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
    }
}
```

8. Szemaforok, eseményjelzők, üzenetküldés

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

- **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

Lab08_dpp_easy – az étkező filozófusok visszatértek (és enni kérnek)

Lab08_4semaphores – futási sorrend biztosítása

Lab08_rtos_signals – eseményjelző küldés/fogadás

Lab08_signals_to_main – jelző küldése thread1-nek

Lab08_rtos_mail – üzenetküldés és fogadás

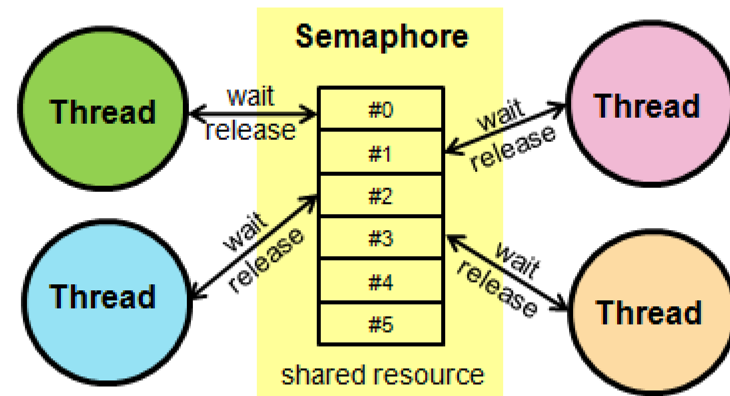
A mintaprogramok az os.mbed.com/users/cspista/code/ oldalon is megtalálhatók

Programszálak kommunikációja

- Az **Mbed-RTOS** számos eszközt biztosít a programszálak szinkronizálására, események jelzésére, illetve üzenetküldésre
  - **Mutex** (kölcsönös kizárás)
  - **Semaphore** (számláló szemafor)
  - **Signal** (eseményjelző bitek)
  - **Queue** (üzenetsor)
  - **MemoryPool** (memóriakészlet)
  - **Mail** (üzenetküldés – összekapcsolt **Queue** és **MemoryPool**)
- Fentiek közül a mai előadásban a **szemaforok** (*semaphore*), az **eseményjelzők** (*signal*) és az **üzenetküldés** (*Mail*) használatával ismerkedünk meg, egyszerű példaprogramokon keresztül

A szemafor

- **A szemafor** olyan absztrakt adattípus, amit az osztott erőforrások egy készletéhez való hozzáférés szabályozásához, illetve programszálak szinkronizálásához használnak a többszálú környezetekben
- Megalkotása **Edsger Dijkstra** holland matematikusnak, a programozás egyik úttörőjének nevéhez fűződik



- **Michael Barr** írása (Mutexes and Semaphores Demystified) szerint a mutexek és szemaforok fogalmának összekeverése történelmi eredetű, s egyszerű analógiák segítségével tisztázza, hogy mi a különbség köztük
- A szemafor önmagában nem oldja meg a több azonos erőforrás megosztásának problémáját, ehhez további, kiegészítő információra, illetve eszközre is szükség van
- A szemafor ***„producer – consumer”*** viszonylatban is használható szinkronizálásra

A Semaphore objektumosztály tagfüggvényei

- Az osztott erőforrások egy készletéhez való hozzáférést szabályozó szemaforok létrehozása és kezelése a **Semaphore** objektumosztály segítségével végezhető, tagfüggvényeit az alábbi táblázatban foglaltuk össze. Az **mbed-rtos**-ban implementált **Semaphore** objektumosztály ún. **számláló szemaforokat** kezel, amelyek értékének felső határát az rtx/TARGET_CORTEX_M/cmsis_os.h/ forrásállomány ***osFeature_Semaphore*** nevű makrója 65 535-re maximálta

Függvénynév	Funkció
Semaphore név(*szám*)	Létrehoz egy "név" nevű szemafor objektumot és inicializálja a megadott számmal (a szám a kezdetben rendelkezésre álló erőforrások száma)
wait(*time*)	Várakozik, amíg a szemaforral kezelt erőforrások valamelyike elérhetővé válik, majd lefoglalja (eggyel csökkenti a szemafor értékét). Ha a szemafor nulla (nincs elérhető erőforrás), a várakozás alapértelmezetten korlátlan ideig tarthat, vagy a ***time*** paraméterben ezredmásodpercekben megadott idő leteltekor időtúllépéssel kilép a várakozásból
release()	Felszabadítja a korábban **wait**() metódussal lefoglalt szemafort (azaz eggyel növeli a szemafor értékét)

Az étkező filozófusok visszatértek

- Az előző előadásban ismertetett "étkező filozófusok" problémája kapcsán az alábbi programban **egy egyszerűsítő könnyítést alkalmazunk:** a szabad evőpálcikák most nincsenek kiosztva a tányérok mellett, hanem középen egy tárolóban helyezkednek el. Így tehát a filozófusok bármelyik két szabad pálcikát felvehetik
- A programban a filozófusokat **egy-egy programszállal** modellezzük, az 5 db pálcika foglaltságát pedig egy **5-ig számláló szemafor** tartja nyilván.

  A programszálak törzsét az előző fejezet végén bemutatott mintaprogramhoz hasonlóan ugyanaz a függvény képezi. A "filozófusaink" tehát csak neveikben különböznek, illetve a véletlenszám generálásnak köszönhetően különböző időpontokban kezdenek gondolkodni, vagy étkezni.
- Étkezéskor két pálcikát kell megszerezni, ezért kétszer csökkentjük az ***s*** szemafor értékét az **s.wait()** függvényhívással. Ennek megfelelően étkezés végén is kétszer növeljük a szemafor értékét (mindkét pálcikát letesszük...) az **s.release()** függvényhívással
- A szemafor kezdeti értékét **a konstruktor meghívásakor** állítjuk be 5-re

Lab08_dpp_easy/main.cpp

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

Semaphore s(5);                         // a pool of 5 chopsticks
Timer mytime;

void notify(const char* name) {
    printf("%s acquired two chopsticks %8.1f\n\r", name,mytime.read());
}

void test_thread(void const* args) {
    while (true) {
        Thread::wait(1000+rand()%500); // Thinking (1 - 1.5 sec)
        s.wait();    s.wait();
        notify((const char*)args);
        Thread::wait(500+rand()%500);  // Eating  (0.5 - 1.0 sec)
        s.release(); s.release();
    }
}

int main (void) {
    mytime.start();
    Thread t2(test_thread, (void *)"Philosopher 2");
    Thread t3(test_thread, (void *)"Philosopher 3");
    Thread t4(test_thread, (void *)"Philosopher 4");
    Thread t5(test_thread, (void *)"Philosopher 5");
    test_thread((void *)"Philosopher 1");
}
```

Lab08_dpp_easy futási eredménye

- A program futási eredménye

```
COM6

Philosopher 5 acquired two chopsticks      1.1
Philosopher 4 acquired two chopsticks      1.2
Philosopher 2 acquired two chopsticks      1.9
Philosopher 1 acquired two chopsticks      2.0
Philosopher 3 acquired two chopsticks      2.7
Philosopher 4 acquired two chopsticks      3.2
Philosopher 5 acquired two chopsticks      3.3
Philosopher 1 acquired two chopsticks      3.9
Philosopher 2 acquired two chopsticks      4.2
Philosopher 3 acquired two chopsticks      4.7
Philosopher 5 acquired two chopsticks      5.1
Philosopher 4 acquired two chopsticks      5.6
Philosopher 1 acquired two chopsticks      6.0
Philosopher 3 acquired two chopsticks      6.6
Philosopher 2 acquired two chopsticks      7.0
Philosopher 5 acquired two chopsticks      7.4
Philosopher 4 acquired two chopsticks      7.9
Philosopher 1 acquired two chopsticks      8.2
Philosopher 3 acquired two chopsticks      8.9
Philosopher 5 acquired two chopsticks      9.2
```

Send | Autoscroll | Show timestamp | Newline | 9600 baud | Clear output

Futási sorrend biztosítása

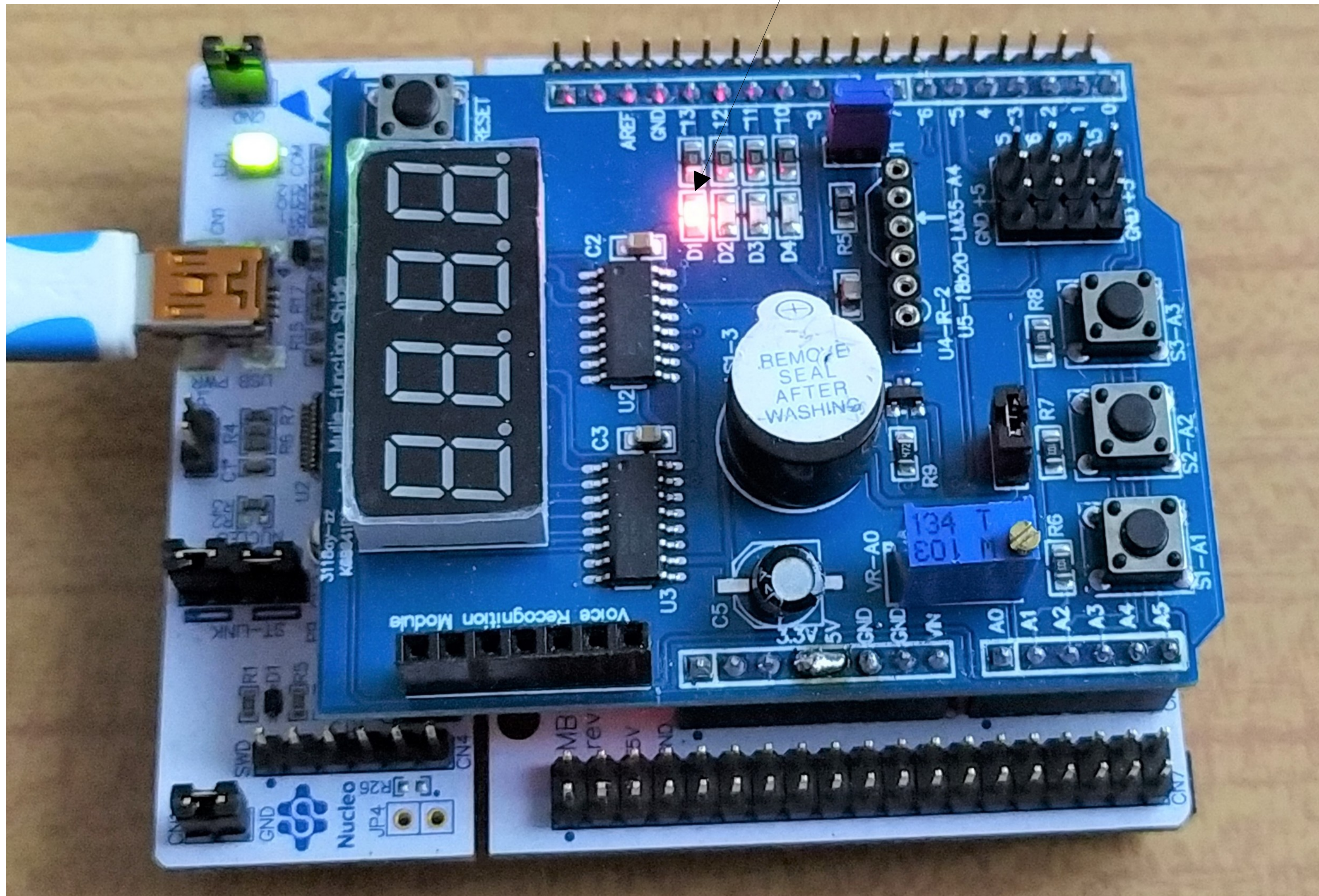
- **A szemaforok** tipikus felhasználásai a programszálak szinkronizálására szolgálnak. Ilyen lehet például:
  - **A programfutás sorrendiségének biztosítása** - ahol a programszálak úgy adogatják egymásnak a szemafor zsetonokat, mint a staféták a váltóbotot a váltófutásnál.
  - **Randevú** (rendezvous) - ahol a szemaforok azt biztosítják, hogy két programszál egy adott pontban várja be egymást
  - **Sorompó** (barrier) - ahol az a cél, hogy több párhuzamosan futó programszál egy adott pontban várja be egymást
- **A következő program** az első esetre mutat egy példát: négy programszálat szinkronizálunk négy szemafor segítségével. Az Arduino Multifunkciós kártya LED-jei az éppen futó task sorszámát jelzik
- A kötött sorrendet az biztosítja, hogy mindegyik programszál "fogyasztóként" egy olyan szemaforra vár, amelynek a "termelője" az előtte futó programszál
- Ahhoz, hogy a staféta elinduljon, "be kell tenni" egy zsetont a rendszerbe, azaz valamelyik szemafort szabadra kell állítani

Lab08_4semaphores/main.cpp

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

Semaphore sem[] = {(1),(0),(0),(0)};           // Set of semaphores
DigitalOut led[] = {(D13),(D12),(D11),(D10)}; // Set of LEDs

void led_thread(void const* args) {
    int i = (int)args-1;                       // Idx of this task
    int inext = (i+1)%4;                       // Idx of next task
    while (true) {
        sem[i].wait();
        led[i] = 0;                            // ith LED on
        Thread::wait(500+rand()%500);
        led[i] = 1;                            // ith LED off
        sem[inext].release();                  // Start new task
    }
}

int main (void) {
    for(int i=0; i<4; i++) {
        led[i]=1;                              // LEDs off
    }
    Thread t2(led_thread, (void *)2U);
    Thread t3(led_thread, (void *)3U);
    Thread t4(led_thread, (void *)4U);
    led_thread((void *)1U);
}
```

Lab08_4semaphores futási eredménye

- A programszálak aktiválását követve a LED-ek egymás után gyúlnak ki, illetve alszanak el



RTOS eseményjelzők (Signals)

- A programszálak (és a megszakítások) **eseményjelzőket** tudnak **küldeni** egymásnak, s a programszálak **várakozhatnak** valamilyen **eseményjelzőre**
- **Az eseményjelzők** a programszálhoz tartozó **TCB** (task control block) egyik elemének bitjei, ezek számát az rtx/TARGET_CORTEX_M/cmsis_os.h/ forrásállomány ***osFeature_Signals*** nevű makrója 16 értékkel definiálja, tehát programszálanként legfeljebb 16 eseményjelzőt használhatunk ($2^0 - 2^{15}$)
- ***osEvent*** **Thread::signal_wait**(*signals*, *timeout*) – várakozás egy, vagy több eseményjelzőre, megadott, vagy korlátlan ideig (statikus tagfüggvény!)
  signals - bitmaszk, kijelöli, hogy mely eseményjelzőkre várakozunk (0: bármelyik)
  timeout – várakozási idő (ms), de lehet 0, vagy ***osWaitForever*** is
- **signal_set**(*signals*) – eseményjelző küldése (az objektumpéldány metódusa)
  signals - bitmaszk, amely kijelöli, hogy melyik eseményjelzőt, vagy eseményjelzőket akarjuk elküldeni. A függvény visszatérési értéke a küldés előtti állapotot jellemző eseményjelző bitcsoport, vagy hibás paraméter esetén a 0x80 000 000 hibakód. Ez a függvény megszakításból is hívható
- **signal_clr**(*signals*) – az eseményjelző(k) soron kívüli törlése (ez a függvény is az objektumpéldány metódusa)

Egyszerű mintapélda

- Az mbed Handbook mintapéldájában az első programszál (a **main()** függvény) rendszeres időközönként elküld egy eseményjelzőt a második programszálnak (thread2), amely mindaddig várakozik, amíg az eseményjelző meg nem érkezik. Minden eseményjelző megérkezésekor átbillenti **LED1** állapotát.

```
#include "mbed.h"
#include "rtos.h"
DigitalOut led(LED1);

void led_thread(void const *argument) {
    while (true) {
        osEvent evt = Thread::signal_wait(0x1);    //Wait for a signal
        led = !led;                                //toggle LED1 state
    }
}

int main (void) {
    Thread thread2(led_thread);
    while (true) {
        Thread::wait(1000);
        thread2.signal_set(0x1);                  //Send a signal for thread2
    }
}
```

Lab08_rtos_signals

- Az egyszerű mintapélda bővített változatában a **Thread::signal_wait()** függvény visszatérési értéket is megvizsgáljuk és kiíratjuk, miközben a **main()** függvény (2^0 – 2^{15}) között minden lehetséges jelzőt kiküld **tread2** számára
- A **Thread::signal_wait()** függvény lehetséges visszatérési értékei:
  - **osOK:** nem érkezett eseményjelző (timeout pedig 0-nak volt megadva)
  - **osEventTimeout:** nem érkezett eseményjelző az időtúllépésig
  - **osEventSignal:** eseményjelző érkezett, a visszatérési érték (amely egy **osEvent** struktúra) **value.signals** komponense tartalmazza a beérkezett eseményjelző biteket
    Megjegyzés: a jelzőbitek kiolvasáskor automatikusan törlődnek
  - **osErrorValue:** a paraméter értéke kívül esik a megengedett tartományon
  - **osErrorISR: signal_wait()** megszakításból nem hívható meg

Lab08_rtos_signals/main.cpp

```
#include "mbed.h"
#include "rtos.h"
DigitalOut led(LED1);

void led_thread(void const *argument) {
  while (true) {
    osEvent evt = Thread::signal_wait(0);       //Wait for any signal
    switch(evt.status) {
      case osEventSignal:
        printf("osEventSignal = %#05x\n",evt.value.signals); //signal event occurred
        break;
      case osOK:
        printf("osOK\n");
        break;
      case osEventTimeout:
        printf("osEventTimeout\n");
        break;
      case osErrorValue:
        printf("osErrorValue\n");
        break;
      default:
        printf("Unknown error flag: %#08x\n",(uint32_t)evt.status);
        break;
    };
    led = !led;
  }
}
```

```
int main (void) {
  int32_t signal_mask = 0x1;
  Thread thread2(led_thread);
  while (true) {
    Thread::wait(1000);
    thread2.signal_set(signal_mask);
    signal_mask <<=1;
    if(signal_mask > 0x8000) signal_mask=0x1;
  }
}
```

Lab08_rtos_signals futási eredménye

COM6

Send

```
osEventSignal = 0x001
osEventSignal = 0x002
osEventSignal = 0x004
osEventSignal = 0x008
osEventSignal = 0x010
osEventSignal = 0x020
osEventSignal = 0x040
osEventSignal = 0x080
osEventSignal = 0x100
osEventSignal = 0x200
osEventSignal = 0x400
osEventSignal = 0x800
osEventSignal = 0x1000
osEventSignal = 0x2000
osEventSignal = 0x4000
osEventSignal = 0x8000
osEventSignal = 0x001
osEventSignal = 0x002
osEventSignal = 0x004
osEventSignal = 0x008
```

Autoscroll Show timestamp Newline 9600 baud Clear output

Variáljuk a Lab08_rtos_signals programot!

1) Módosítsuk a programot úgy, hogy **Thread::signal_wait(2,0)** álljon benne!
   A függvényhívás első paramétere a 0x2 eseményjelző figyelését írja elő, a második paraméter szerint a várakozási idő nulla. Ekkor az **osOK** hibakód lesz a visszatérési érték, ha a fenti függvényhívás kiadásakor még nincs beállítva a **0x2** eseményjelző.
2) Módosítsuk a programot úgy, hogy **Thread::signal_wait(2,500)** álljon benne!
   Ekkor az **osEventTimeout** hibakód lesz a visszatérési érték, ha a **0x2** eseményjelző 500 ezredmásodpercen belül nem kerül beállításra
3) Módosítsuk a programot úgy, hogy **Thread::signal_wait(0x10 000)** álljon benne! Ez érvénytelen bitmaszk (>16 bit), ezért az **osErrorValue** hibakód lesz a visszatérési érték
4) Módosítsuk a programot úgy, hogy **Thread::signal_wait(0x5),** vagy bármilyen egynél több biten 1-et tartalmazó szám álljon benne! Ekkor csak az összes előírt eseményjelző beérkezése után szűnik meg a várakozás (**LED1** ritkábban vált állapotot).
5) Módosítsuk a programot úgy, hogy **thread2.signal_set(signal_mask)** függvényhívásnál **0x5,** vagyis ugyanaz legyen a bitmaszk, amit a 4. feladatnál a **Thread::signal_wait()** paramétereként megadtunk! Ekkor egyszerre több eseményjelzőre várunk, de azok egyszerre, csoportos küldéssel meg is érkeznek, ezért **LED1** állapotváltása 1 másodpercenként történik (mint az eredeti változatban)

- Fentiek kipróbálásához mellékeltük a sorszámmal azonosított bináris állományokat

Eseményjelző küldése az első programszálnak

- **Probléma:** hogyan küldhetünk eseményjelzőt a **main()** függvénynek, amely az első programszál, de nincs explicit módon megadott **Thread** típusú neve?
- **1. lépés:** kérjük le a **main()** programszál taszk azonosítóját!

```
osThreadId mainThreadID;  // globális változó
int main (void) {
    mainThreadID = Thread::gettid();
...
```

- **2. lépés:** az **mbed-RTOS API** alatti rétegekbe belenyúlva, használjuk az CMSIS RTOS API **osSignalSet()** függvényét, amelynek átadható egy **osThreadID** típusú azonosító
- **osSignalSet(***osThreadID, signals***)** – eseményjelző(k) beállítása
  osThreadID – a megcímzett programszál azonosítója
  signals – bitmaszk, amely kijelöli, hogy melyik eseményjelzőt, vagy eseményjelzőket akarjuk elküldeni

Lab08_signals_to_main/main.cpp

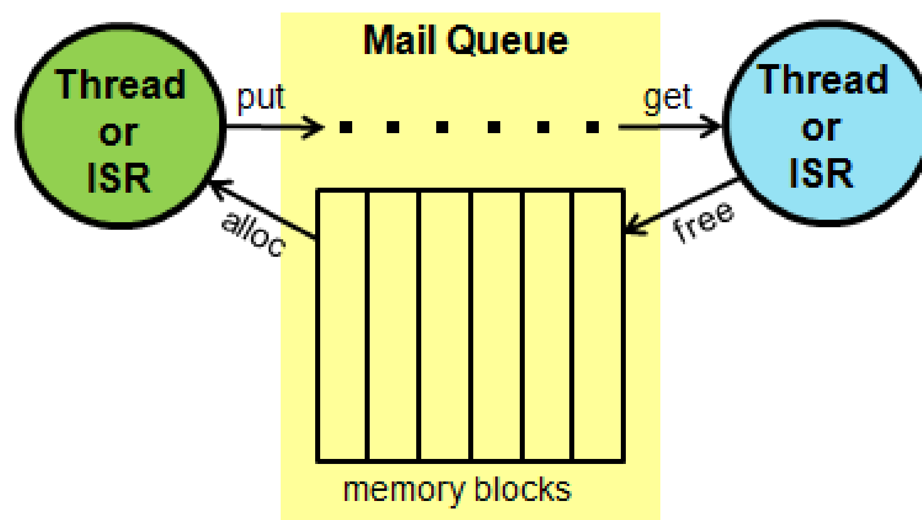
- Most most a **main()** függvényben várunk eseményjelzőre és kapcsolgatjuk **LED1**-et
- Az eseményjelzőt a másik programszál küldi, 1 másodpercenként

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

DigitalOut led(LED1);
osThreadId mainThreadID;

void signal_thread(void const *argument) {
    while (true) {
        Thread::wait(1000);
        osSignalSet(mainThreadID, 0x1);
    }
}

int main (void) {
    mainThreadID = Thread::gettid();
    Thread thread(signal_thread);

    while (true) {
        // Signal flags that are reported as event are automatically cleared.
        osSignalWait(0x1, osWaitForever);
        led = !led;
    }
}
```

Üzenetküldés a Mail objektumosztály használatával

- Az előző előadásban ismertetett **Queue** objektumok segítségével csak mutatókat (vagy mutatónak álcázott egész számokat) tudtunk küldeni, összetett adatok esetén a küldőnek kell tárolnia azokat
- A **Mail** objektumoknál azonban küldéskor tárterületet foglalhatunk le és abban adhatjuk át az adatokat
- A **Mail** objektumosztály a korábban tárgyalt **Queue** és a külön is használható **MemoryPool** egyesítésének tekinthető
- A **Mail** objektumosztály is *templát osztály*, azaz nekünk kell megadnunk, hogy milyen adatstruktúrákat tároljon, és hány darabot Például:

```
typedef struct {
    float    voltage;
    float    current;
    uint32_t counter;
} message_t;

Mail <message_t,16>  mailbox;
```



A Mail objektumosztály

- Üzenetküldés szempontjából az **mbed-RTOS Mail** is templát osztály, ugyanúgy használható, mint a **Queue** üzenetsor
- Memóriakezelés (lefoglalás, eltárolás, felszabadítás) szempontjából pedig ugyanolyan, mint a **MemoryPool** (memóriakészlet)
- A **Mail** osztály tagfüggvényei megszakításból is hívhatók, de csak 0 várakozási idő megadásával

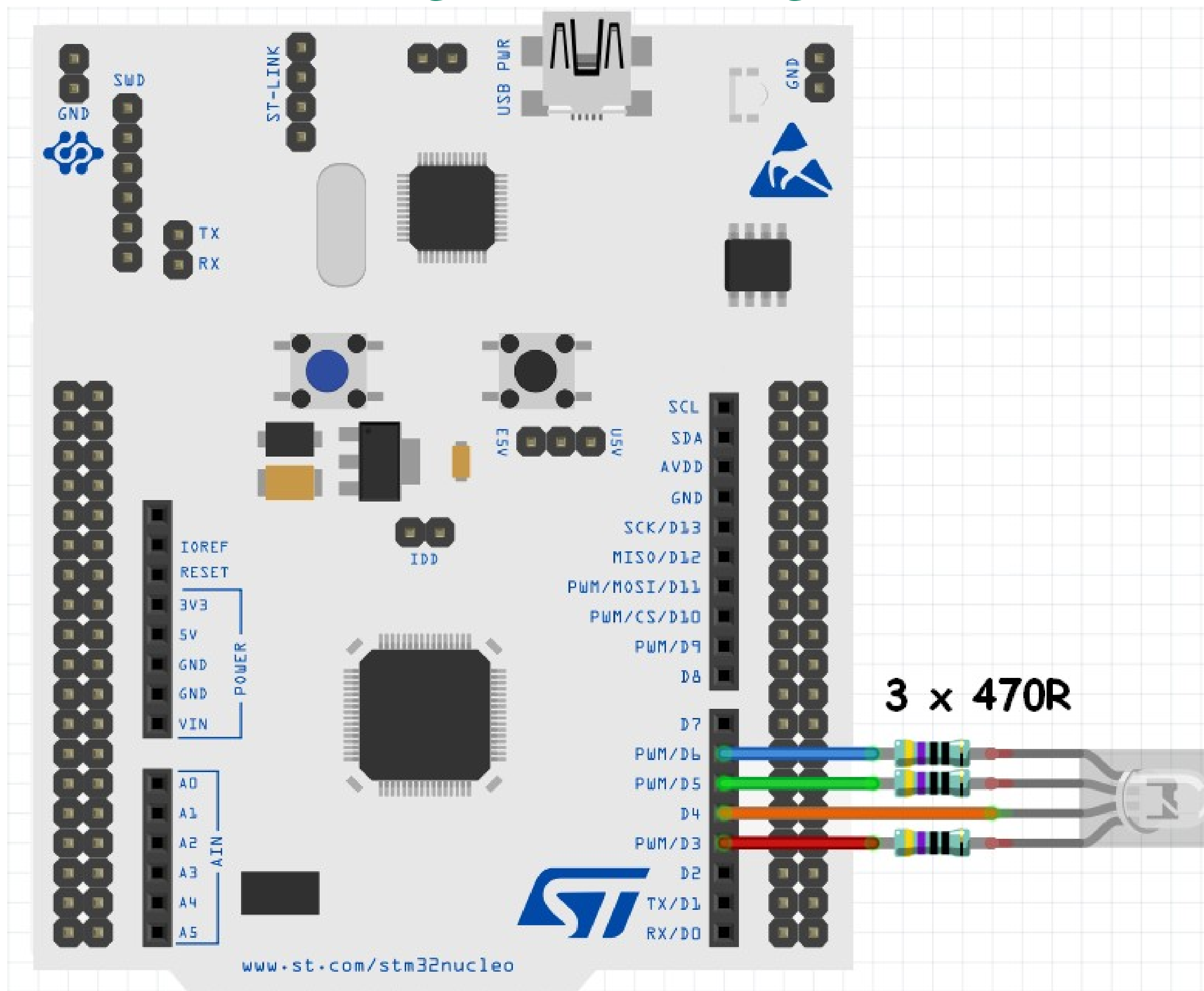
Függvénynév	Funkció
Mail <mail_t,n> név	Létrehoz egy "**név**" nevű **Mail** objektumot és helyet biztosít **n** darab, **mail_t** típusú adatstruktúrának
p = **alloc**(*timeout*)	Lefoglal egy **mail_t** típusú adatstruktúrának való helyet
free(*p*)	Felszabadítja a korábban **alloc**() metódussal lefoglalt és a **p** mutatóval kijelölt helyet
put(*p*)	A **p** mutatóval címzett üzenetet elküldi
evt = **get**(*timeout*)	Üzenet fogadása. Üzenet érkezésekor **evt.status** értéke **osEventMail** lesz, az üzenetre mutató pointer **evt.value.p**

Lab08_rtos_mailbox

- Ebben a programban a **main()** függvényben folytonos színátmeneteket „termelünk” és az RGB színkomponenseket **Mail** üzenetek formájában adjuk át a megjelenítést végző programszálnak
- A közös anódú RGB LED bekötése:

  D3 ⟶ RED (katód)
  D4 ⟶ közös anód
  D5 ⟶ GREEN (katód)
  D6 ⟶ BLUE (katód)

- A közös anódot normális helyen természetesen a tápfeszültségre kötik, itt csak lustaságból került a D4 kivezetésre



Lab08_rtos_mailbox/main.cpp

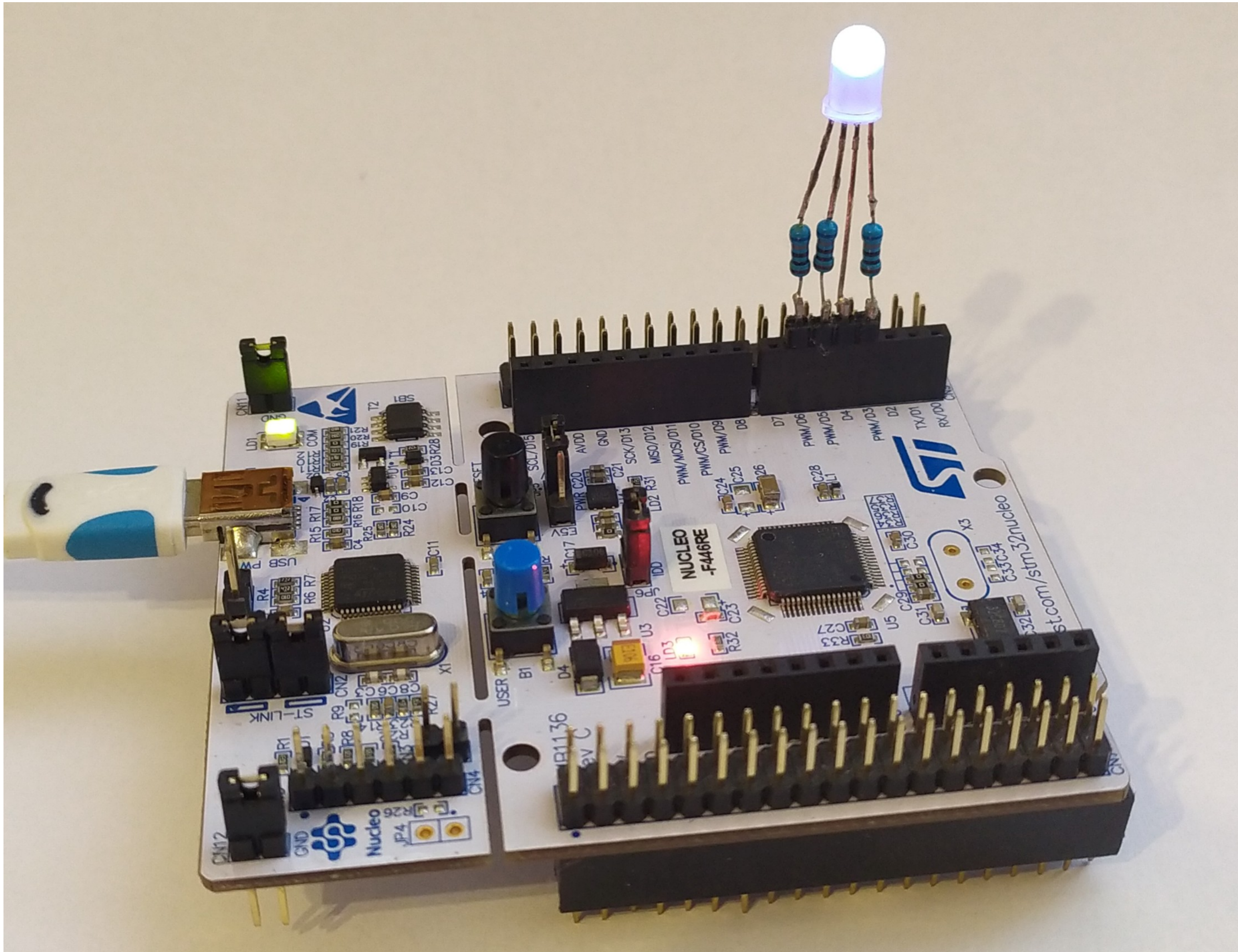
```
#include "mbed.h"
#include "rtos.h"
PwmOut rled(D3); PwmOut gled(D5); PwmOut bled(D6);
DigitalOut led_common(D4);

typedef struct { float red; float green; float blue; } message_t;
Mail <message_t,4> mbox;                                    //Mailbox for 4 messages

void led_thread(void const *argument) {
    rled.period_ms(20);                                     //Set period to 20 ms
    rled.write(1.0f);                                       //Initialize to 0% duty cycle
    gled.period_ms(20);                                     //Set period to 20 ms
    gled.write(1.0f);                                       //Initialize to 0% duty cycle
    bled.period_ms(20);                                     //Set period to 20 ms
    bled.write(1.0f);                                       //Initialize to 0% duty cycle
    while (true) {
        osEvent evt = mbox.get();                           //Wait for a message
        if(evt.status == osEventMail) {
            message_t *mymessage = (message_t*)evt.value.p;
            rled = 1.0f - mymessage->red;                   // Invert data for common anode LED
            gled = 1.0f - mymessage->green;
            bled = 1.0f - mymessage->blue;
            mbox.free(mymessage);                           //Free up memory
        }
    }
}
```

Lab08_rtos_mailbox/main.cpp

```
float frand(void) {
    int32_t rv = 0x8000 -(rand()&0xFFFF);
    return (rv*rv/1073741824.0f);
}

int main (void) {
    float RGB1[3], RGB2[3], INC[3];
    led_common = 1;                                         // for common anode...
    Thread thread2(led_thread);
    for (int x=0; x<3; x++) { RGB1[x] = frand(); }          // Create a random color
    while (true) {
        for (int x=0; x<3; x++) { RGB2[x] = frand(); }      // Create new random color
        for (int x=0; x<3; x++) {
            INC[x] = (RGB1[x] - RGB2[x]) / 25;              // Determine increments
        }
        for (int s=0; s<25; s++) {                          // Send color to thread2
            message_t *message = mbox.alloc();              // Allocate memory
            message->red = RGB1[0];
            message->green = RGB1[1];
            message->blue = RGB1[2];
            mbox.put(message);                              // Send data as message
            Thread::wait(100);
            for (int x=0; x<3; x++) {RGB1[x] -= INC[x];} // Approach to second colour
          }
      }
    }
}
```

Lab08_rtos_mailbox futási eredmény

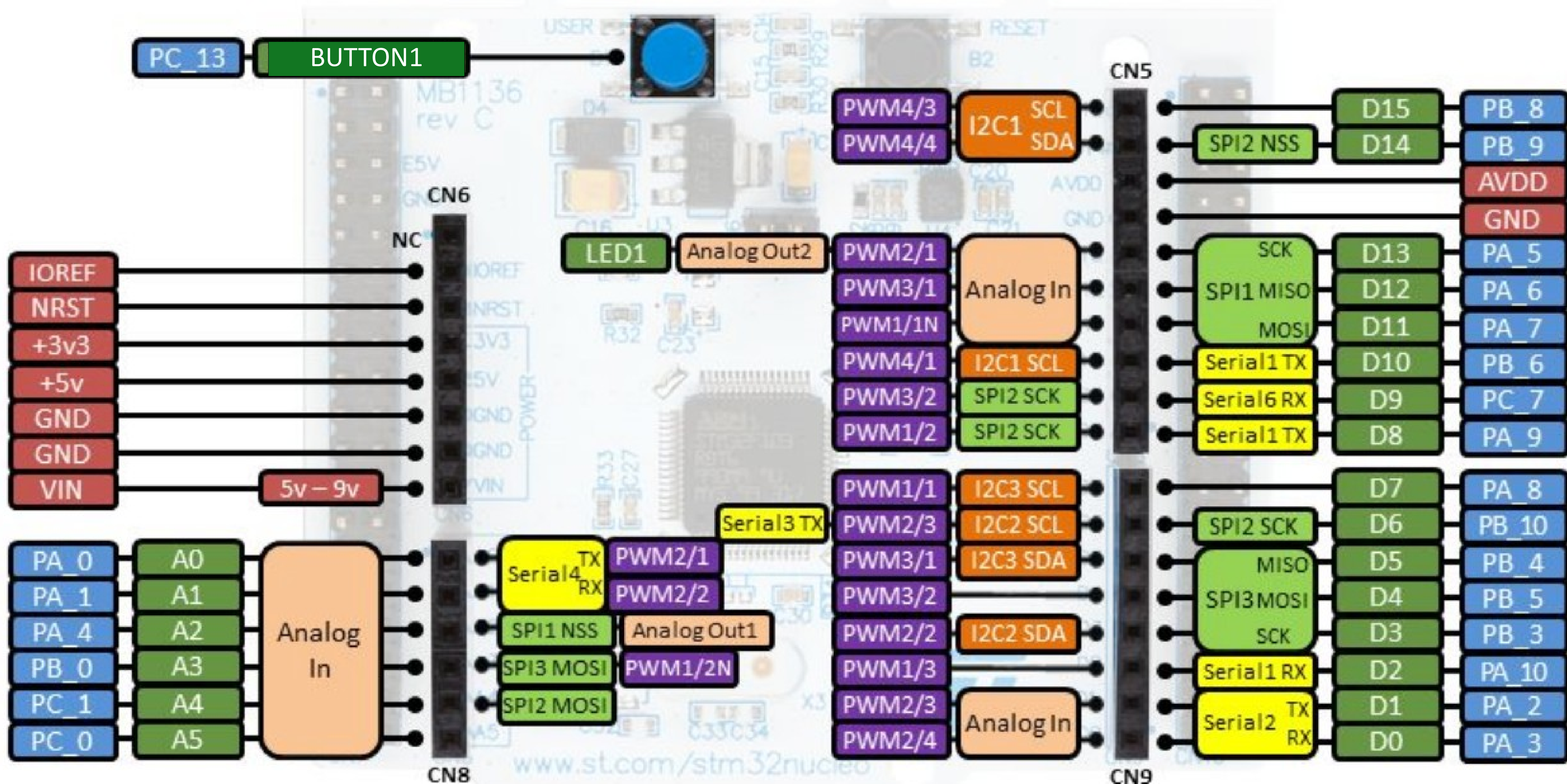
- A három ellenállással egy tüskesorra forrasztott LED könnyen csatlakoztatható a kártyához…



Arduino kompatibilis kivezetések

Nucleo F446RE
Arduino Headers

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



Morpho kivezetések