STM32 mikrovezérlők programozása ARM mbed környezetben

```
#include "mbed.h"

DigitalOut myled(LED1);     // PA_5 alias LED1

int main()
{
    while(1) {
        myled = 1; // LED is ON
        wait(0.2); // 200 ms
        myled = 0; // LED is OFF
        wait(1.0); // 1 sec
    }
}
```

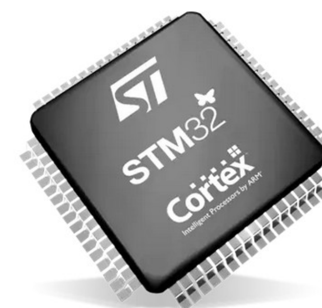
7. RTOS alapok

Felhasznált és ajánlott irodalom

- Cserny István: **A FRDM-KL25Z kártya programozása mbed környezetben**
- Rob Toulson, Tim Wilmhurst:
  Fast and Effective Embedded Systems Design: Applying the ARM mbed
- Perry Xiao: Designing Embedded Systems and the IoT eith ARM mbed
- Dogan Ibrahim: ARM-based Microcontroller Projects Using mbed
- Miro Samek: **Practical UML Statecharts in C/C++**
- **ARM mbed honlap: https://os.mbed.com/**
  - ARM mbed Compiler: https://ide.mbed.com/compiler/
  - ARM mbed 2 Handbook: **https://os.mbed.com/handbook/RTOS**
  - ARM mbed forráskód: https://github.com/ARMmbed/mbed-os

Adatlapok:

- **STM32F446RE** adatlap és termékinfo
- **STM32F446** Family Reference Manual



Példaprogramok

STM32_RTOS – egyszerű demó két LED-del

RTOS_display – négyjegyű, hétszegmens kijelző

RTOS_queue – adatküldés programszálak között

RTOS_mutex – kölcsönös kizárás használata

RTOS_philosophers – az étkező filozófusok problémája

A mintaprogramok az os.mbed.com/users/cspista/code/ oldalon is megtalálhatók

A többfeladatos rendszerek kihívásai

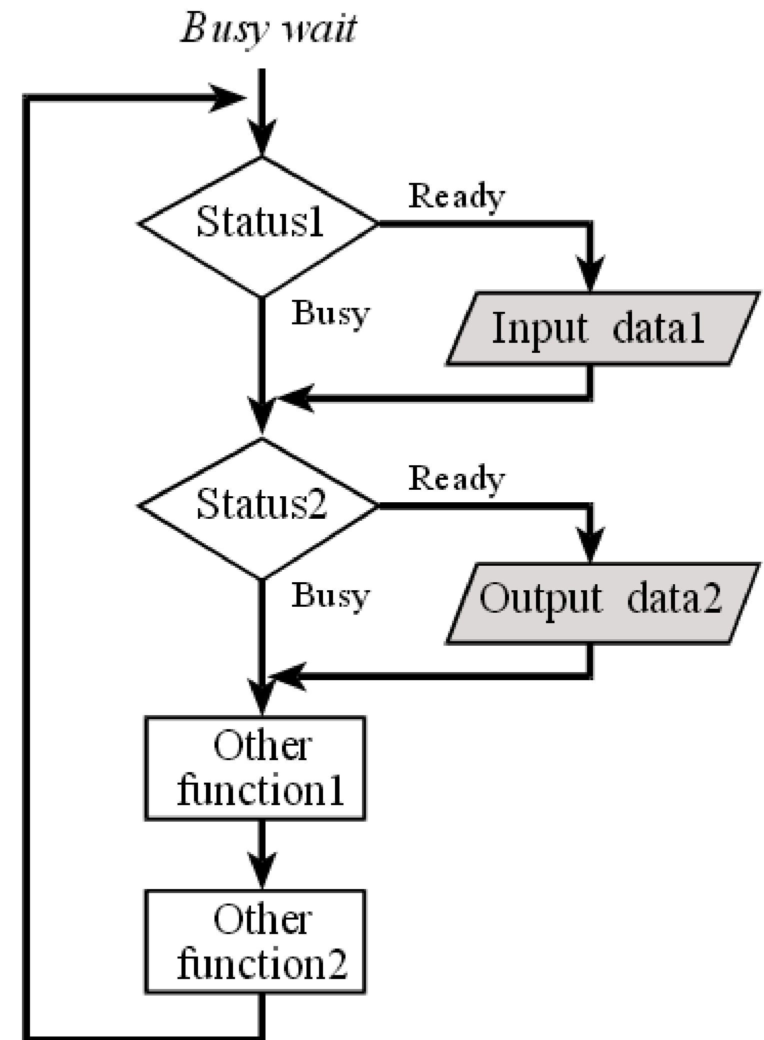
- Egy számítógépes program általában egyszerre csak egy dologgal foglalkozik: beolvas egy állományt, számolást végez a beolvasott adatokkal, végül kiírja az eredményeket.
- Egy beágyazott rendszer számos dolgot kell, hogy végezzen egyidejűleg: pl. leolvas egy hőmérőt, működtet egy szelepet, gombnyomásokra figyel, ellenőrzi az eltelt időt, számlálja az elhaladó üvegeket egy palackozóban stb. Ez a többfeladatos (multitasking) programozás lényege, hogy (látszólag) egyidejűleg több feladattal is tudunk foglalkozni
- Egy mikrovezérlő program természetesen egyszerre csak egy dologgal tud foglalkozni. De ha a feladatok között gyorsan váltogatva fut a program, akkor úgy tűnik, mintha egyidejűleg végezne több dolgot
- Hogyan írhatunk ilyen programot, ami megosztja a figyelmét több feladat között – lehetőleg úgy, hogy ne zavarodjunk bele sem mi, sem a mikrovezérlő a több feladat közötti váltogatás bonyodalmaiba?
  A következőkben felsorolunk néhány elterjedt módszert:

Szuperhurok

- **A szuperhurok** egy végtelen ciklusból álló programszerkezet, amely az összes végrehajtandó feladatot tartalmazza

A szuperhurok módszer jellemzői:

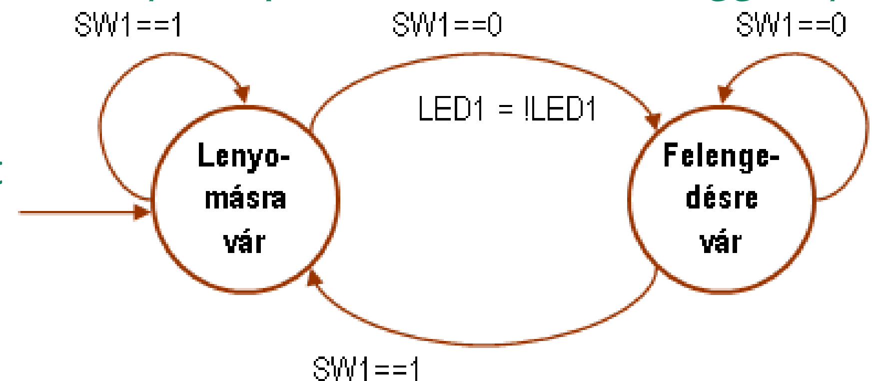
- **Hagyományos technológia**, egyszerű alkalmazásokra jól megfelel
- A program **fix sorrendű funkciók** egymás utáni sorozata
- **Időkritikus feladatok** kiszolgálása **csak megszakításokkal** valósítható meg
- **Időzítések implementálása** nehézkes, a változtatások kihatnak az egész hurokra



Véges állapotgép módszer

- **A véges állapotgép** (Finite State Machine) egy absztrakció, amely úgy írja le a rendszer viselkedését, mint egy automatát, amely a bemenetek hatására váltogatja az állapotait (a módszer célje a modell viselkedését a programunkban realizálni)
- Egy véges állapotgép az alábbi öt alapvető dologgal jellemezhető:
  - **A lehetséges állapotok** véges halmaza, melyek egyike a kiindulási állapot
  - **A bejövő adatok** véges halmaza.
  - A rendszer által generált **kimenő adatok** véges halmaza
  - **Az állapotok közötti átmenetek,** mint a pillanatnyi állapot és a bemenetek függvénye
  - **A kimenő adatok előállításának szabályai** (pillanatnyi állapot és a bemenetek függvénye)

Példa: LED ki-be kapcsolgatása nyomógombbal. Az állapotgép megadható szövegesen, irányított gráffal vagy táblázat formájában. Itt a nyomógomb állapotait kell modellezni.
A gyakorlatban az alkalmazások természetesen több és bonyolultabb állapotgépek együtteséből állnak...



Pillanatnyi állapot	Következő állapot	Állapotváltás feltétele	Tevékenység
Lenyomásra vár	Felengedésre vár	SW1 == 0	LED1 = !LED1
Felengedésre vár	Lenyomásra vár	SW1 == 1	

Lab01_button_ledswitch/main.cpp

- A 2021. szeptember 23-i előadásban az előző oldalon bemutatott állapotgépet így implementáltuk (másképp is lehetett volna...)

```
#include "mbed.h"

DigitalOut myled(LED1);                  // PA_5 alias LED1
DigitalIn mybutton(BUTTON1,PullUp);      // PC13 alias BUTTON1
int bState, waitforpress=1, led_state=0;
int main() {
    while(1) {
        bState = mybutton;
        if(waitforpress) {               //Ha lenyomásra várunk és
            if(!bState) {                //Ha lenyomás történt...
                led_state = !led_state;  //LED állapotának átbillentése
                myled = led_state;
                waitforpress = false;    //Következő stáció: felengedésre várunk
            }
        } else {                         //Ha felengedésre vártunk és
            if(bState) {                 //Ha felengedést észlelünk...
                waitforpress = true;     //Következő stáció: lenyomásra várunk
            }
        }
        wait(0.02);                      // 20 ms pergésmentesítő késleltetés
    }
}
```

Körkörös ütemezés, kooperatív rendszer

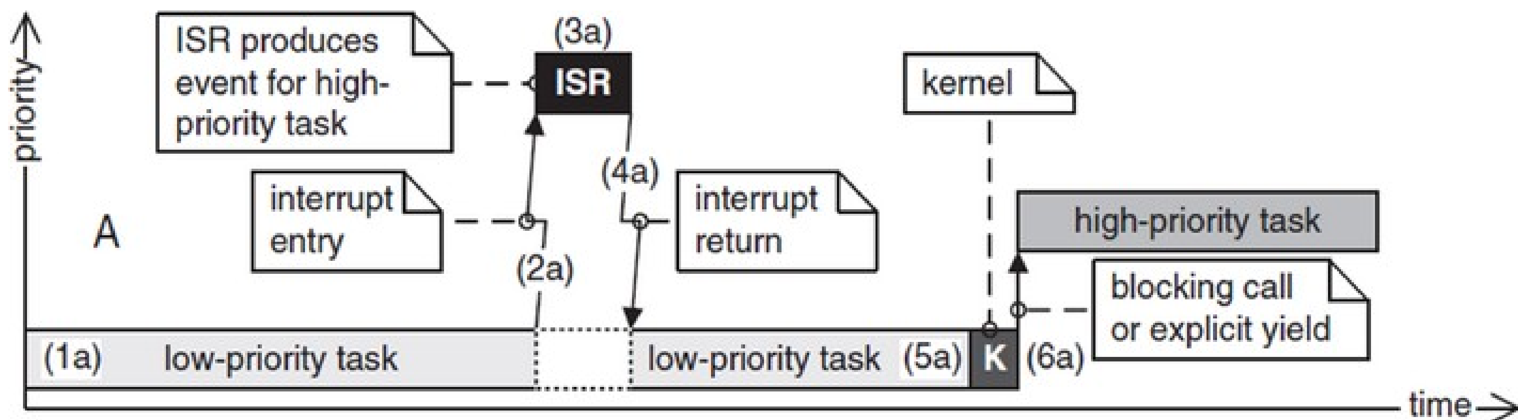
- Az egyszerű szuperhurok módszert átalakíthatjuk úgy, hogy a főprogramban csak definiáljuk a feladatokat (task), majd végül átadjuk a vezérlést egy ütemezőnek, mely legegyszerűbb esetben **körkörös ütemezést** (round robin scheduling) valósít meg
Az ütemező szokásos elnevezései: scheduler, kernel, dispatcher
- Mit nyerünk ezzel?
  - Az ütemező megírható újrafelhasználható kódként
  - Az ütemező továbbfejleszthető különféle szolgáltatásokkal, vagy ütemezési stratégiákkal, megtéve ezzel az első lépést az operációs rendszerek felé
- Ha például a körkörös ütemező megjegyzi, hogy az egyes feladatok hol tartottak, amikor visszaadták a vezérlést az ütemezőnek, valamint lehetőség ad arra, hogy a feladatok megadhassák, hogy mennyi idő múlva akarják folytatni a tevékenységüket, akkor eljutottunk a **kooperatív többfeladatos rendszerekhez**

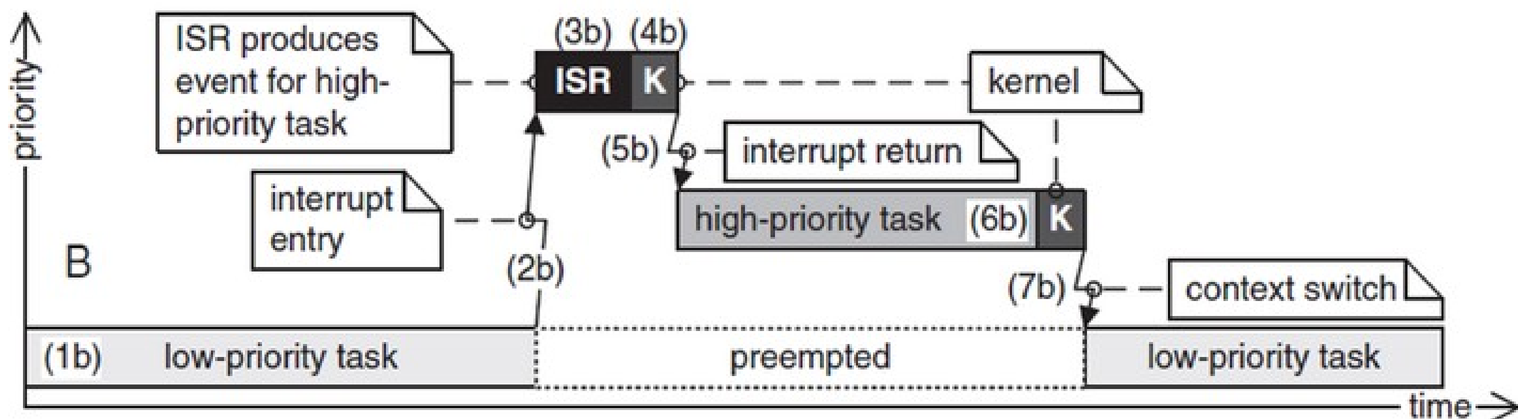
Megszakításos ütemezés

- **A prioritási szintek** bevezetése és a **megszakításos ütemezés** (preemptive scheduling) a következő lépés a valós idejű operációs rendszerek (**Real-Time Operating System, RTOS**) megvalósítása felé
- **A programszálak megszakíthatósága** azt jelenti, hogy ha egy magasabb prioritású programszál futtatható állapotba került (egy esemény bekövetkezte miatt), akkor az éppen futó, alacsonyabb prioritású programszál futása félbeszakad, s csak akkor folytatódik, amikor a magasabb prioritású programszál már befejezte a tevékenységét (újabb eseményre vagy időzítésre várakozó állapotba kerül)
- **A megszakíthatóság** azt is jelenti, hogy - a kooperatív működéssel szemben - az ütemező akkor is megszakíthatja az éppen futó programszálat, ha egy vele azonos prioritású feladat már régóta futásra vár.
- Az azonos prioritású programszálak **időosztásos alapon** (time sharing), körkörös ütemezéssel (round robin scheduling) futnak. Az **RTOS** kezdeti konfigurálásától függ, hogy egy-egy időszelet milyen időtartamú (általában 5 - 50 ms).

Kooperatív vs. megszakításos ütemezés

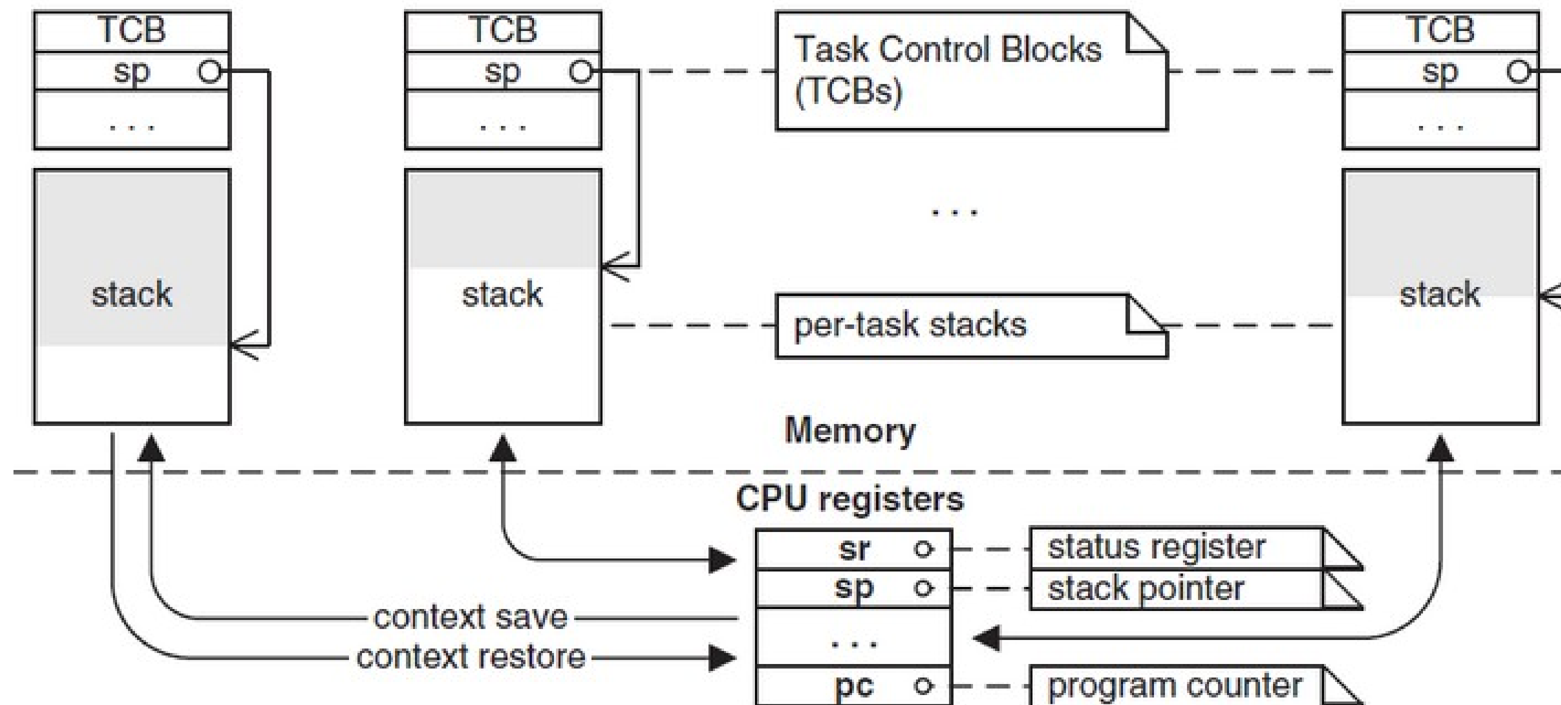
- Kooperatív esetben a magasabb prioritású feladat csak akkor kerül sorra, amikor az alacsony prioritású szál átadja a vezérlést
- A második példában az ütemező félbeszakítja az alacsony prioritású task futását, amikor a magasabb prioritás task futásra kész állapotba kerül





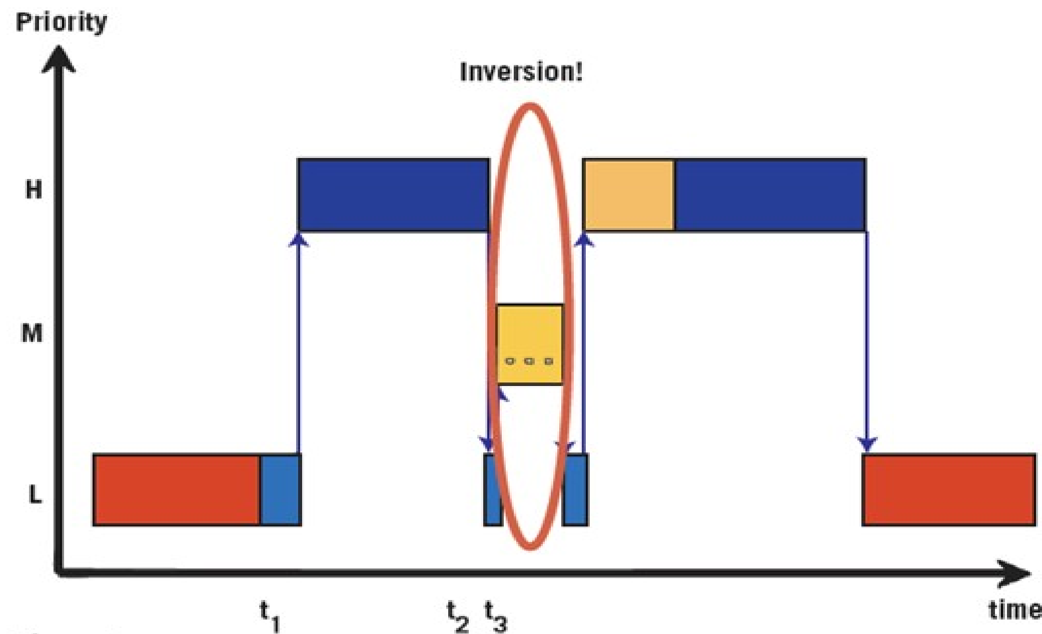
A megszakíthatóság következményei

- A programszálak megszakíthatósága miatt - mivel az bármikor bekövetkezhet - szükséges, hogy a programszálak folytatásához minden szükséges információt elmentsünk (CPU regiszterek, visszatérési cím, lokális változók). Minden programszál saját veremterülettel és a taszkleíró táblában egy-egy bejegyzéssel kell, hogy rendelkezzen (ez utóbbi az ütemezőnek kell, hogy a programszálak állapotát nyilvántartsa)



A prioritás inverzió problémája

- A programszálak a közös használatú erőforrásokat ideiglenesen lefoglalhatják. Ez mellékhatásokhoz vezethet: pl. prioritás inverzió



1) Az alacsony prioritású L programszál megszerez egy erőforrást (pl mutex)
2) A magas prioritású H programszál félbeszakítja L futását (t1)
3) A H programszál futása megszakad, ha az L által lefoglalt erőforrásra kell várnia (t2)
4) Ha egy közepes prioritású M programszál ekkor megszerzi a futási jogot, akkor a prioritás megfordul, mert bármeddig futhat, blokkolva H és L működését.

- **Megoldás:** az erőforrást birtokló L programszál prioritásának ideiglenes megemelése

Valós idejű rendszerek jellemzői

- **Hard RTOS:** követelmény a feladatokhoz tartozó határidők szigorú betartása, elmulasztásuk katasztrófához vezethet
- **Firm RTOS:** ezeknél szintén követelmény a határidők betartása, s bár lekésésük nem vezet katasztrófához, de olyan nemkívánatos következménnyel járhat, ami lerontja a termék minőségét
- **Soft RTOS:** Ennél a típusú RTOS-oknál megengedett az időkorlátok alkalmankénti túllépése
- **Egyensúlyt kell teremteni** a gazdag funkcionalitás, a kritikus határidők betartása és a kiszámíthatóság megőrzése között, az alábbi szempontok szerint:
  - A programszálak közötti váltás (*context switching*) lehetőleg gyors legyen
  - Az "interrupt latency" minél rövidebb legyen (ne tiltsuk le a megszakítást)
  - Megszakítás végén az ütemező értékelje ki a helyzetet, s a legmagasabb prioritású programszálnak adja át a vezérlést – ez a *„dispatch”*
  - Megbízható és időhöz kötött mechanizmusokat kell bevezetni a folyamatok egymás közötti kommunikációjának megvalósításához
  - Az RTOS támogassa a többfeladatú programfutást (multitasking) és a megszakításos ütemezést (preemptive scheduling)

mbed-RTOS

- **Az mbed programkönyvtár** 2012 óta **RTOS támogatást** is nyújt, amely a CMSIS RTOS API-n és a nyílt forrásúvá tett **ARM/Keil RTX** valósidejű operációs rendszeren alapul
- Egyszerű példa, amelyben két LED-et más-más frekvenciával villogtatunk
- A program elején az **rtos.h** fejléc állományt is be kell csatolni. A main függvény az első programszál, emellett egy másik programszálat is indítunk (Thread objektumok).
- A korábban használt **wait()** blokkoló várakozások helyett RTOS környezetben a **Thread::wait()** tagfüggvényt kell használni, amely nem blokkolja a többi programszál futását

Lab07_STM32_RTOS/main.cpp

```
#include "mbed.h"
#include "rtos.h"

DigitalOut led1(D13);
DigitalOut led2(D12);
Thread thread;

void led2_thread() {
    while (true) {
        led2 = !led2;
        Thread::wait(1000);
    }
}

int main() {
    thread.start(led2_thread);

    while (true) {
        led1 = !led1;
        Thread::wait(500);
    }
}
```

Programszálak (threads)

- A programszál (thread) önálló végrehajtási egységként működő objektum, szekvenciálisan végrehajtható utasítás-sorozat
- Az első programszál mindig a **main()** függvény, ezt nem kell **Thread** objektumként példányosítani
- Programszálakat a **Thread** objektumosztály példányosításával hozhatunk létre. Csak az első paraméter megadása kötelező, a többi paraméternek van alapértelmezett értéke

```
Thread mythread(void(*task)(void const *argument),  // Függvénymutató
                              void *argument=NULL,  // Átadandó adat mutatója
             osPriority priority=osPriorityNormal,  // Programszál prioritása
           uint32_t stack_size=DEFAULT_STACK_SIZE,  // Veremtár mérete
                unsigned char *stack_pointer=NULL   // Veremtár mutató
```

A Thread konstruktor paraméterei

- **A konstruktor első paramétere** egy speciális függvénymutató. Az a függvény, amire rámutatunk, adja meg a programszál végrehajtandó utasítás-sorozatát, s formailag az alábbi feltételeket kell, hogy kielégítse:
  - A függvény végtelen ciklust tartalmaz, vagyis sohasem "térhet vissza".
  - A függvény **void** típusú, azaz nincs visszatérési értéke.
  - A függvény egyetlen paramétere **void const *arg**, azaz típus nélküli mutató
- **A konstruktor második paramétere** egy típus nélküli mutató, amelyet át akarunk adni a programszálnak. Ez többnyire egy struktúrára vagy tömbre mutató pointer, melynek értelmezése és felhasználása a programozó kompetenciájába tartozik. Ennek a paraméternek főleg akkor van szerepe, ha ugyanabból a függvényből több programszálat is létre akarunk hozni. Más esetekben ez a paraméter elhagyható, ekkor **az alapértelmezett NULL érték** adódik át

A Thread konstruktor paraméterei

- **A konstruktor harmadik paramétere** a programszál futási prioritását adja meg (alapértelmezetten **osPriorityNormal**)
- A programszálakhoz rendelhető prioritás értékeket az alábbi táblázatban foglaltuk össze:

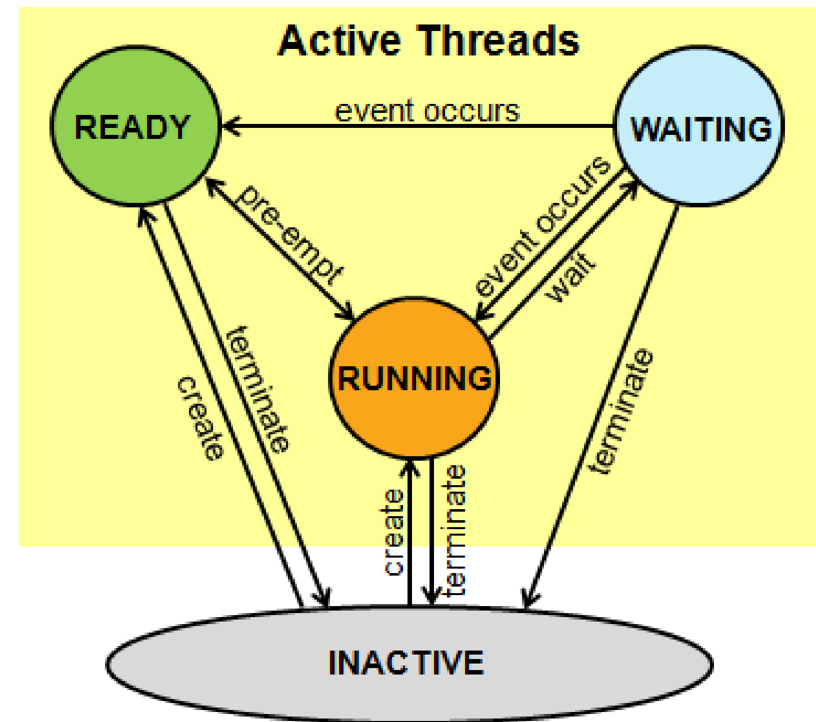
Szimbólum	Érték	Prioritás
osPriorityIdle	-3	Tétlen (legalacsonybb prioritás)
osPriorityLow	-2	Alacsony
osPriorityBelowNormal	-1	Normál alatti
osPriorityNormal	0	Normál
osPriorityAboveNormal	1	Normál fölötti
osPriorityHigh	2	Magas
osPriorityRealTime	3	Valósidejű (legmagasabb prioritás)
osPriorityError	0x84	Nem meghatározott, vagy érvénytelen prioritás

- **A konstruktor negyedik és ötödik paraméterével** a programtárhoz rendelt veremtár méretét és helyét szabhatjuk át

A programszál lehetséges állapotai

- Egy **Thread** objektum az alábbi állapotok valamelyikében lehet:
- **RUNNING:** A programszál pillanatnyilag futó állapotban van. Egyszerre csak egyetlen programszál lehet ebbe az állapotban.
- **READY:** A programszál futásra kész állapotban van (egyidejűleg több programszál is lehet ebben az állapotban)

Amint az éppen futó programszál abbahagyja a tevékenységét, az ütemező a **READY** állapotú programszálak közül a legmagasabb prioritásúnak adja át a vezérlést (ez lesz az új **RUNNING** állapotú programszál).



- **WAITING:** A programszál valamilyen eseményre várakozik.
- **INACTIVE:** Nem létrehozott, vagy leállított programszál(ak).

A Thread osztály publikus tagfüggvényei

A függvény neve	Funkciója
`terminate()`	Bezárja a programszál futását és kiveszi az aktív feladatok közül
`set_priority(pri)`	Beállítja/módosítja a programszál futási prioritását
`get_priority()`	Lekérdezi az aktuális programszál prioritását
`signal_set(signals)`	Beállítja az aktuális programszál megadott jelzőbitjeit
`signal_clr(signals)`	Törli az aktuális programszál megadott jelzőbitjeit
`get_state()`	Lekérdezi az aktuális programszál állapotát
`stack_size()`	Lekérdezi a programszál rendelkezésére álló veremtár méretét
`free_stack()`	Lekérdezi a programszál még szabad veremtárterületének méretét
`used_stack()`	Lekérdezi a programszál által felhasznált veremtárterület méretét
`max_stack()`	Lekérdezi a programszál futása során felhasznált max. veremtárterületet

- A **Thread** osztály **publikus tagfüggvényei** az objektumpéldányhoz rendeltek, így az objektumpéldány metódusaként hívhatók meg, mint pl. a **set_priority()** tagfüggvény az alábbi példában

```
Thread thread2(music);                    // Define a new task
thread2.set_priority(osPriorityHigh);     // Give it high priority
```

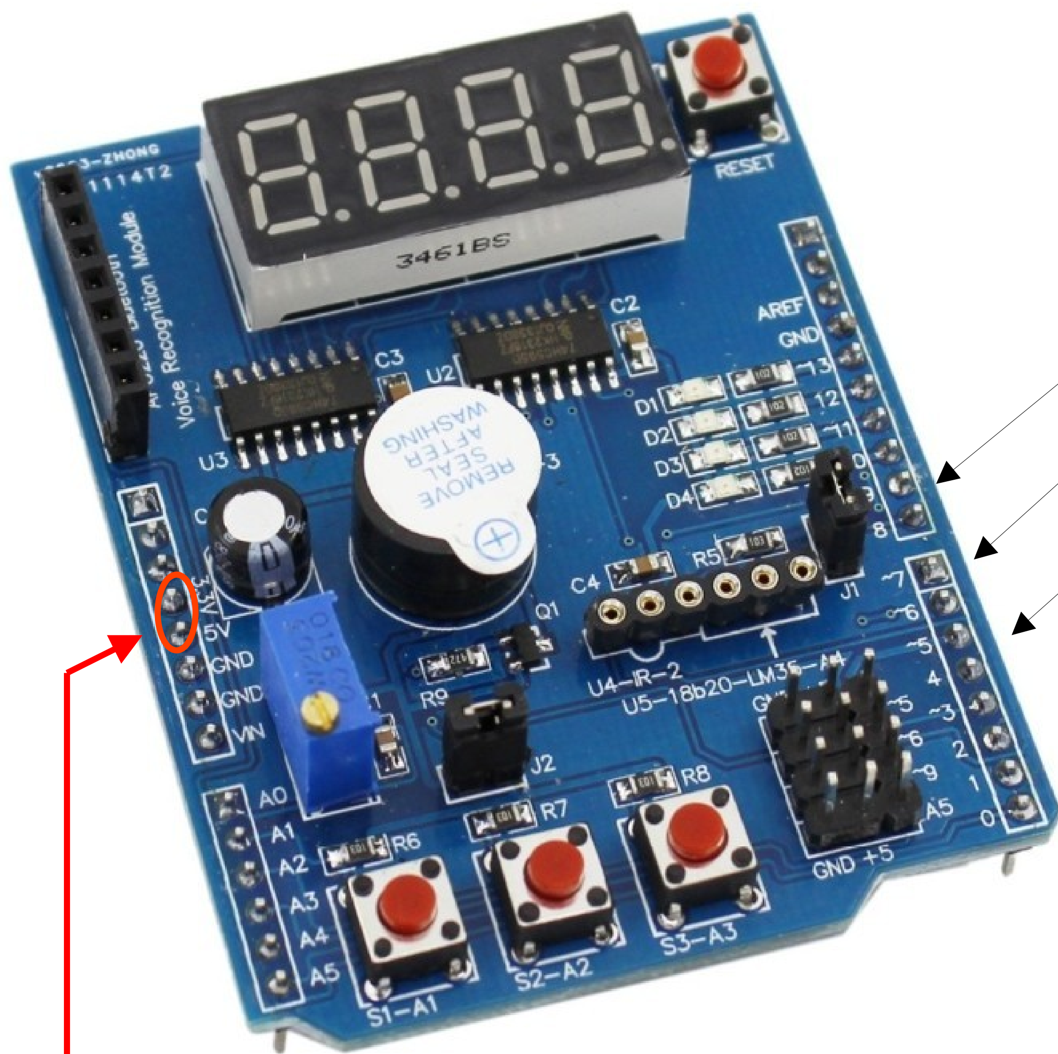
A Thread osztály statikus tagfüggvényei

- A **statikus publikus tagfüggvények** az osztályhoz tartoznak, nem objektumpéldányhoz. Meghívásuk az osztály neve és a **scope** operátor segítségével történhet, mint az alábbi példában:

```
Thread::wait(1000);                         //Wait for 1000 msec
```

A függvény neve	Funkciója
`signal_wait(signals,time)`	Várakozás indítása egy vagy több eseményjelzőre vonatkozóan. Az opcionális **time** paraméter a maximális várakozási időt adja meg ezredmásodpercekben, alapértelmezés: végtelen várakozás
`wait(time)`	Várakozás ezredmásodpercekben megadott ideig
`yield()`	Átadja a vezérlést a következő READY állapotú programszálnak.
`gettid()`	Lekérdezi az éppen futó programszál azonosító számát
`attach_idle_hook()`	A tétlen task által meghívandó függvény hozzárendelése
`attach_terminate_hook()`	A task leállításakor meghívandó függvény hozzárendelése

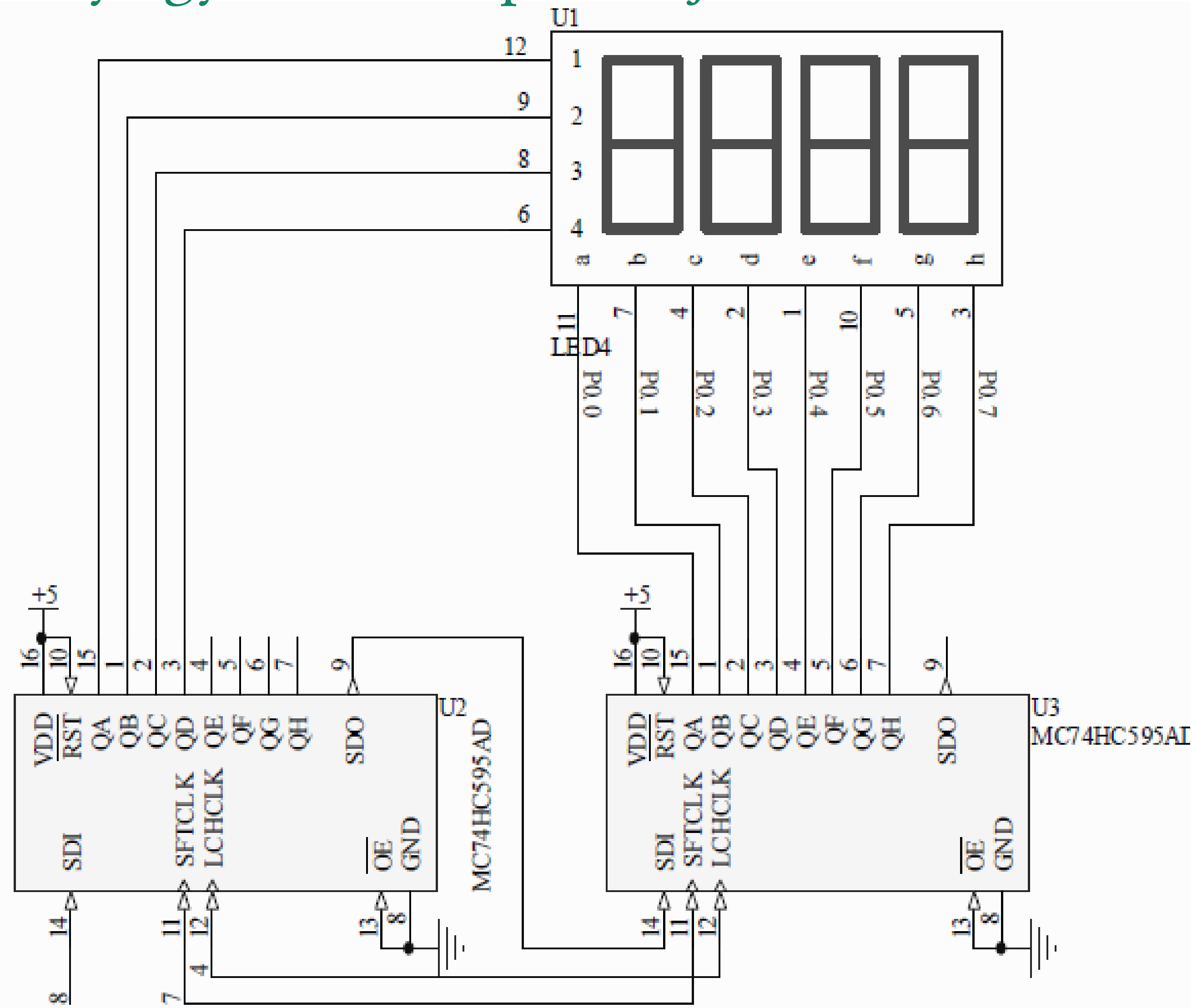
A multifunkciós kártya számkijelzője



- **A bekötés:**
  - **SDA** (data) – **D8**
  - **SCK** (clock) – **D7**
  - **~CS** (latch) – **D4**
- Technikai okokból célszerű a kártyát 3,3 V-ossá alakítani:
  - Az 5V-os tüskét eltávolítjuk
  - Az 5V-os és a 3,3 V-os jelzésű pontokat közösítjük

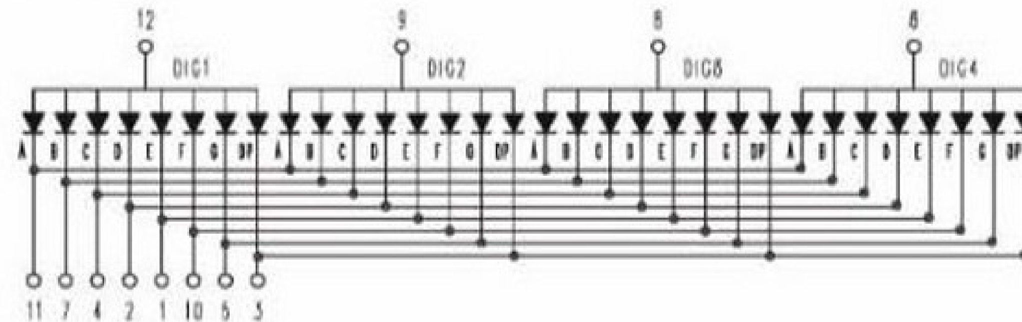
A multifunkciós kártya számkijelzője

- A multifunkciós bővítőkártya egy 4 számjegyű, hétszegmenses kijelzőt is tartalmaz, amely egy **3461BS** típusú kijelzőből és két **74HC595** léptető regiszterből áll
- Az első regiszter a 7 szegmens + DP LED katódokat vezérli (negatív logika)
- A második regiszter alsó négy bitje a négy számjegy kiválasztására szolgál (a közösített anódokat táplálja)
- Bitsorrend: `MSBFIRST`

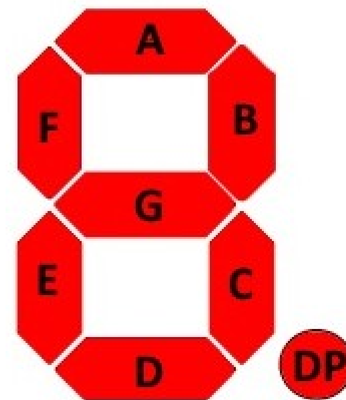


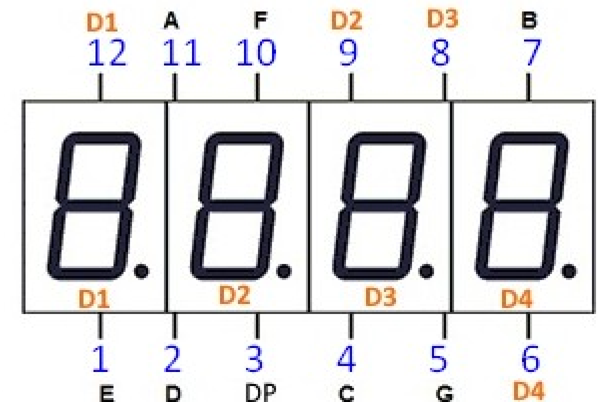
Négyszámjegyű LED kijelző

- A **3461B** típusú 7-szegmenses 4-számjegyű kijelző szegmens-vezérlő vonalai közösítve vannak



- A 8. szegmens a tizedespont
- A szegmensek közösítése miatt csak multiplex (időosztásos) vezérlés használható: egyidejűleg csak egy számjegy jeleníthető meg
- A számjegyválasztó vezetékek egy-egy számjegy LED-jei közösített anódjainak kivezetései
- A **3461B** típusú kijelző vezérlése 8 + 4 kivezetést igényel, ezért célszerű soros perifériabővítővel kombinálni (I2C vagy SPI)





A ShiftOut programkönyvtár

- Ollie Milton ShiftOut mbed könyvtára csupán a ***ShiftOut.h*** fejléc állományból áll, s a soros adatküldést szoftveresen (**DigitalOut** objektumok állítgatásával) végzi
- **ShiftOut***(clkpin, datapin, latchpin, registerCount=0x08)* – a konstruktor, ahol az opcionális *registerCount* paraméter a bitszámot jelenti
- **write**(int data) – ez a tagfüggvény kilépteti az adatot (MSBfirst)
- Egyszerű példa: egy nulla megjelenítése a Multifunkciós kártya kijelzőjének bal szélső pozícióban

```
#include "mbed.h"
#include "ShiftOut.h"

ShiftOut display(D7, D8, D4);    // clk=D7, data=D8, latch=D4)

display.write(0xC0);             // Nullát rajzol
display.write(0x01);             // A bal szélső számjegyet választja ki
```

RTOS_display – multiplex kijelzéssel

```
#include "mbed.h"
#include "rtos.h"
#include "ShiftOut.h"

uint8_t segment_data[4] = {0xFF, 0xFF, 0xFF, 0xFF};
const uint8_t SEGMENT_MAP[]= {0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90};
const uint8_t SEGMENT_SELECT[] = {0x01, 0x02, 0x04, 0x08};

Thread thread;
DigitalOut buzzer(D3); // 0 aktiválja!

void led2_thread() {
  ShiftOut display(D7, D8, D4);
  while (true) {
    for(int i = 0; i<4; i++) {
      display.write(segment_data[i]);
      display.write(SEGMENT_SELECT[i]);
      Thread::wait(2);
    }
  }
}
```

Szegmens adatok

Számjegyek inverz szegmensképe

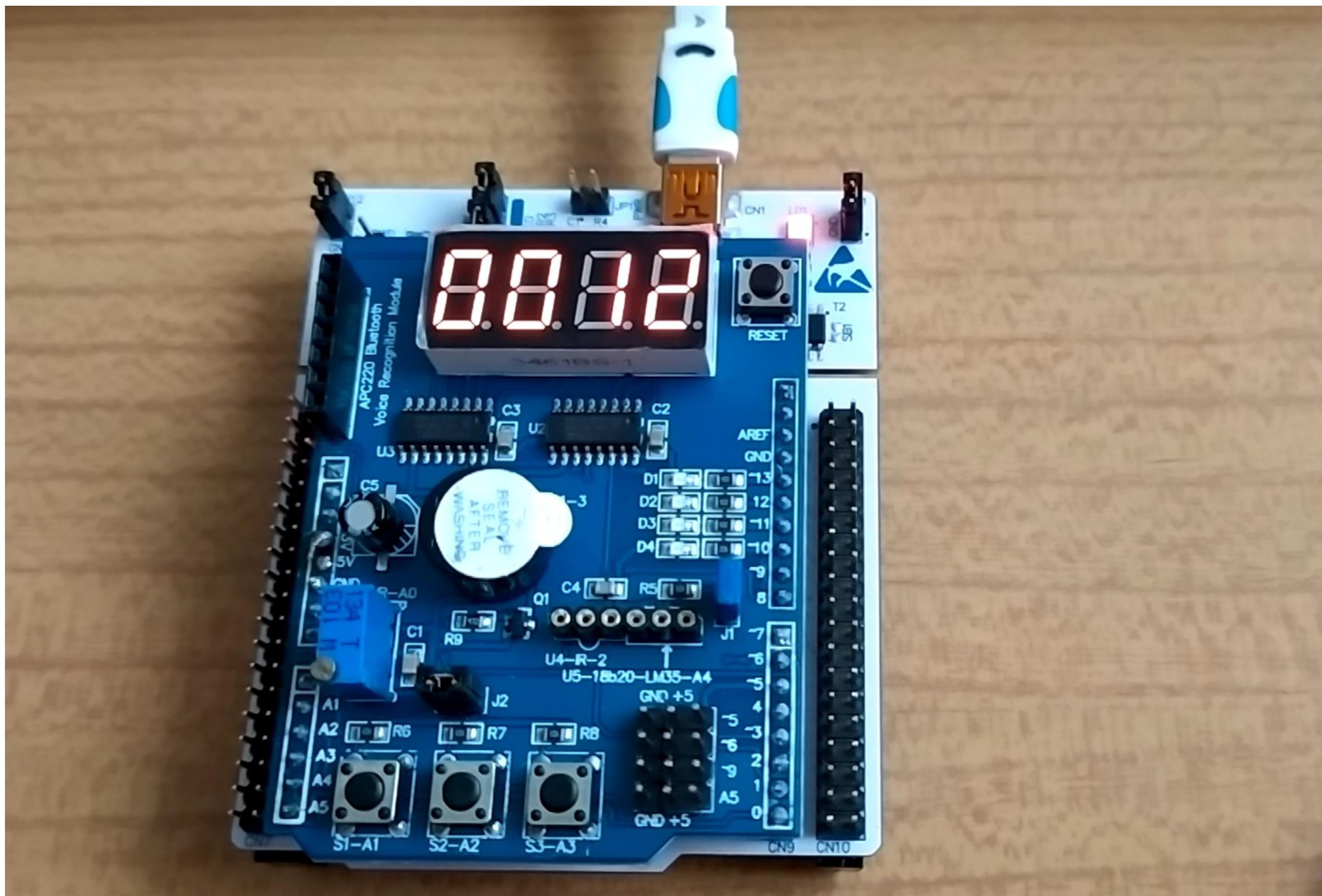
Számjegyválasztó értékek

```
int main() {
  int n = 0;
  buzzer = true;
  thread.start(led2_thread);

  while (true) {
    segment_data[0] = SEGMENT_MAP[(n/1000) %10];
    segment_data[1] = SEGMENT_MAP[(n/100) % 10];
    segment_data[2] = SEGMENT_MAP[(n/10) % 10];
    segment_data[3] = SEGMENT_MAP[n % 10];
    n = n+1;
    buzzer = (n%100) != 0; // Kerek százasoknál
    Thread::wait(250);     // csippant egyet
  }
}
```

Írassuk ki az **n** számláló értékét a multifunkciós kártyán (a multiplex kijelzést a **led_thread** szál végzi)!

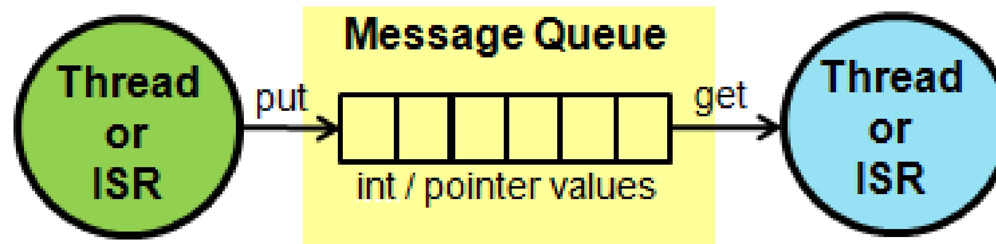
RTOS_display – multiplex kijelzéssel

- A program futási eredménye



RTOS üzenetsor (Message Queue)

- A **Queue** objektumosztály segítségével az **mbed-RTOS** programszálai adatokat, vagy adatstruktúrákra mutató pointereket adhatnak át egymásnak. Az elküldött üzenetek sorbarendezve, egy FIFO tárba kerülnek (first in – first out)



- Az üzenetek típusa nincs megkötve, a **Queue** objektumosztály *templát* osztályként van definiálva, példányosításkor nekünk kell megadni az üzenetek típusát és az üzenetsor méretét (max. darabszám), például:

```
typedef struct {
    float    voltage;        /* ADC feszültségmérés eredménye  */
    float    current;        /* ADC árammérés eredménye        */
    uint32_t counter;        /* Egy kiolvasott számláló értéke */
} message_t;

Queue <message_t, 16> queue;   /* 16 elemű üzenetsor létrehozása */
```

RTOS üzenetsor (Message Queue)

- Az alábbi példában előjel nélküli egésznek definiáljuk az üzenetet, ami az előző oldali példával ellentétben közvetlenül átadódik a fogadónak:

```
typedef uint32_t message_t;
Queue <message_t, 16> queue;
```

- **Üzenetküldés** a **Queue** objektumpéldány **put()** metódusával végezhető:
  `osStatus put (T* data, uint32_t timeout = 0)`
- **data** - az absztrakt T típusú üzenetre mutató pointer, vagy - egyszerű esetben maga az adat, pointernek "álcázva" (azaz típuskényszerítve)
- **timeout** – max. várakozási idő [ms] (alapértelmezetten nincs várakozás)
- **Visszatérési érték:** az alábbi státusz, vagy hibakódok valamelyike:
  - **osOK**: az üzenet bekerült az üzenetsorba
  - **osErrorResource**: az üzenetsor betelt, nincs hely az üzenetnek
  - **osErrorTimeoutResource**: a megadott várakozási idő alatt nem szabadult fel memória a betelt üzenetsorban
  - **osErrorParameter**: érvénytelen paraméter
- **Megjegyzés: put**() megszakításból is hívható, de csak 0 várakozási értékkel

RTOS üzenetsor (Message Queue)

- **Üzenetfogadás** a **Queue** objektumpéldány **get()** metódusával:
```
osEvent get (uint32_t timeout = osWaitForever)
```
- **timeout** – max. várakozási idő [ms] (alapértelmezés: végtelen várakozás, 0 érték megadása esetén pedig nincs várakozás)
- A függvény visszatérési értéke egy **osEvent** típusú struktúra, melynek **status** nevű eleme az alábbi státusz, vagy hibakódok valamelyike lehet:
  - **osOK**: nincs beérkezett üzenet (ha timeout 0-nak volt megadva)
  - **osEventTimeout**: a várakozási idő alatt nem érkezett üzenet
  - **osEventMessage**: üzenet érkezett, melyet az **osEvent** típusú struktúra **value** nevű eleméből érték szerint **value.v** hivatkozással, mutatóként pedig **value.p** hivatkozással vehetünk elő
  - **osErrorParameter**: érvénytelen paraméter, vagy a paraméter értéke kívül esik a megengedett tartományon.
- **Megjegyzés**: Ez a függvény megszakításból is hívható, de csak 0 várakozási értékkel

RTOS_queue/main.cpp 2/1.

- Írjuk át az **RTOS_display** programot úgy, hogy a kiírandó számot egy üzenetsorban adjuk át a megjelenítést végző programszálnak!

```
#include "mbed.h"
#include "rtos.h"
#include "ShiftOut.h"
DigitalOut buzzer(D3);
Serial pc(USBTX,USBRX);                    // UART via ST-Link
typedef uint32_t message_t;                // Simple message format
Queue <message_t, 4> queue;                // Four elements in a row

int main (void) {
    uint16_t raw;                          // Counter variable
    Thread thread2(display_thread);        // Start display thread
    pc.baud(115200);
    buzzer = true;                         // Shut up buzzer
    while (true) {
        Thread::wait(1000);                // Wait for 1 sec
        raw++;                             // Increment counter
        queue.put((message_t*)raw);        // Put counter value in the queue
    }
}
```

Típuskényszerítés

RTOS_queue/main.cpp 2/2.

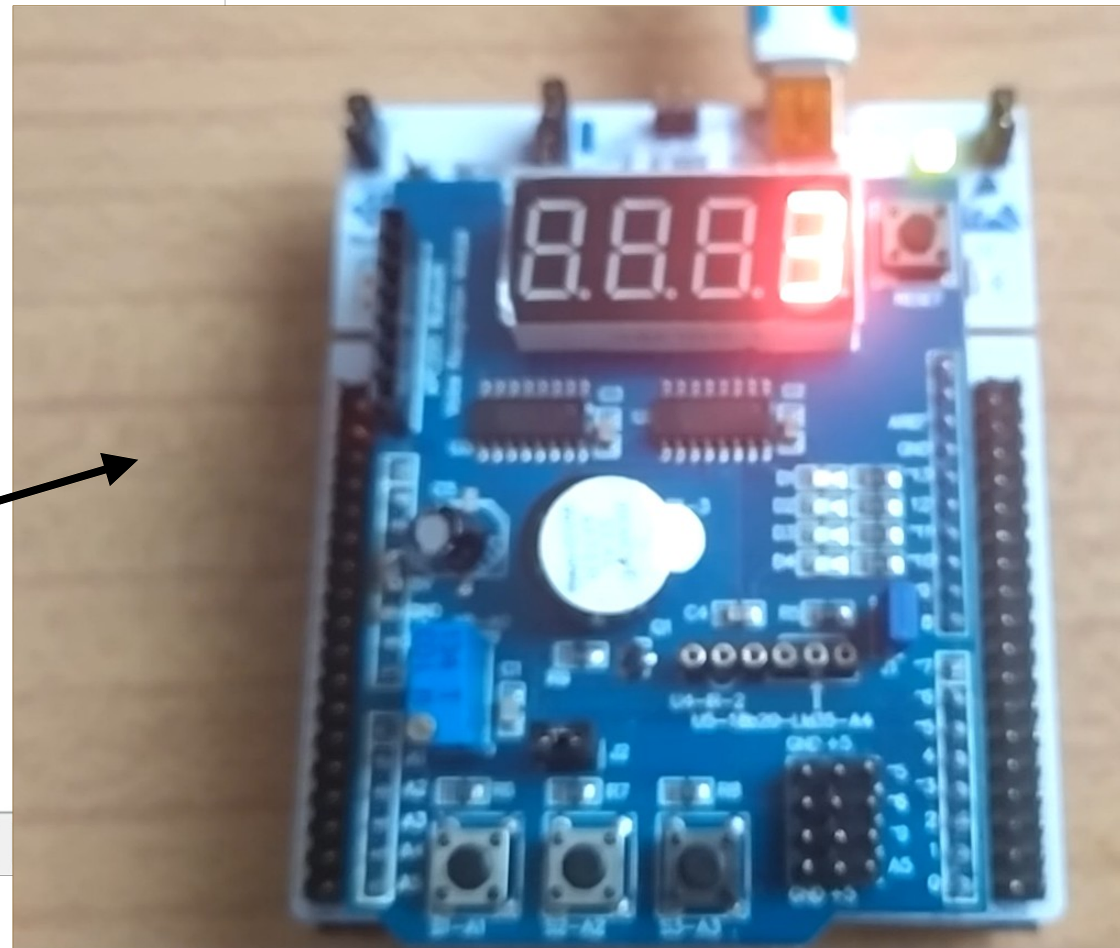
```
void display_thread(void const *argument) {
    ShiftOut display(D7, D8, D4); uint8_t segment_data[4] = {0xFF, 0xFF, 0xFF, 0xFF};
    const uint8_t SEGMENT_MAP[]= {0xC0,0xF9,0xA4,0xB0,0x99,0x92,0x82,0xF8,0x80,0x90};
    const uint8_t SEGMENT_SELECT[] = {0x01, 0x02, 0x04, 0x08}; uint8_t i, idx = 0;
    while (true) {
        osEvent evt = queue.get(0);      // Check for a message without waiting
        switch(evt.status) {
            case osEventMessage:
                printf("osEventMessage = %#05x\n",evt.value.v);  //message arrived
                int n = evt.value.v;
                segment_data[0] = SEGMENT_MAP[(n/1000) %10];
                segment_data[1] = SEGMENT_MAP[(n/100) % 10];
                segment_data[2] = SEGMENT_MAP[(n/10) % 10];
                segment_data[3] = SEGMENT_MAP[n % 10];  break;
            case osOK:          // pc.printf("osOK\n");  //no error, no message arrived
                i = idx++ & 0x03;
                display.write(segment_data[i]);
                display.write(SEGMENT_SELECT[i]); break;
            case osEventTimeout:
                pc.printf("osEventTimeout\n");  break;  //timeout occurred
            case osErrorParameter:
                pc.printf("osErrorParameter\n"); break; //invalid parameter
            default: pc.printf("Unknown error flag: %#08x\n",(uint32_t)evt.status);
        };
        Thread::wait(2);
    }
}
```

RTOS_queue

- A program lassított változatában (1000 ms frissítések, 10 000 ms adatküldések között) jól megfigyelhető az adatküldés és a frissités folyamata

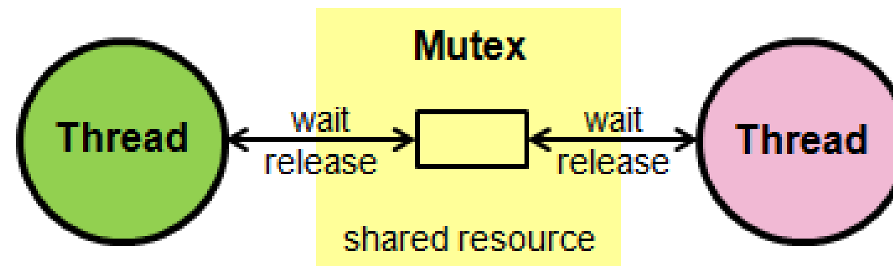
```
osOK
osOK
osEventMessage = 0x002
osOK
osOK
osOK
osOK
osOK
osOK
osOK
osOK
osOK
osEventMessage = 0x003
osOK
osOK
osOK
osOK
```

☑ Autoscroll ☐ Show timestamp Newline



Mutex – kölcsönös kizárás

- A **mutex** név az angol *mutual exclusion* rövidítése, kölcsönös kizárást jelent. Többfeladatos programozásnál előfordul, hogy két vagy több programszál egyidejűleg ugyanazt az erőforrást akarja használni, ami nem engedhető meg, mert pl. összekeverednének a soros porton vagy az USB porton kiküldött adatok



- Ilyenkor versengés alakul ki, s az erőforráshoz hamarabb forduló programszál egy **mutex** segítségével lefoglalja az eszközt, majd használat után megszünteti a lefoglalást
- A kritikus szakaszhoz később érkező programszál pedig várakozásra kényszerül, amíg az erőforrás fel nem szabadul. A programszálak tehát kölcsönösen kizárják egymást a kritikus szakaszon, nem fordulhat elő, hogy egyidejűleg vezérlik a közös használatú perifériát

A Mutex objektumosztály tagfüggvényei

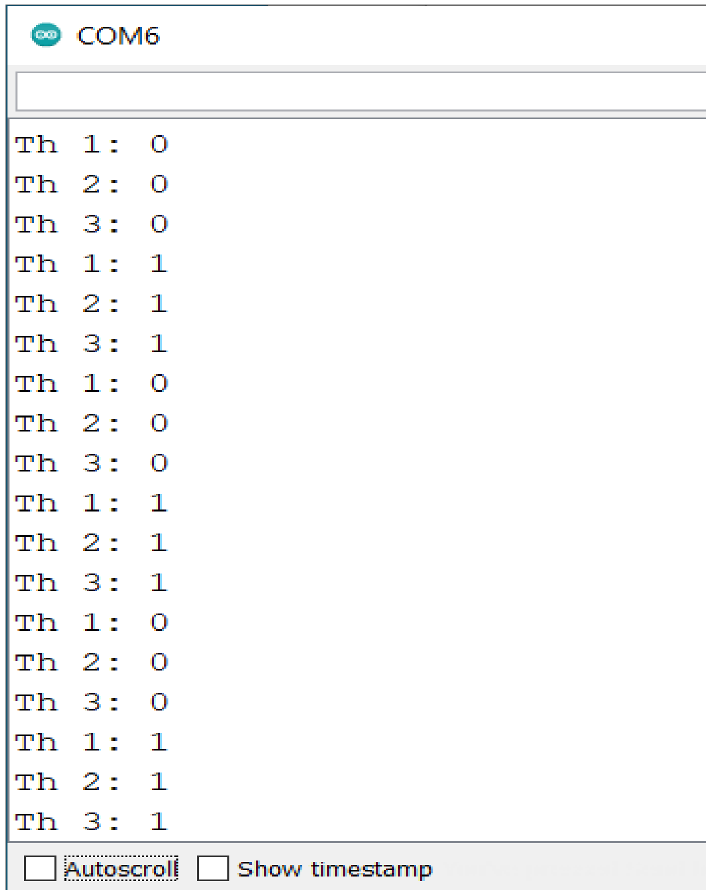
- A kölcsönös kizárást biztosító mutexek létrehozása és kezelése a **Mutex** objektumosztály segítségével végezhető, tagfüggvényeit az alábbi táblázatban foglaltuk össze

A függvény neve	Funkciója
Mutex név	Létrehoz és inicializál egy *név* nevű mutexet
lock(time)	Várakozik, amíg a mutex elérhetővé válik. Ha a mutex foglalt, a várakozás alapértelmezetten korlátlan ideig tart, vagy a **time** paraméterben ezredmásodpercekben megadott idő leteltekor időtúllépéssel kilép a várakozásból
trylock()	Megpróbálja lefoglalni a mutex példányt, de nem várakozik, ha nem sikerült lefoglalni
unlock()	Felszabadítja a korábban lefoglalt mutex példányt

- **Cortex-M0** mikrovezérlők esetében a megosztott **stdio** erőforrások (pl. ***printf()***) hozzáférés védelméről nekünk kell gondoskodnunk
- Megszakítás szinten nem hívhatjuk meg a **Mutex** osztály tagfüggvényeit!

RTOS_mutex/main.cpp

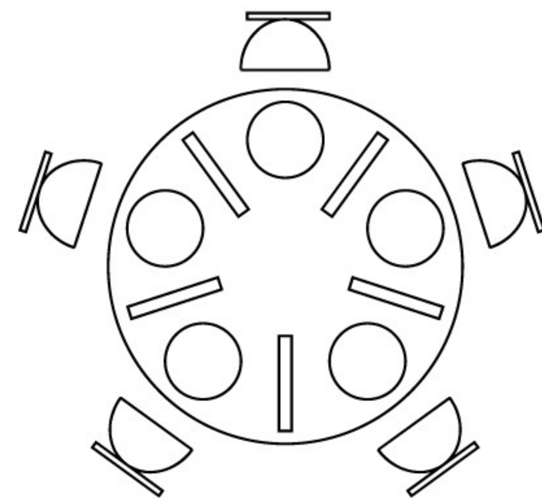
- Ebben az egyszerű példában a kiíratás a megosztott erőforrás, három programszál verseng érte
- Valójában csak a **Cortex M0** MCU-k esetén kell *stdio_mutex*




```
#include "mbed.h"
#include "rtos.h"
Serial pc(USBTX,USBRX);      //UART via ST-Link
Mutex stdio_mutex;

void notify(const char* name, int state) {
    stdio_mutex.lock();
    pc.printf("%s: %d\n\r", name, state);
    stdio_mutex.unlock();
}

void test_thread(void const *args) {
    while (true) {
        notify((const char*)args, 0);
        Thread::wait(1000);
        notify((const char*)args, 1);
        Thread::wait(1000);
    }
}

int main() {
    pc.baud(115200);
    Thread t2(test_thread, (void *)"Th 2");
    Thread t3(test_thread, (void *)"Th 3");
    test_thread((void *)"Th 1");
}
```

Az étkező filozófusok problémája

- Az étkező filozófusok esete (az angol szakirodalomban: **Dining Philosopher Problem**) egy olyan probléma, ami könnyen és gyorsan megérthető, de felmerülő problémái valós informatikai problémákká növik ki magukat
- Egy tibeti kolostorban öt filozófus él. Minden idejüket egy asztal körül töltik. Mindegyikük előtt egy tányér, amelyből sohasem fogy ki a rizs. A tányér mellett jobb és bal oldalon is egy-egy pálcika található, az ábra szerinti elrendezésben.



A filozófusok életüket az asztal melletti gondolkodással töltik. Amikor megéheznek, étkeznek, majd ismét gondolkodóba esnek a következő megéhezésig.
És ez így megy az idők végezetéig. Az étkezéshez egy filozófusnak meg kell szereznie a tányérja melletti mindkét pálcikát. Ennek következtében amíg eszik, szomszédjai nem ehetnek. Amikor befejezte az étkezést, leteszi a pálcikákat, amelyeket így szomszédjai használhatnak.

- Elemezzük, milyen problémákkal szembesülhetnek a filozófusaink:
  - **Holtpont** - előfordulhat-e olyan eset, amikor valamelyik filozófus nem eszik, és nem is gondolkodik?
  - **Kiéheztetés** - előfordulhat-e olyan eset, hogy valamelyik filozófus éhen hal?

A probléma modellezése

- **Filozófusok:** egy-egy programszál, amely véletlen hosszúságú időtartamig várakozik (gondolkodik), majd megpróbálja megszerezni a két pálcikát (megosztott használatú erőforrások) és véletlen időtartamig várakozik (étkezés), ezután leteszi a pálcikákat (elengedi az erőforrásokat) és kezdődik újra a gondolkodás.
- **Pálcikák:** A pálcikák megosztott erőforrások, ezért ezeket egy-egy mutex képviseli.
- **A holtpont elkerülése** a programozó felelőssége. Ha egyszerre minden filozófus megragadja a jobb kezénél levő pálcikát, akkor máris holtpontra jutottunk!
- **Stratégia:** Ha valamelyik filozófus nem tudja megszerezni mindkét pálcikát, akkor leteszi a kezéből az elsőnek felvettet is, hogy a szomszédjai étkezni tudjanak, majd véletlenszerűen választott rövid ideig tartó várakozás után újra próbálkozik.
- **Algoritmus:** Az *i.* programszál futásakor a **trylock()** metódussal ellenőrizzük, hogy sikerült-e a jobboldali pálcikához tartozó *i-1.* mutexet lefoglalni. Ha sikerült, akkor megpróbáljuk lefoglalni a másikat (az *i%5.* mutexet) is. Ha ez is sikerült, akkor kezdődhet a falatozás, majd letesszük a pálcikákat és kezdődhet a gondolkodás. Ha a második mutexet nem sikerült lezárni, akkor feloldjuk az első mutexet, s véletlen időtartam elteltével próbálkozunk újra.
- Azért kell *i%5* az *i* helyett, mert az 5. filozófus balján a 0. sorszámú pálcika van!

RTOS_philosophers/main.cpp 2/1.

- Az evőpálcikákat egy-egy mutex képviseli

```
#include "mbed.h"
#include "rtos.h"
// Mutex stdio_mutex;
Mutex chopstick[5];          // mutexarray representing 5 chopsticks
Serial pc(USBTX,USBRX);      // UART via ST-Link

void notify(int num, int state) {
    // stdio_mutex.lock();  // We don’t need it for the NUCLEO board
    if(state) { pc.printf("Philospher %d is EATING \n\r", num);}
      else { pc.printf("Philospher %d is thinking \n\r", num); }
    // stdio_mutex.unlock();
}

int main() {
    pc.baud(115200);
    Thread t2(philosopher, (void *)2U);
    Thread t3(philosopher, (void *)3U);
    Thread t4(philosopher, (void *)4U);
    Thread t5(philosopher, (void *)5U);
    philosopher((void *)1U);
}
```

RTOS_philosophers/main.cpp 2/2.

- A filozófus programszálak ugyanazt a ***philosopher*** függvényt futtatják, csak az asztalnál elfoglalt helyükben (sorszám) különböznek

```
void philosopher(void const *args) {
  while (true) {
    if(chopstick[(int)args-1].trylock()) {
      if(chopstick[(int)args%5].trylock()) {
        notify((int)args,1);                  //Start EATING
        Thread::wait(1000+rand()%1000);       // 1-2 sec
        chopstick[(int)args%5].unlock();      //Release chopsticks
        chopstick[(int)args-1].unlock();
        notify((int)args,0);                  //Start Thinking
        Thread::wait(2000+rand()%2000);       //Get's hungry after this time...
      } else {
          chopstick[(int)args-1].unlock();
          Thread::wait(100+rand()%100);       //Wait for random time if failed
      }
    } else {
        Thread::wait(100+rand()%100);         //Wait for random time if failed
    }
  }
}
```

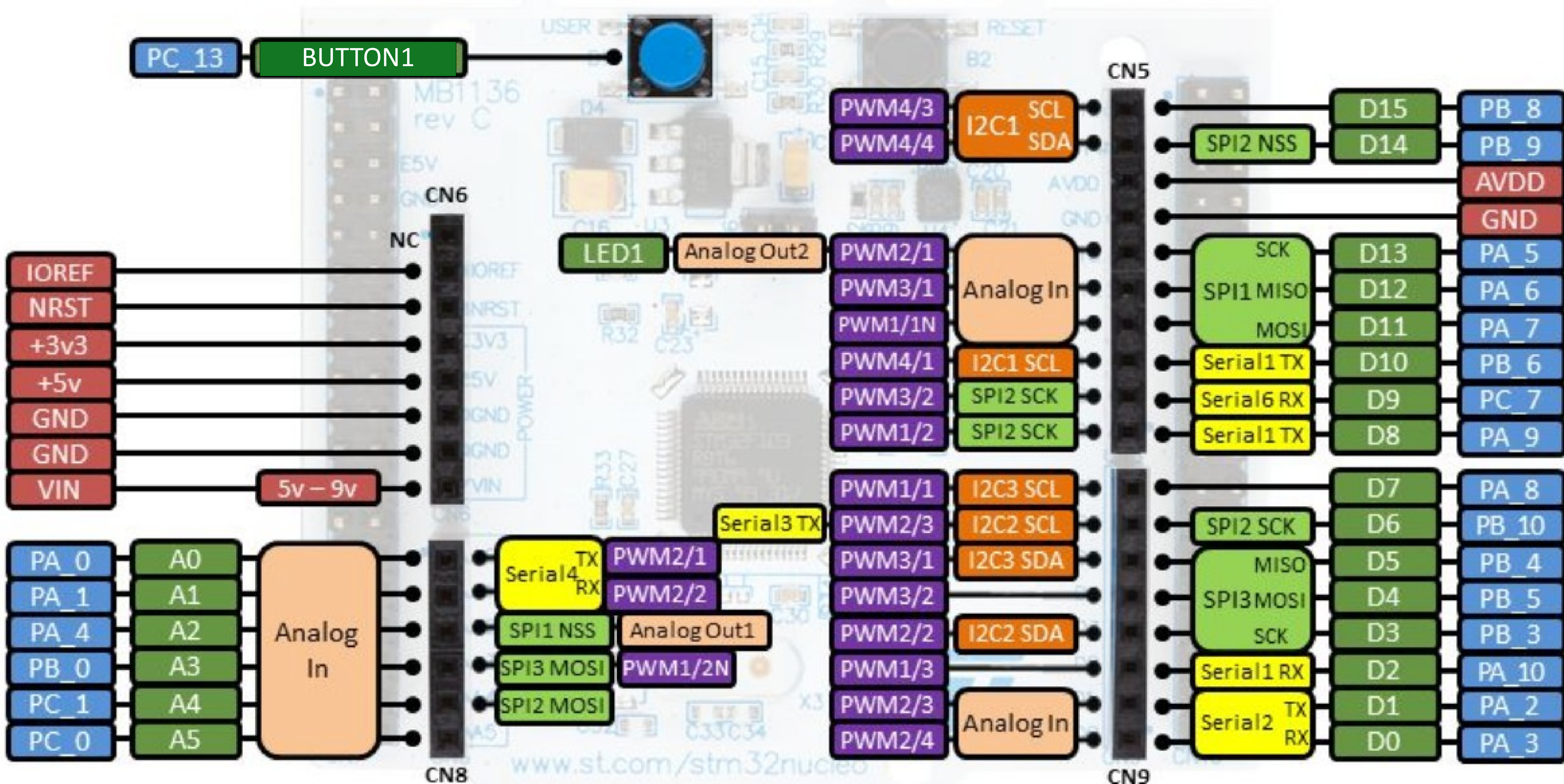
RTOS_philosophers

- A program futási eredménye

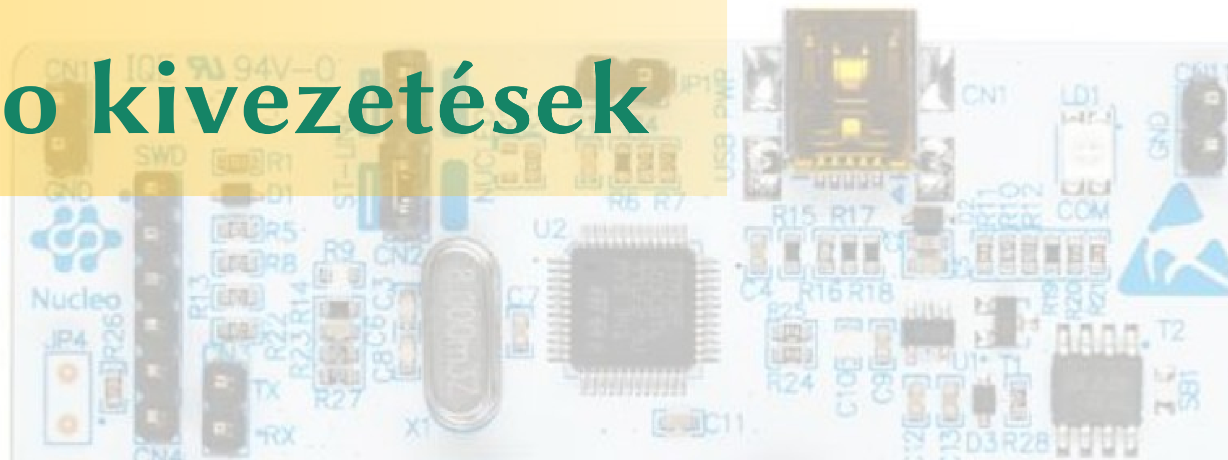
```
COM6
Send
Philospher 1 is EATING
Philospher 3 is EATING
Philospher 3 is thinking
Philospher 4 is EATING
Philospher 1 is thinking
Philospher 2 is EATING
Philospher 2 is thinking
Philospher 4 is thinking
Philospher 5 is EATING
Philospher 5 is thinking
Philospher 1 is EATING
Philospher 3 is EATING
Philospher 1 is thinking
Philospher 3 is thinking
Philospher 2 is EATING
Philospher 4 is EATING
Philospher 4 is thinking
Philospher 5 is EATING
Autoscroll  Show timestamp  Newline  115200 baud  Clear output
```

Arduino kompatibilis kivezetések

- **Kivezetés azonosításra a kék, illetve a sötétzöld címkék használhatók (pl. PA_5, D13, vagy LED1)**



Morpho kivezetések



Nucleo F446RE
Morpho Headers