 0b1111          ; GPIO11 HIGH (deaktiválás)
.wrap                    ; 6. A ciklustörzs végének jelölése
```

```
static const struct pio_program pio_7seg_program = { .instructions = pio_7seg_program_instructions,
    .length = 18,   .origin = -1,
};
static inline pio_sm_config pio_7seg_program_get_default_config(uint offset) {
    pio_sm_config c = pio_get_default_sm_config();
    sm_config_set_wrap(&c, offset + pio_7seg_wrap_target, offset + pio_7seg_wrap);
    sm_config_set_sideset(&c, 5, true, false);
    return c;
}
#include "hardware/pio.h"
#include "hardware/clocks.h"
#include "pio_7seg.pio.h"
PIO pio_7seg_init(PIO pio, uint sm, uint offset, uint base_pin) {  // A PIO program inicializálása
    pio_sm_config c = pio_7seg_program_get_default_config(offset); // A PIO állapotgép konfigurációja
    sm_config_set_out_pins(&c, base_pin, 8);           // GPIO 0-7 szegmensek kiosztása
    sm_config_set_sideset_pins(&c, base_pin + 8);      // GPIO 8-11 digitek
    uint clkdiv = clock_get_hz(clk_sys) / (1000 * 4); // Clock beállítás: 1 kHz multiplex frekvencia
    sm_config_set_clkdiv(&c, clkdiv);
    pio_gpio_init(pio, base_pin);                      // GPIO kivezetések inicializálása
    for (int i = 0; i < 12; i++) {
        pio_gpio_init(pio, base_pin + i);
        pio_sm_set_consecutive_pindirs(pio, sm, base_pin + i, 1, true);
        gpio_put(base_pin + i, (i > 7) ? 1 : 0);
    }
    pio_sm_init(pio, sm, offset, &c);                 // PIO állapotgép indítása
    pio_sm_set_enabled(pio, sm, true);
    return pio;
}
```

A PIO állapotgépet inicializáló függvények

pio_7seg.ino – 2/1.

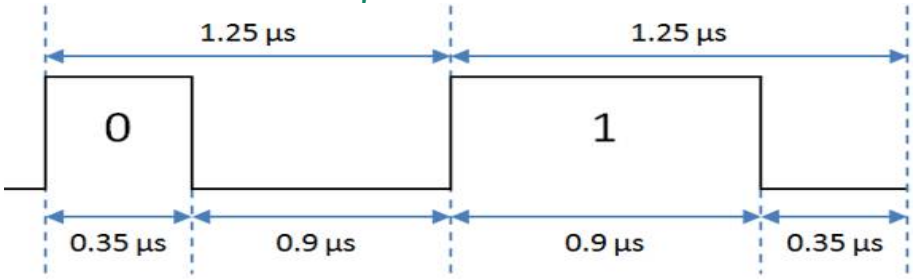
```
#include "hardware/pio.h"
#include "pio_7seg.pio.h" // Generált PIO fejlécfájl

// Konstansok
const uint base_pin = 0;  // GPIO 0 az első szegmensvezérlő
const uint sm = 0;        // PIO állapotgép index

uint counter = 0;         // Számláló kezdőértéke
PIO pio = pio0;           // Az első PIO példány használata

// Számok kódolása szegmensekre (a G, F, E, D, C, B, A sorrendben)
const uint8_t digit_segments[10] = {
    0b00111111, // 0
    0b00000110, // 1
    0b01011011, // 2
    0b01001111, // 3
    0b01100110, // 4
    0b01101101, // 5
    0b01111101, // 6
    0b00000111, // 7
    0b01111111, // 8
    0b01101111  // 9
};
```

pio_7seg.ino – 2/2.

```
// Szám megjelenítése a kijelzőn
void display_number(uint number) {
    uint32_t data = 0;
    // 4 digit kódolása (legkisebb helyiérték az első)
    for (int i = 0; i < 4; i++) {
        uint digit = number % 10;        // Szám kiszámítása
        number /= 10;                    // Következő számra lépünk
        data |= (digit_segments[digit] << (8 * i)); // Bájtok összeállítása
    }
    pio_sm_put_blocking(pio, sm, data); // Adat küldése a PIO FIFO-ba
}

void setup() {
    uint offset = pio_add_program(pio, &pio_7seg_program);
    pio_7seg_init(pio, sm, offset, base_pin);
}

void loop() {
    display_number(counter); // Szám megjelenítése
    counter = (counter + 1) % 10000;    // Szám növelése (max 9999)
    delay(100);

}
```

pio_ws2812.ino – 2/1. oldal

- Az **RP2040 Zero** kártya gyárilag tartalmaz egy WS2812 LED-et (GPIO16), ehhez készítünk programot
- A **WS2812** egyedi protokollja időzítési alapú, egyetlen adatvezetéken keresztül kommunikál. Egy 24 bites adat (3x8 bit a zöld, piros, és kék komponensekhez) határozza meg a színt
- Az időzítéseket, s a 24 bites adatok kiküldését egy **PIO** állapotgépre bízzuk, s az adatokat a főprogramból a **FIFO**-n keresztül küldjük ki

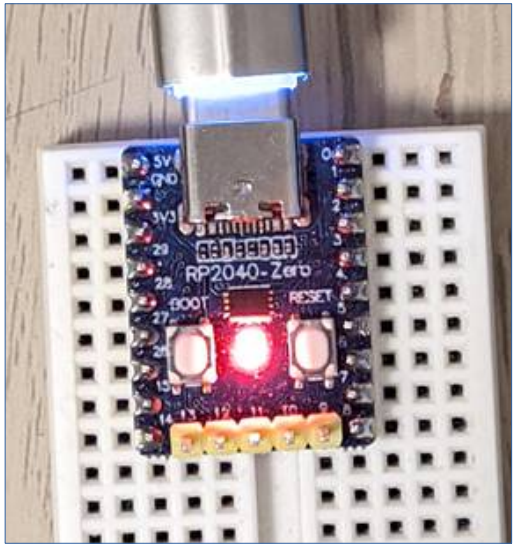



```
#include "hardware/pio.h"
#include "hardware/clocks.h"

#define LED_PIN 16

PIO pio = pio0;     // A PIO 0. modul s azon belül
uint sm = 0;        // a 0. állapotgépet használjuk

void setup() {
  uint offset = pio_add_program(pio, &ws2812_program);
  ws2812_program_init(pio,sm,offset,LED_PIN,800000);
}

void loop() {
  pio_sm_put_blocking(pio,sm,0x00FF0000); // Piros
  delay(1000);
  pio_sm_put_blocking(pio,sm,0xFF000000); // Zöld
  delay(1000);
  pio_sm_put_blocking(pio,sm, 0x0000FF00); // Kék
  delay(1000);
  pio_sm_put_blocking(pio,sm, 0x00000000); // LED ki
  delay(1000);
}
```

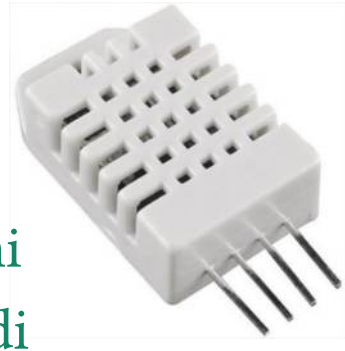
pio_ws2812.ino – 2/2. oldal

```
static const uint16_t instructions[] = {
  0x6221, // 0: out x,1   side 0 [2]
  0x1123, // 1: jmp !x,3  side 1 [1]
  0x1400, // 2: jmp 0     side 1 [4]
  0xa442, // 3: nop       side 0 [2]
};

pio_program_t ws2812_program = { .instructions = instructions, .length = 4, .origin = -1 };

void ws2812_program_init(PIO pio, uint sm, uint offset, uint pin, float freq) {
  pio_gpio_init(pio, pin);
  pio_sm_set_consecutive_pindirs(pio, sm, pin, 1, true);
  pio_sm_config c = pio_get_default_sm_config();
  sm_config_set_wrap(&c, offset, offset + ws2812_program.length - 1);
  sm_config_set_sideset(&c, 1, false, false);
  sm_config_set_sideset_pins(&c, pin);
  sm_config_set_out_shift(&c, false, true, 24); // 24 bites adatküldés
  sm_config_set_fifo_join(&c, PIO_FIFO_JOIN_TX);
  int cycles_per_bit = 10; // T1 + T2 + T3
  float div = clock_get_hz(clk_sys) / (freq * cycles_per_bit);
  sm_config_set_clkdiv(&c, div);
  pio_sm_init(pio, sm, offset, &c);
  pio_sm_set_enabled(pio, sm, true);
}
```

Az adatbeolvasás automatikus (AUTOPULL opció)
A programciklus szervezését a wrap beállítások biztosítják
Ez a négy utasítás elegendő egy-egy adatbit kiküldésére

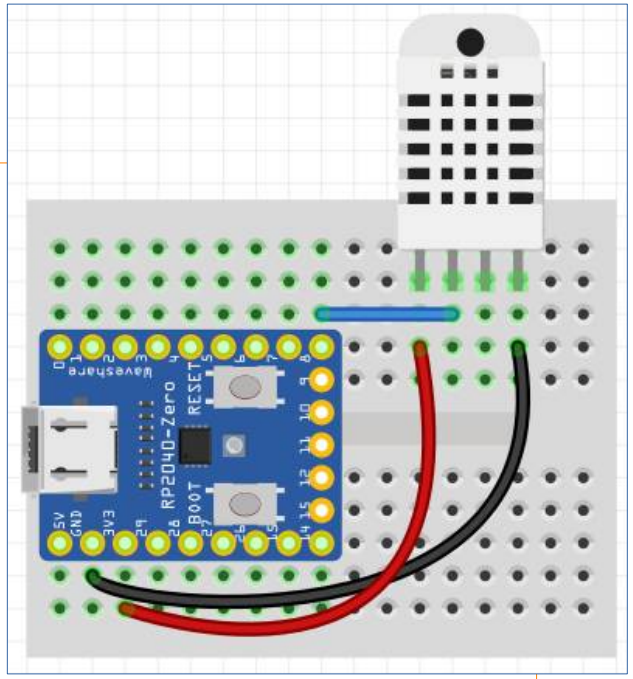


DHT szenzorok kiolvasása PIO állapotgéppel



- A **DHT** szenzorok, mint például a **DHT22,** kiolvasása kihívást jelenthet, mivel az egyvezetékes protokolljuk egyedi jellege miatt az MCU-kban nincsen hozzájuk illeszkedő hardveres periféria, így ezek kezeléséhez szoftveres időzítésekkel kell dolgozni
- Az **RP2040 PIO** modulja ebben segít, mivel lehetővé teszi az egyedi protokoll hardveres megvalósítását, csökkentve ezáltal a CPU terhelését
- **Valentin Milea** pont egy ilyen programkönyvtárat tett közzé a **GitHub**-on, DHT sensor library for the Raspberry Pi Pico néven (átneveztük **pio_dht**-re)
- A fenti programkönyvtár egy C mintaprogramot is tartalmaz, azt adaptáltuk az általunk használt Arduino MbedOS RP2040 Boards környezethez, illetve kihagytunk belőle 1-2 lényegtelen dolgot (pl. a számunkra érdektelen Celsius – Fahrenheit átalakító függvényt). A program használatához telepítenünk kell a **pio_dht** könyvtárat az Arduino ***libraries*** mappájába

pio_dht.ino

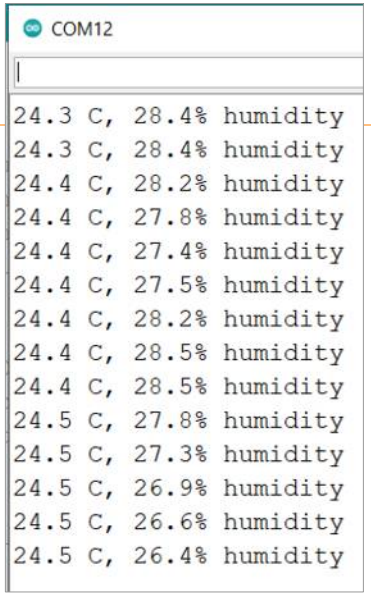



```
#include <pio_dht.h>
static const dht_model_t DHT_MODEL = DHT22;
static const uint DATA_PIN = 8;

dht_t dht;

void setup() {
  Serial.begin(115200);
  dht_init(&dht, DHT_MODEL, pio0, DATA_PIN, true /* pull_up */);
}

void loop() {
  dht_start_measurement(&dht);
  float humidity, temperature_c;
  dht_result_t result = dht_finish_measurement_blocking(&dht, &humidity, &temperature_c);
  if (result == DHT_RESULT_OK) {
    Serial.print(temperature_c, 1); Serial.print(" C, ");
    Serial.print(humidity, 1);      Serial.println("% humidity");
  } else if (result == DHT_RESULT_TIMEOUT) {
    Serial.println("DHT sensor not responding. Please check your wiring.");
  } else { assert(result == DHT_RESULT_BAD_CHECKSUM);
    Serial.println("Bad checksum"); }
  delay(2000); // 2 másodperces mintavétel
}
```

pio_dht.pio

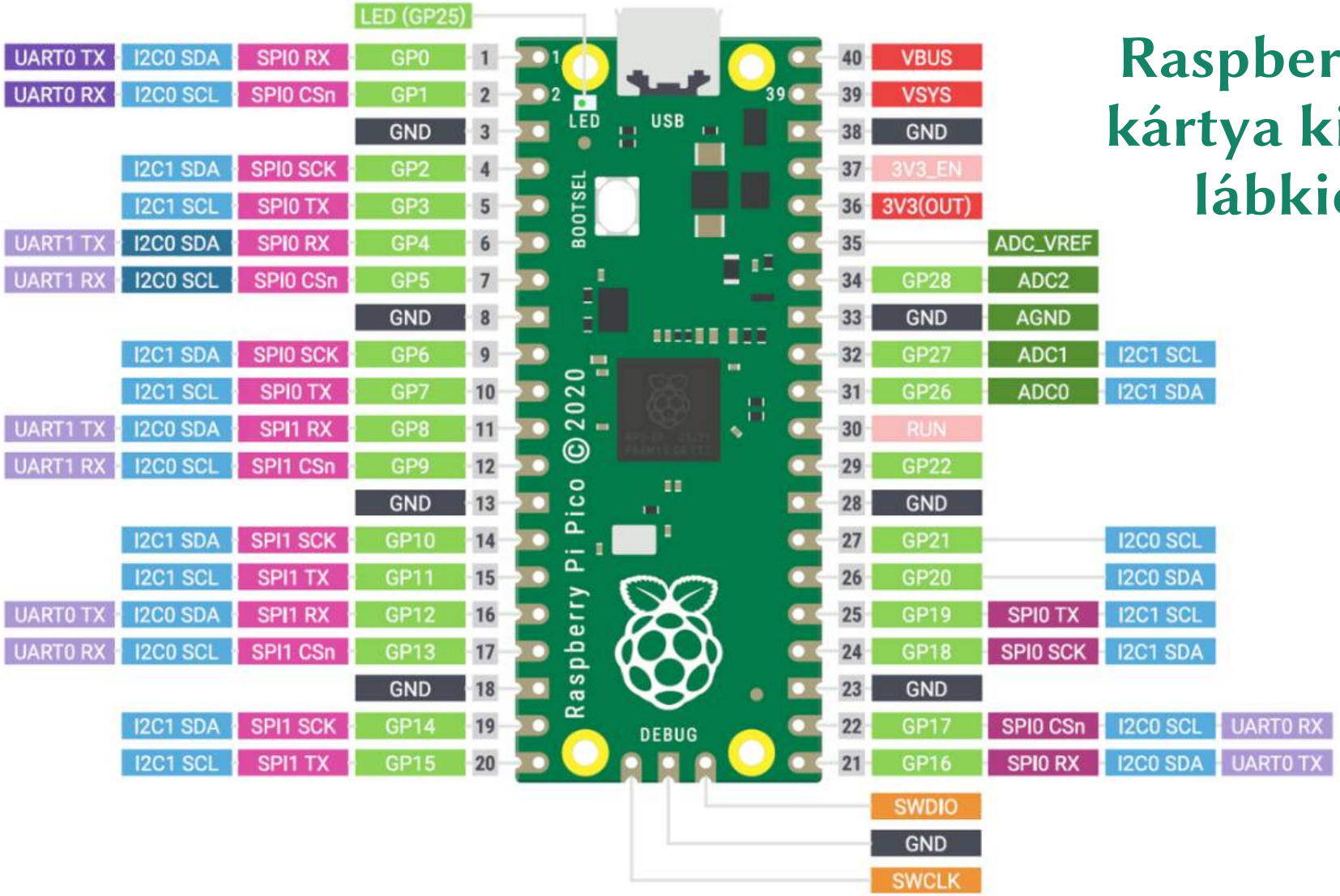
```
.program dht

.define public start_signal_clocks_per_loop      1
.define public pulse_measurement_clocks_per_loop 2


loop_until_start_signal_done:
    jmp y-- loop_until_start_signal_done
    ; back to hi-z, DHT sensor will drive the
signal
    set pindirs 0

    ; wait until DHT sensor is ready
loop_until_ready_lo:
    jmp pin loop_until_ready_lo

loop_until_ready_hi:
    jmp pin loop_until_lo
    jmp loop_until_ready_hi

    ; process DHT payload
loop_until_lo:
    jmp pin loop_until_lo
```

```
loop_until_hi:
    jmp pin pulse_loop_init
    jmp loop_until_hi

    ; measure pulse duration
pulse_loop_init:
    ; reset counter
    mov y, osr
pulse_loop:
    jmp pin pulse_tick
    jmp short_pulse_detected
pulse_tick:
    jmp y-- pulse_loop
    ; counter has reached zero, long pulse detected
    ; shift in 1 bit
    set x, 1
    in x, 1
    jmp loop_until_lo

short_pulse_detected:
    ; pin was driven low, short pulse detected
    ; shift in 0 bit
    in null, 1
    jmp loop_until_hi
```

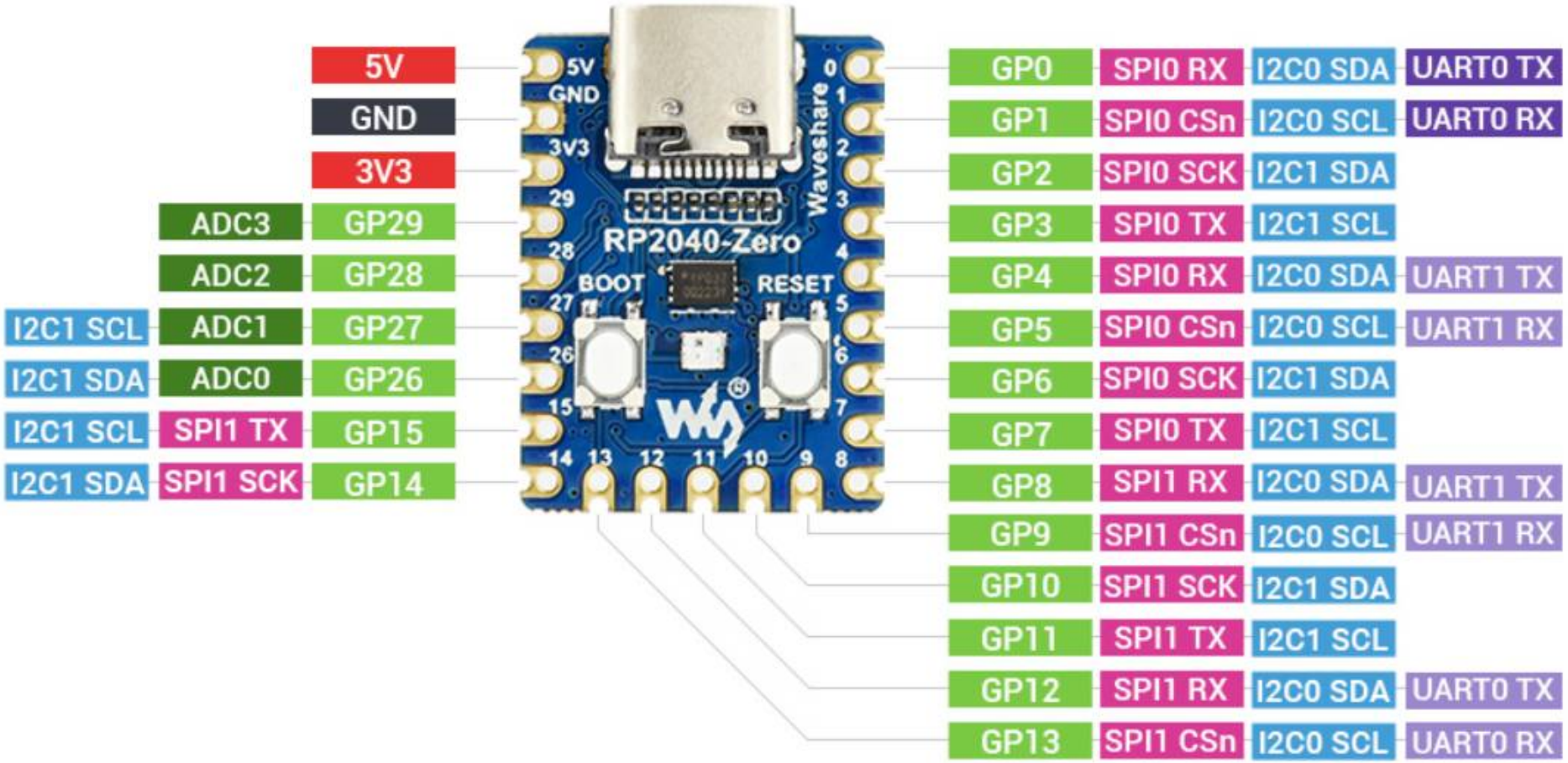


Forrás: Valentin Milea: DHT sensor library for the Raspberry Pi Pico

Raspberry Pi Pico kártya kivezetések lábkiosztása



A kivezetések lábkiosztása



További kivezetések

- Néhány kivezetés a kártya hátoldalán van kivezetve