Beágyazott real-time operációs rendszerek

Naszály Gábor
<naszaly@mit.bme.hu>

Méréstechnika és Információs Rendszerek Tanszék

Fogalmak

Beágyazott operációs rendszerek

- A beágyazott rendszerek operációs rendszerei… ☺

Real-time operációs rendszerek

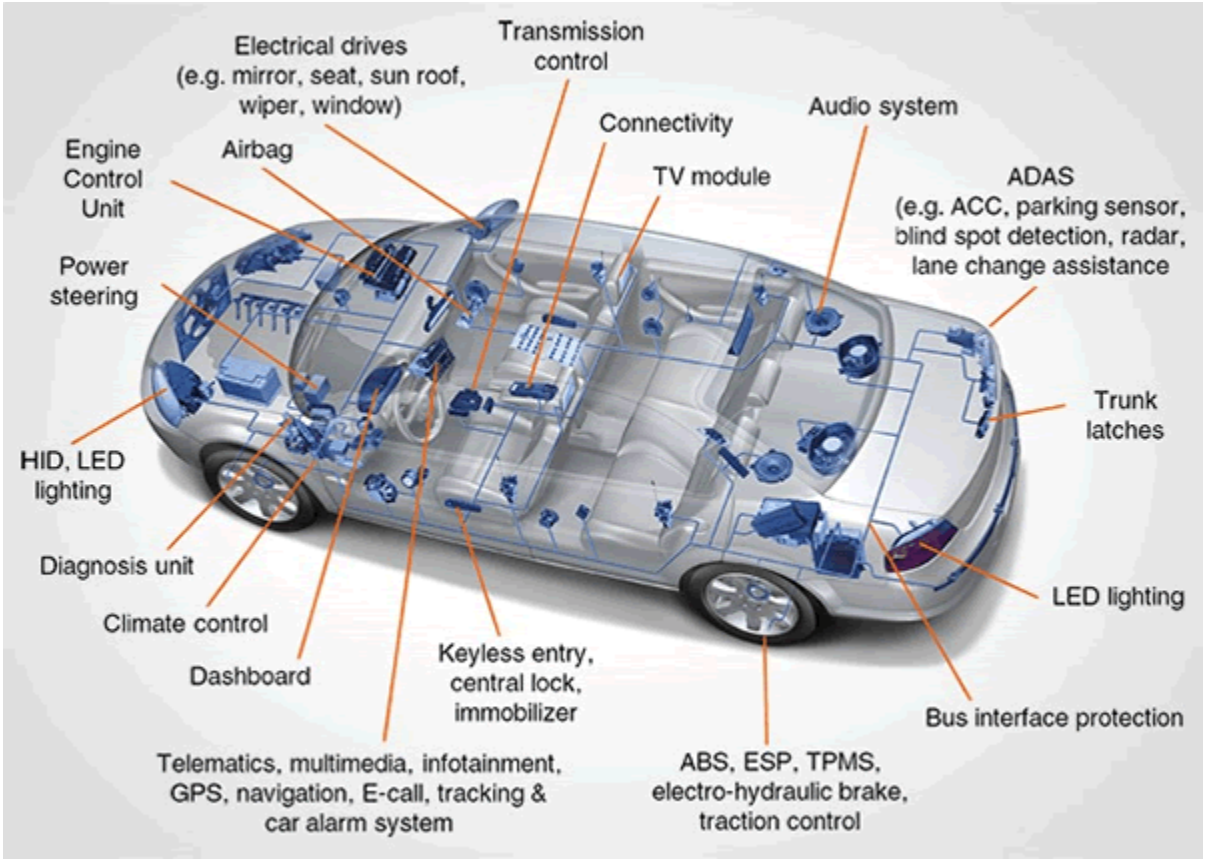
- A real-time rendszerek operációs rendszerei… ☺

Beágyazott rendszerek

- Pongyola definíció:
  - „Minden olyan számítógépes rendszer, ami nem egy hagyományos értelemben vett számítógép.”

Beágyazott rendszerek

- Alkalmazási területek – Autóiparipar



Beágyazott rendszerek

- Alkalmazási területek – Szórakoztató elektronika

Beágyazott rendszerek

- Alkalmazási területek – Orvosi berendezések

Beágyazott rendszerek

- Alkalmazási területek – Háztartási berendezések

Beágyazott rendszerek

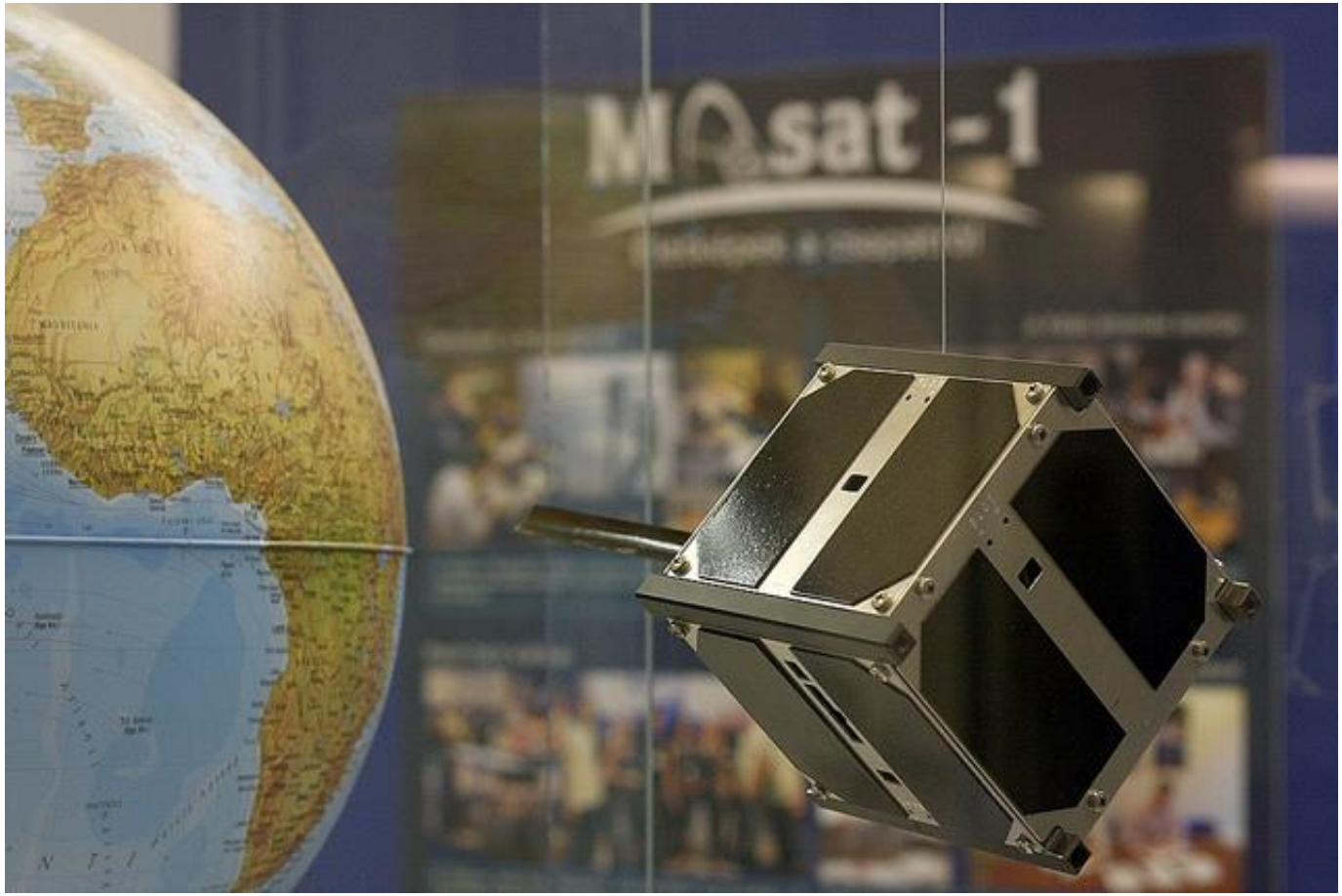
- Alkalmazási területek – Mérőberendezések

Beágyazott rendszerek

- Alkalmazási területek – Hálózati berendezések

Beágyazott rendszerek

- Alkalmazási területek – Űrkutatás



Beágyazott rendszerek

- Pontosabb definíció:
- Olyan **speciális számítógépes rendszerek**, amelyeket egy **jól meghatározott feladatra** találtak ki
- Ezen feladat ellátása érdekében **a külvilággal intenzív információs kapcsolatban állnak**
  - Található bennük valamilyen **feldolgozó egység**
  - Érzékelik a külvilág bizonyos paramétereit (**szenzorok**), és gyakran be is avatkoznak abba (**beavatkozók**)
  - A gépi komponensek között különféle **kommunikációs interfészek**en, protokollokon keresztül áramlik az információ
  - Biztosítanak valamilyen **felhasználói felületet** (humán operátor számára kijelzők, kezelő szervek)

Beágyazott rendszerek

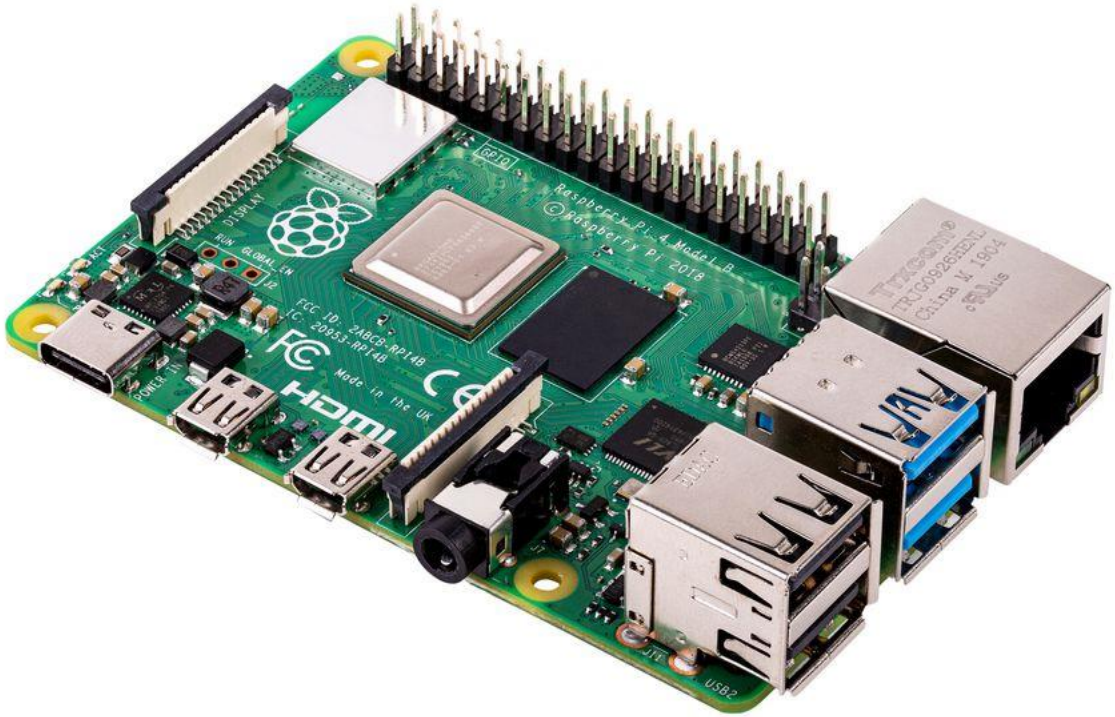
- A klasszikus értelemben vett beágyazott rendszerek általában nem személyi számítógépek.
- De vannak kivételek…

Beágyazott rendszerek

- Nem klasszikus értelemben vett (high-end) beágyazott rendszerek – Ipari PC-k

Beágyazott rendszerek

- Nem klasszikus értelemben vett (high-end) beágyazott rendszerek – Kártya méretű PC-k



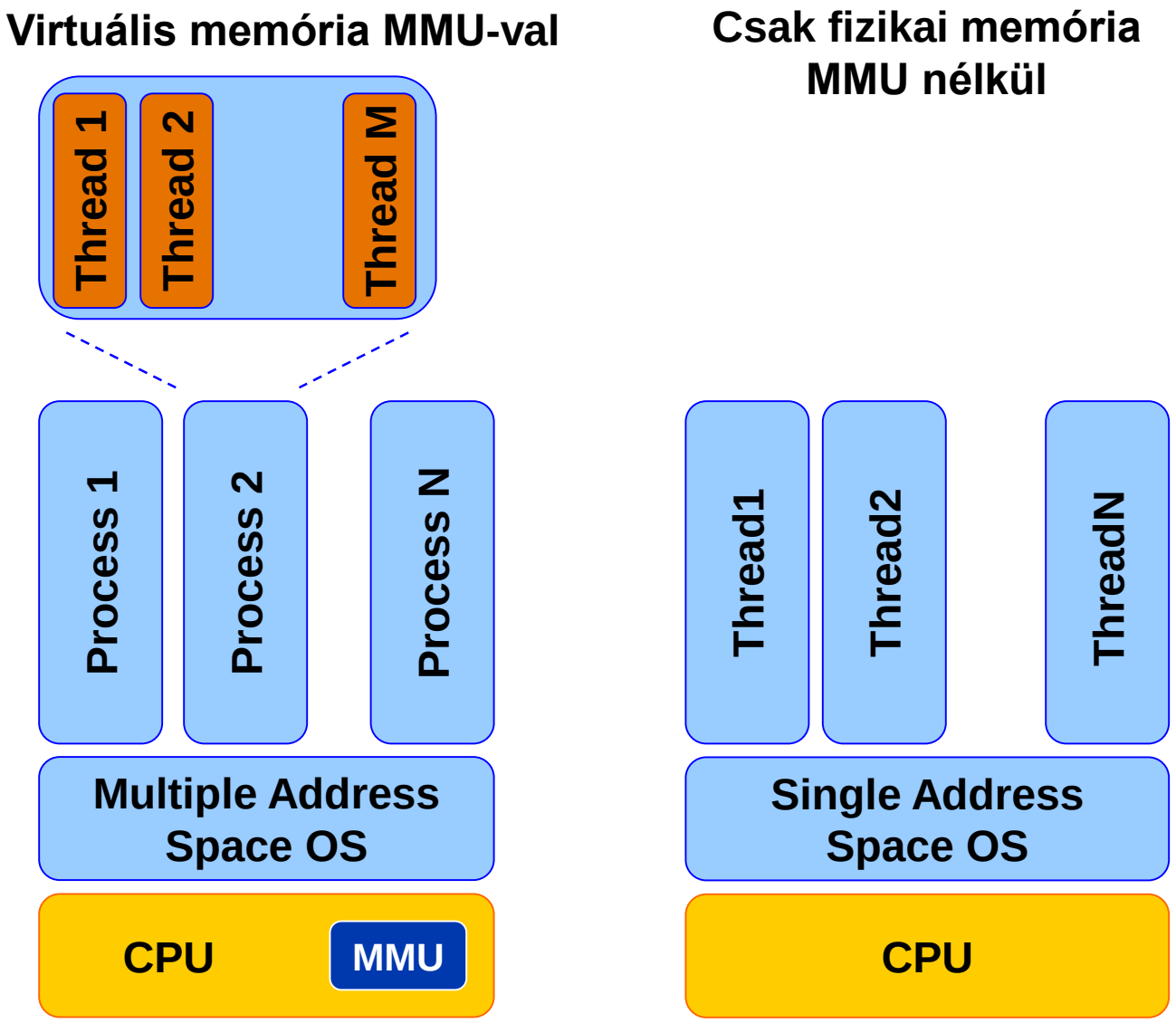
Beágyazott rendszerek

- Nem klasszikus értelemben vett (high-end) beágyazott rendszerek – Okostelefon?

Beágyazott rendszerek

	Klasszikus	High-end
Processzor órajel	~10 – x100 MHz	~ xGHz
Processzor védelmi szintek	Nincs	Van
MMU (virtuális tárkezelés)	Nincs	Van
Folyamatok, szálak?	Csak szálak	Folyamatok (és bennük szálak)
Linux futhat rajtuk?	Nem	Igen

Beágyazott rendszerek



Beágyazott rendszerek

- **Rengeteg belső periféria** a CPU-n kívül:
  - Memóriák: program (flash), adat (SRAM)
  - Analóg-Digitál, Digitál-Analóg átalakítók (ADC, DAC)
  - Időzítő áramkörök (timers)
  - Kommunikációs interfészek
    - Egyszerűbbek: U(SART), I2C, SPI
    - Bonyolultabbak: USB (device, host), Ethernet (100M)
    - Terület specifikus: CAN, LIN, FlexRay (ezek autóipari buszok)
  - Általános célú I/O (General Purpose I/O, GPIO)
    - Egyszerű beavatkozások (pl. LED kigyújtása, nyomógomb beolvasása)
    - HW-ből nem támogatott kommunikáció SW-es megvalósítása

Real-time rendszerek

- A rendszer adott eseményre adott időn belül válaszol
- Az adott időnek nem feltétlenül kell kicsinek lennie (bár gyakran az)

Real-time rendszerek

- Csoportosításuk (szigorú módszer)
  - Egy rendszer vagy real-time (azaz betartja a határidőket), vagy nem, nincs középút. (Lásd jog: pl. nem szabad elvenni más tulajdonát, nincs olyan, hogy „csak kicsit loptam”.)

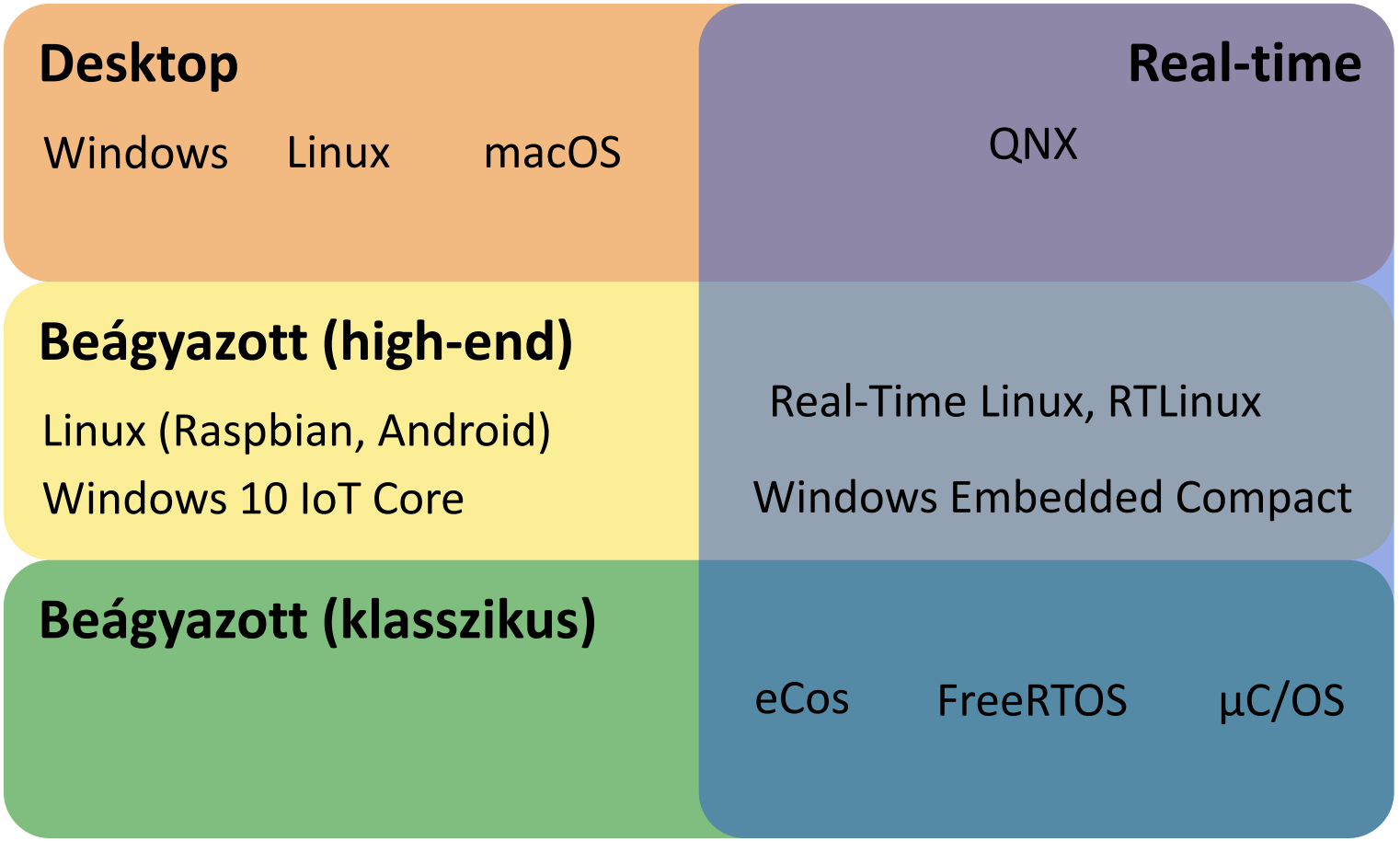
Real-time rendszerek

- Csoportosításuk (megengedőbb módszer)
  - Hard real-time rendszer
    - Mindent meg kell tennünk, hogy a rendszer a határidőket 1 valószínűséggel tudja tartani.
    - A határidő elmulasztása katasztrófát okozhat (lásd légiközlekedés).
  - Soft real-time rendszer
    - Próbáljuk úgy megvalósítani a rendszert, hogy közel 1 valószínűséggel tudjon határidőn belül válaszolni a kérésekre.
    - A határidő elmulasztása káros, de nem végzetes. (Pl. az 5 másodperces azonnali átutalás 1 óra múlva teljesül.)

Beágyazott és real-time rendszerek

- A legtöbb (klasszikus értelemben vett) beágyazott rendszer real-time követelményeknek kell, hogy eleget tegyen.
- Így a két fogalom metszete nagy.

Operációs rendszerek csoportosítása



Real-time operációs rendszerek

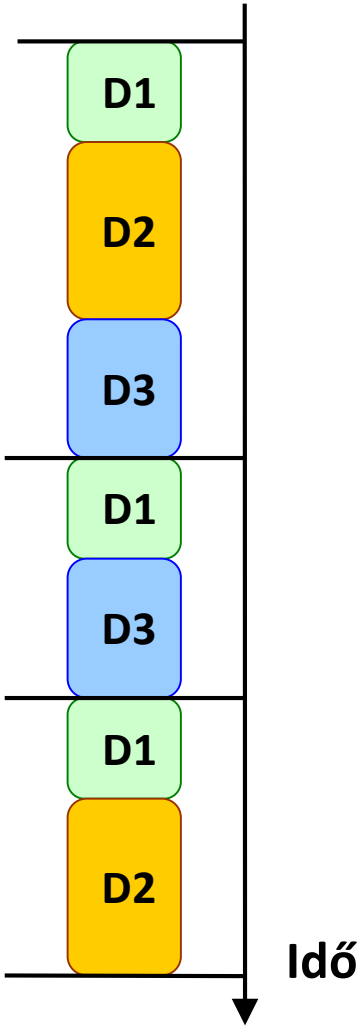
A klasszikus értelemben vett beágyazott rendszerekhez

Út az RTOS (mint szoftver architektúra) felé

- Kezdetben assembly nyelven programozták a beágyazott rendszereket
- Később a C vált uralkodóvá (csak az maradt assemblyben, amit nem lehetett máshogy megvalósítani, vagy így volt hatékonyabb)
- Kezdetben nem használtak RTOS-t (ma sem mindig). A feladat bonyolultsága és egyéb követelmények szabják meg, végül milyen SW architektúrát használunk.
- Nézzünk meg egy párat a legegyszerűbbtől kezdve

Ciklikus programszervezés

```
void main(void) {

  while(1) {

    if ( !! Device 1 needs service ) {
       !! Handle Device 1 and its data
    }

    if ( !! Device 2 needs service ) {
       !! Handle Device 2 and its data
    }

    if ( !! Device 3 needs service ) {
       !! Handle Device 3 and its data
    }

    ...

  }

}
```



Ciklikus programszervezés

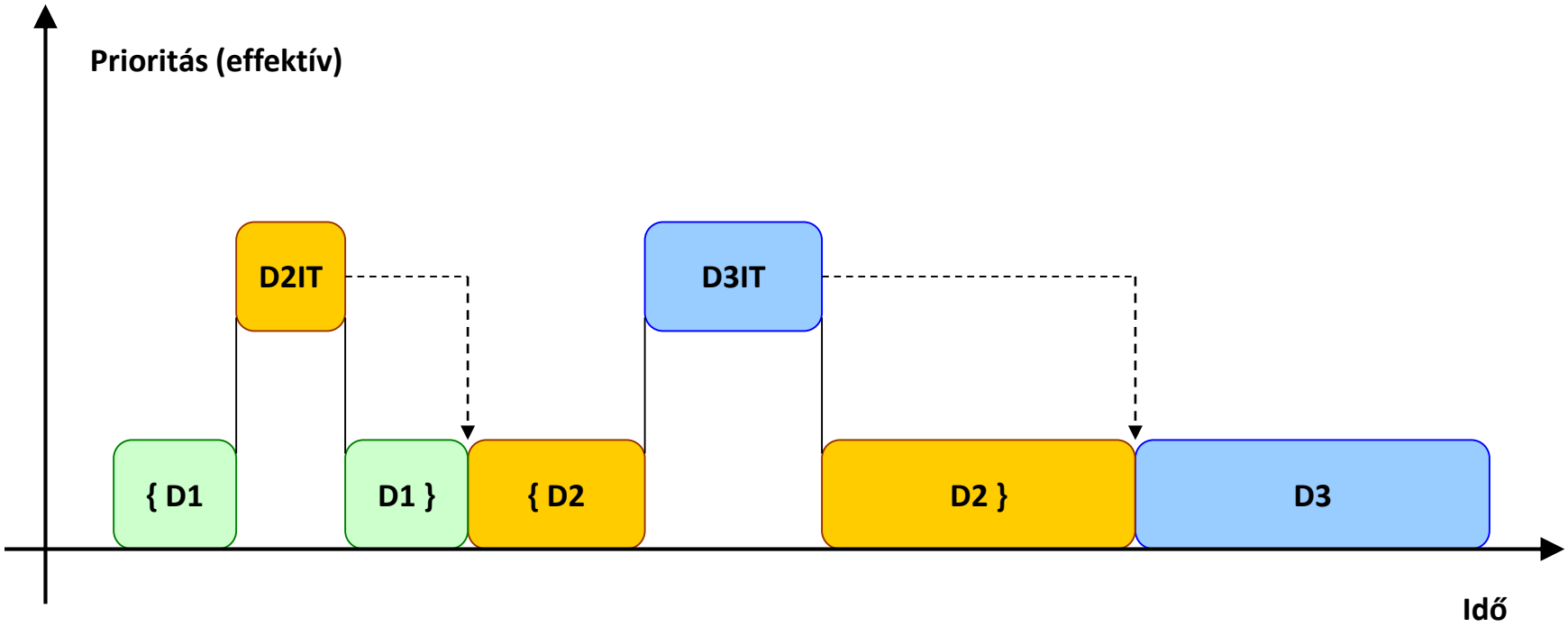
- A legegyszerűbb (nincs megszakítás, nincs közös erőforrás probléma)
- Nagy a szórása a válaszidőnek
- Worst-case válaszidő: a feladatok válaszidejeinek összege (azaz lineárisan nő a taszkok számával)
- Egyes feladatok tekintetében javítható a válaszidő, ha többször kérdezzük le őket a főhurokban
- „Törékeny” az architektúra (új taszk hozzáadása felboríthat mindent)

Ciklikus programszervezés (IT-kkel)

```
BOOL Device1_flag = FALSE;
BOOL Device2_flag = FALSE;
BOOL Device3_flag = FALSE;

void interrupt Device1_ISR (void) {
   !! Handle Device 1 time critical part
   Device1_flag = TRUE;
}

void interrupt Device2_ISR (void)  {
   !! Handle Device 2 time critical part
   Device2_flag = TRUE;
}

void interrupt Device3_ISR (void)  {
   !! Handle Device 3 time critical part
   Device3_flag = TRUE;
}
```

```
void main (void)
{
  while(1)
  {
     if ( Device1_flag ) {
       Device1_flag = FALSE;
          !! Handle Device 1 and its data
     }

     if (Device2_flag ) {
       Device2_flag = FALSE;
          !! Handle Device 2 and its data
     }

     if (Device3_flag ) {
       Device3_flag = FALSE;
          !! Handle Device 3 and its data
     }

     ...
  }
}
```

Ciklikus programszervezés (IT-kkel)



Ciklikus programszervezés (IT-kkel)

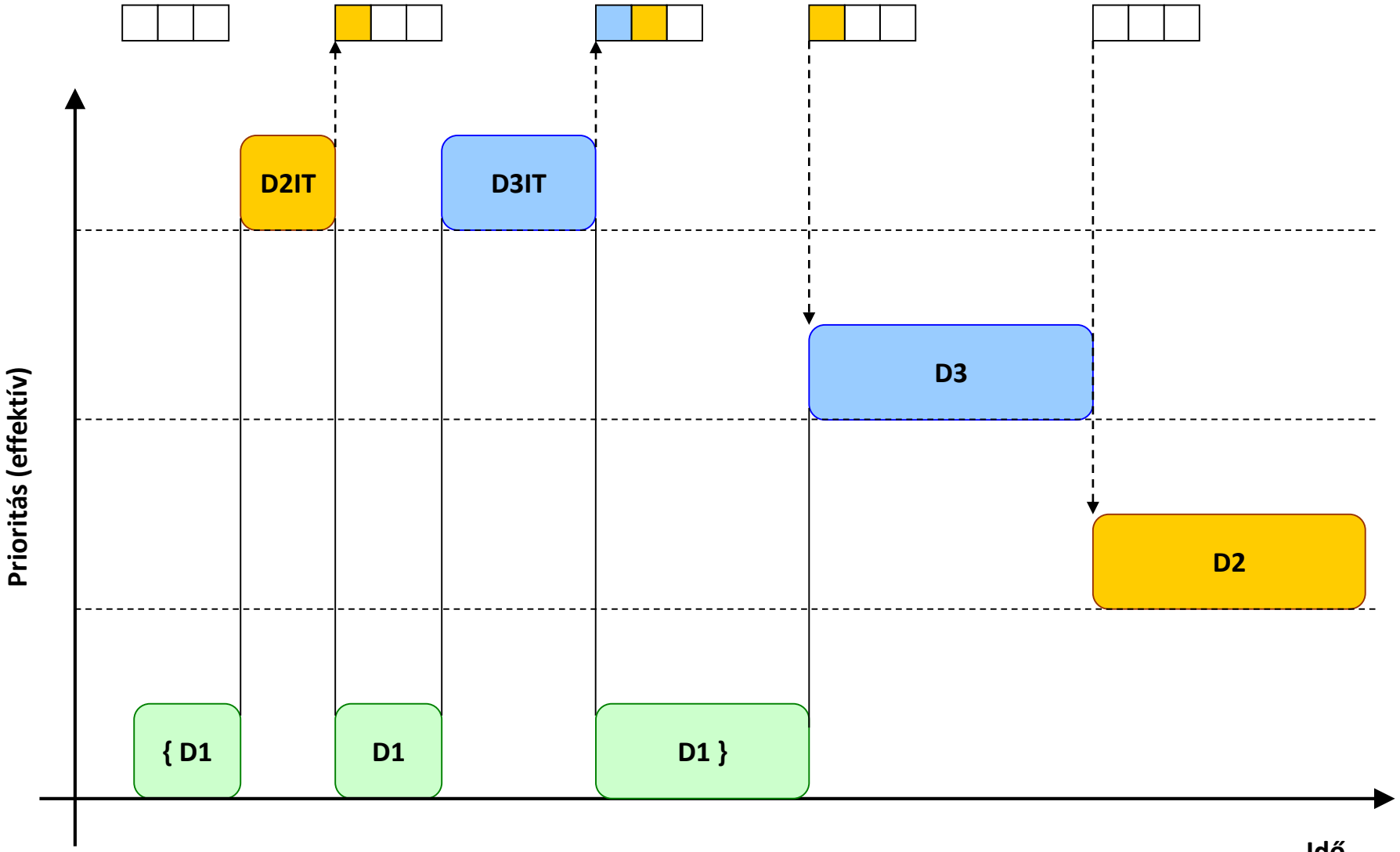
- Kicsit jobban kezeli az időkritikus részeket (a megszakítások magasabb effektív prioritáson futnak)
- Ha lehetséges interruptokhoz prioritást rendelni, még tovább lehet finomítani az architektúrát
- Jelentkezhet a közös erőforrások problémája
- Nagy a szórása a válaszidőnek
- Worst-case válaszidő: az összes feladat válaszideje + a megszakítások (arányos a taszkok számával)
- Egyes feladatok tekintetében javítható a válaszidő, ha többször kérdezzük le őket a főhurokban
- „Törékeny” az architektúra (új taszk hozzáadása felboríthat mindent)

Függvény-sor alapú ütemezés

```
!! Define queue of function pointers

void interrupt Device1 (void)
{
   !! Handle Device 1 time critical part
   !! Put Device1_func to call queue
}

void interrupt Device2 (void)
{
   !! Handle Device 2 time critical part
   !! Put Device2_func to call queue
}

void interrupt Device3 (void)
{
   !! Handle Device 3 time critical part
   !! Put Device3_func to call queue
}
```

```
void main (void)
{
  while(1)
  {
     while(!! Function queue not empty)
              !! Call first from queue
  }
}

void Device1_func (void)
{   !! Handle Device 1   }

void Device2_func (void)
{   !! Handle Device 2   }

void Device3_func (void)
{   !! Handle Device 3  }
```

Függvény-sor alapú ütemezés



Függvény-sor alapú ütemezés

- Képes a prioritások kezelésére (mind taszk, mind IT szinten)
- Jelentkezhet a közös erőforrások problémája az IT és a főprogram között
- Worst-case válaszidő (a legmagasabb prioritású feladatra) = a leghosszabb taszk válaszideje + IT
- A worst-case válaszidő nem nő lineárisan a taszkok számával
- A válaszidő jóval kevésbé szór
- Egy új feladat nem borítja fel az eddigi időzítést
- Nem preemptív

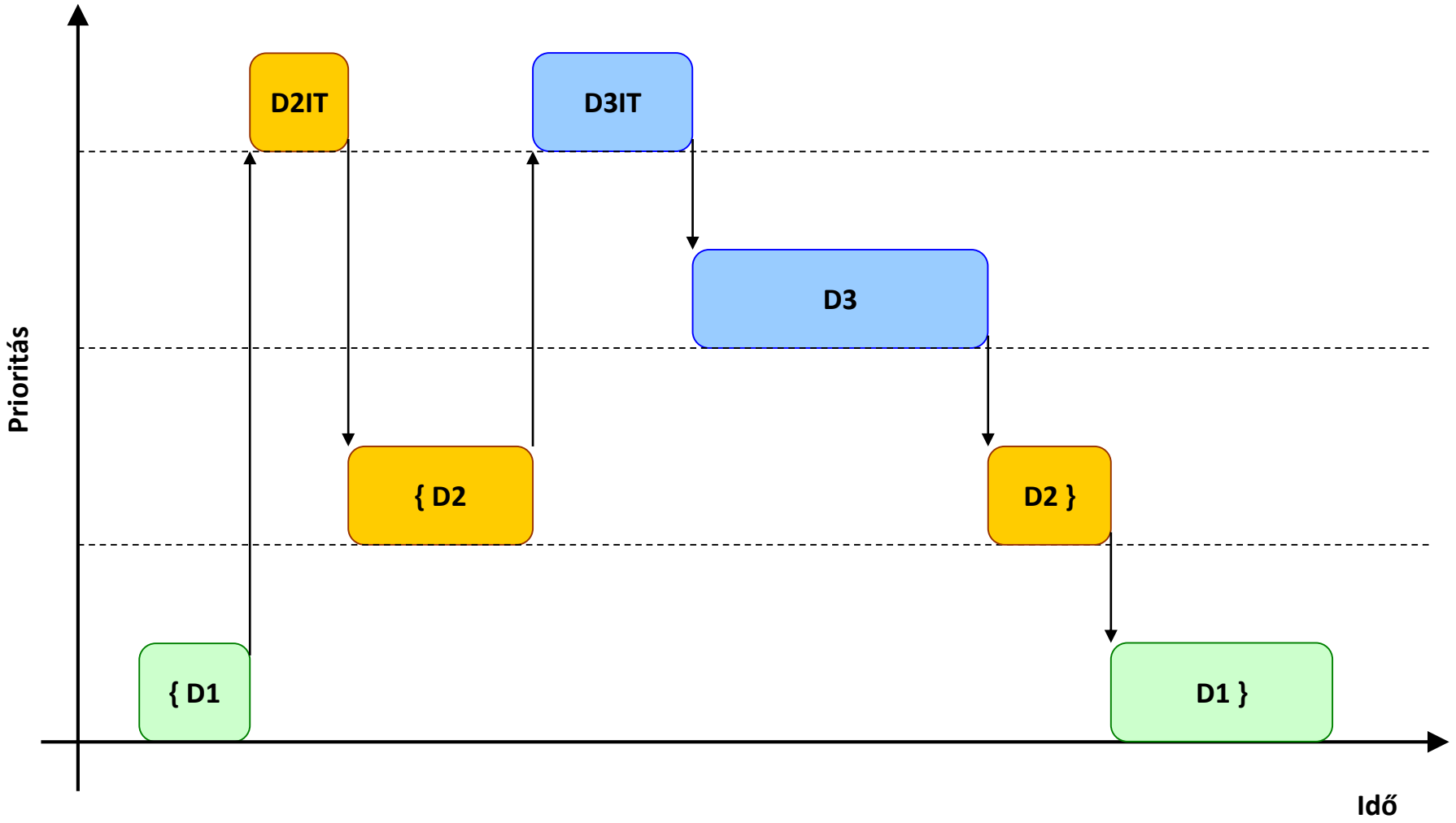
RTOS architektúra (preemptív)

```
void interrupt Device1 (void) {
   !! Handle Device 1 time critical part
   !! Set signal to Device1_task
}

void interrupt Device2 (void) {
   !! Handle Device 2 time critical part
   !! Set signal to Device2_task
}

void interrupt Device3 (void) {
   !! Handle Device 3 time critical part
   !! Set signal to Device3_task
}
```

```
void main (void) {
  !! Initialize OS
  !! Start OS scheduler
}
```

```
void Device1_task (void) {
  while (1) {
    !! Wait for signal to Device1_task
    !! Handle Device 1
  }
}

void Device2_task (void) {
  while (1) {
    !! Wait for signal to Device2_task
    !! Handle Device 2
  }
}

void Device3_task (void) {
  while (1) {
    !! Wait for signal to Device3_task
    !! Handle Device 3
  }
}
```

RTOS architektúra (preemptív)



RTOS architektúra (preemptív)

- Képes a prioritások kezelésére (mind taszk, mind IT szinten)
- Jelentkezhet a közös erőforrások problémája az IT és a főprogram, valamint az egyes taszkok között is
- Worst-case válaszidő (a legmagasabb prioritású feladatra) = a taszk váltási idő + IT (mindkettő kicsi)
- A worst-case válaszidő nem nő az új feladatok hozzáadásával
- A válaszidő szórása nagyon alacsony
- Kód overhead

RTOS architektúra (preemptív)

- Tipikus képviselők ebben a kategóriában
  - FreeRTOS (eredetileg Real Time Engineering Ltd., ma Amazon Web Services)
  - µC/OS-II, µC/OS-III (eredetileg Micriµm, ma Silabs)
  - eCOS
  - Keil RTX
  - Freescale/NXP MQX
  - Express Logic ThreadX
  - TI DSP/BIOS ás TI-RTOS
  - és sokan mások…

FreeRTOS

Története

- A FreeRTOS >15 éves múltra tekint vissza:
  - Eredetileg **Richard Barry** kezdte el fejleszteni 2003 körül
  - Később Richard cége, a **Real Time Engineers Ltd.** folytatta a fejlesztését
  - 2017-ben a projektet az **Amazon Web Services** (AWS) vette át
  - Richard továbbra is dolgozik a projekten az azt gondozó AWS csapat tagjaként

Licencelés

- MIT Open Source License:
  - Ingyenes és nyílt forrású
  - Kereskedelmi alkalmazásban használható
  - Jogdíj mentes
  - Technikai támogatás van, de fizetős
  - Garancia nincs
  - Jogi védelem nincs
- Megjegyzés:
  - A 10-es verzió előtt a GNU GPLv2 licence egy módosított (könnyítést tartalmazó) verziója volt érvényben.

A FreeRTOS család tagjai

Amazon Web Services:

Amazon FreeRTOS (a:FreeRTOS)

HighIntegritySystems:

OPENRTOS®

SAFERTOS®

SAFERTOS CORE

A FreeRTOS család tagjai

OPENRTOS®

- A WITTENSTEIN high integrity systems áll mögötte
- A FreeRTOS egy kereskedelmi licencelésű változata
- Nem ingyenes
- Cserébe adnak garanciát és IP védelmet

A FreeRTOS család tagjai

SAFERTOS®

- **A WITTENSTEIN high integrity systems áll mögötte**
- **A FreeRTOS funkcionális modelljén alapszik, de nem egyezik meg vele.**

- **Teljes mértékben átdolgozták és az alapoktól építették fel biztonság kritikus elveket szem előtt tartva. Nincs dinamikus memória foglalás, alaposabb paraméter ellenőrzés OS hívásoknál.**

A FreeRTOS család tagjai

Amazon FreeRTOS (a:FreeRTOS)

- Az Amazon Web Services fejleszti
- FreeRTOS kernel + egyéb szoftver csomagok:
  - Kommunikáció (pl. TCP/IP, MQTT)
  - Titkosítás (pl. TLS)
- Cél: a beágyazott eszközök számára könnyű és biztonságos legyen akár az egymás közti, akár egy felhő felé a kommunikáció, illetve a távoli frissítés.
- MIT licenc

A FreeRTOS főbb jellemzői

- Forráskódja könnyű portolhatóságra lett tervezve (jelenleg hivatalosan több mint 35 beágyazott architektúrára támogatott)
- Tervezésénél fontos szempont volt a kis méret, egyszerűség és könnyű használat
- A forráskódhoz rengeteg demó alkalmazást mellékelnek a kezdeti lépések könnyebbé tételéhez

A FreeRTOS főbb jellemzői

- Az ütemezés egysége:
  - Taszk (a legelterjedtebb)
  - Ko-rutin (taszknál egyszerűbb eszköz, legacy)
  - Hibrid (taszk + ko-rutin)
- A taszkok ütemezése:
  - Prioritásos
  - Preemptív (alapértelmezetten)
  - Round-robin time-slicing segítségével az azonos prioritási szinteken (ez az alapértelmezett)

Fontosabb OS szolgáltatások

- **taszkok** (tasks)
- **bináris szemaforok** (binary semaphores)
- **mutexek** (mutexes)
- **sorok** (queues)
- **esemény jelző bitek** (event groups /or event flags/)
- **szoftver időzítők** (software timers)
- **memória kezelés** (memory management)

Ritkábban használt OS szolgáltatások

- **ko-rutinok** (co-routines)
- **óraütés nélküli üresjárás** (tickless idle mode)
- **számláló szemaforok** (counting semaphores)
- **rekurzív mutexek** (recursive mutexes)
- **közvetlen taszk értesítések** (direct to task notifications)
- **bájt folyam** és változó méretű **üzenet pufferek** (stream buffers és message buffers)
- **várakozás egyszerre több RTOS objektumra** (blocking on multiple RTOS objects)

Ritkábban használt OS szolgáltatások

- **kicsatolások az RTOS kódjából** (hook functions)
- **TLS** (thread local storage pointers)
- **verem túlcsordulás detektálás** (stack overflow detection)
- **nyomkövetés** (trace features)
- **futás idejű statisztikák** (run time statistics)
- **MPU támogatás** (memory protection support)

A FreeRTOS felépítése

Alkalmazás

FreeRTOSConfig.h

FreeRTOS: architektúra független kód

list.c	***stream_buffer.c***	**list.h**	***stream_buffer.h***
tasks.c		**task.h**	***message_buffer.h***
queue.c		**queue.h**	**semphr.h**
event_groups.c		***event_groups.h***	
timers.c		***timers.h***	
croutine.c		***croutine.h***	**FreeRTOS.h**

FreeRTOS: architektúra függő kód

port.c **portmacro.h**

FreeRTOS: mem. man.

heap_(1-5).c

Szoftver

CPU

Időzítő

Hardver

Konfigurációs lehetőségek

- **`FreeRTOSConfig.h`** fejléc fájlban számos `#define` sor található
  - Vannak ki-/bekapcsolható funkciók, pl.:
    - `#define configUSE_PREEMPTION ( 1 )`
    - `#define configUSE_MUTEXES ( 1 )`
    - `#define configIDLE_SHOULD_YIELD ( 0 )`
    - `...`
  - Vannak számszerűsíthető konfigurációs opciók is, pl.:
    - `#define configTICK_RATE_HZ ( 100 )`
    - `#define configMAX_PRIORITIES ( 6 )`
    - `...`

Konfigurációs lehetőségek

- Az OS kódja használja a konfigurációs definíciókat:

```
/* A task being unblocked cannot cause an immediate
context switch if preemption is turned off. */
#if ( configUSE_PREEMPTION == 1 )
{
    /* Preemption is on, but a context switch should
    only be performed if the unblocked task has a
    priority that is equal to or higher than the
    currently executing task. */
    if( pxTCB->uxPriority >= pxCurrentTCB->uxPriority )
    {
        xSwitchRequired = pdTRUE;
    }
    else
    {
        mtCOVERAGE_TEST_MARKER();
    }
}
#endif /* configUSE_PREEMPTION */
```

Részlet a tasks.c forrás fájlból

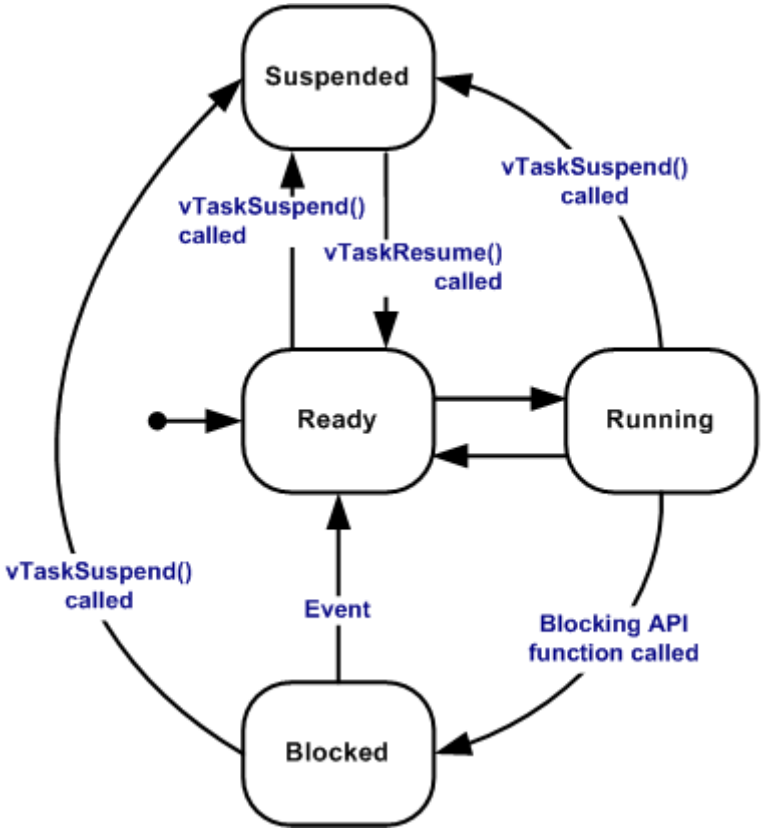
Konfigurációs lehetőségek

- Az OS kódja használja a konfigurációs definíciókat:

```
/* Setup the systick timer to generate the tick interrupts at the
 * required frequency. */
__attribute__(( weak )) void vPortSetupTimerInterrupt( void )
{
/* Calculate the constants required to configure the tick interrupt */

    /* Stop and clear the SysTick. */
    portNVIC_SYSTICK_CTRL_REG = 0UL;
    portNVIC_SYSTICK_CURRENT_VALUE_REG = 0UL;

    /* Configure SysTick to interrupt at the requested rate. */
    portNVIC_SYSTICK_LOAD_REG =
        ( configSYSTICK_CLOCK_HZ / configTICK_RATE_HZ ) - 1UL;
    portNVIC_SYSTICK_CTRL_REG =
        ( portNVIC_SYSTICK_CLK_BIT |
          portNVIC_SYSTICK_INT_BIT |
          portNVIC_SYSTICK_ENABLE_BIT );
}
```

Részlet a /portable/GCC/ARM_CM3/port.c forrás fájlból (az olvashatóság kedvéért picit átszerkesztve)

Task állapotok



Tipikus alkalmazás

- **`main()`**:
  - Inicializálás
  - Taszkok létrehozása → `xTaskCreate()`
  - Az ütemező elindítása → `vTaskStartScheduler()`
    - Ez a függvény vissza már nem tér a main()-be. Az OS mindig valamelyik taszkot ütemezi. Ha egyik saját taszkunk sincs futásra kész állapotban, akkor a beépített idle taszk fog futni.

Tipikus alkalmazás

- Taszkok
  - Megvalósítás függvényekben
  - Kétféle taszk szervezés lehetséges:
    - Egyszeri lefutású:
      - A taszk egyszer fut le
      - Futása során kiválthat olyan eseményeket, amelyek más taszkok futását előidézik
      - A futása végén törli magát → `vTaskDelete()`
    - Végtelen ciklusú
      - Az elején lehet egy opcionális inicializációs szakasz
      - Utána egy végtelen ciklus található
      - Amiben el kell helyezni olyan OS hívást, aminek hatására várakozó állapotba kerül (hogy az alacsonyabb prioritásúakat ne éheztesse ki)

Egyszerű FreeRTOS példa alkalmazás

Demo

A megvalósított funkció

- Két taszk: egy magas és egy alacsony prioritású
- Végtelen ciklus szervezésűek
- A cikluson belül:
  - Kiírnak egy rövid szöveget a standard kimenetre („Hi” illetve „Lo”)
  - Majd időre várakozó állapotba teszik magukat (a magas prioritású 1, az alacsony fél másodpercre)

Az alkalmazás kódja

```c
int main(void) {
   [...] // Init stdio: use UART0

   xTaskCreate(
      prvTaskHi,
      "Hi",
      mainTASK_HI_STACK_SIZE,
      NULL,
      mainTASK_HI_PRIORITY,
      NULL);

   xTaskCreate(
      prvTaskLo,
      "Lo",
      mainTASK_LO_STACK_SIZE,
      NULL,
      mainTASK_LO_PRIORITY,
      NULL);

   vTaskStartScheduler();
   return 0; // Never reached
}
```

```c
static void prvTaskHi(void *pvParam) {
   while (1) {
      printf("Hi\n");
      vTaskDelay(configTICK_RATE_HZ);
   }
}

static void prvTaskLo(void *pvParam) {
   while (1) {
      printf("Lo\n");
      vTaskDelay(configTICK_RATE_HZ / 2);
   }
}
```

prvTaskHi, prvTaskLo: Függvény pointerek a taszkok kódjának otthont adó függvényekre.

mainTASK_HI_STACK_SIZE, mainTASK_LO_STACK_SIZE: A taszkok vermének méretét megadó konstansok. Fontos, hogy elegendően nagyok legyenek. Beágyazott környezetben a **printf()** relatíve nagy verem fogyasztó lehet...

mainTASK_HI_PRIORITY, mainTASK_LO_PRIORITY: A taszkok prioritása: az alacsonyé 1, a magasé 2 (az idle taszk prioritása 0).

A felhasznált hardver

- A felhasznált eszköz a laborokon is szereplő Silicon Labs EFM32 Giant Gecko Starter Kit (STK3700):

Debugger áramköri rész (összeköttetés számítógéppel):

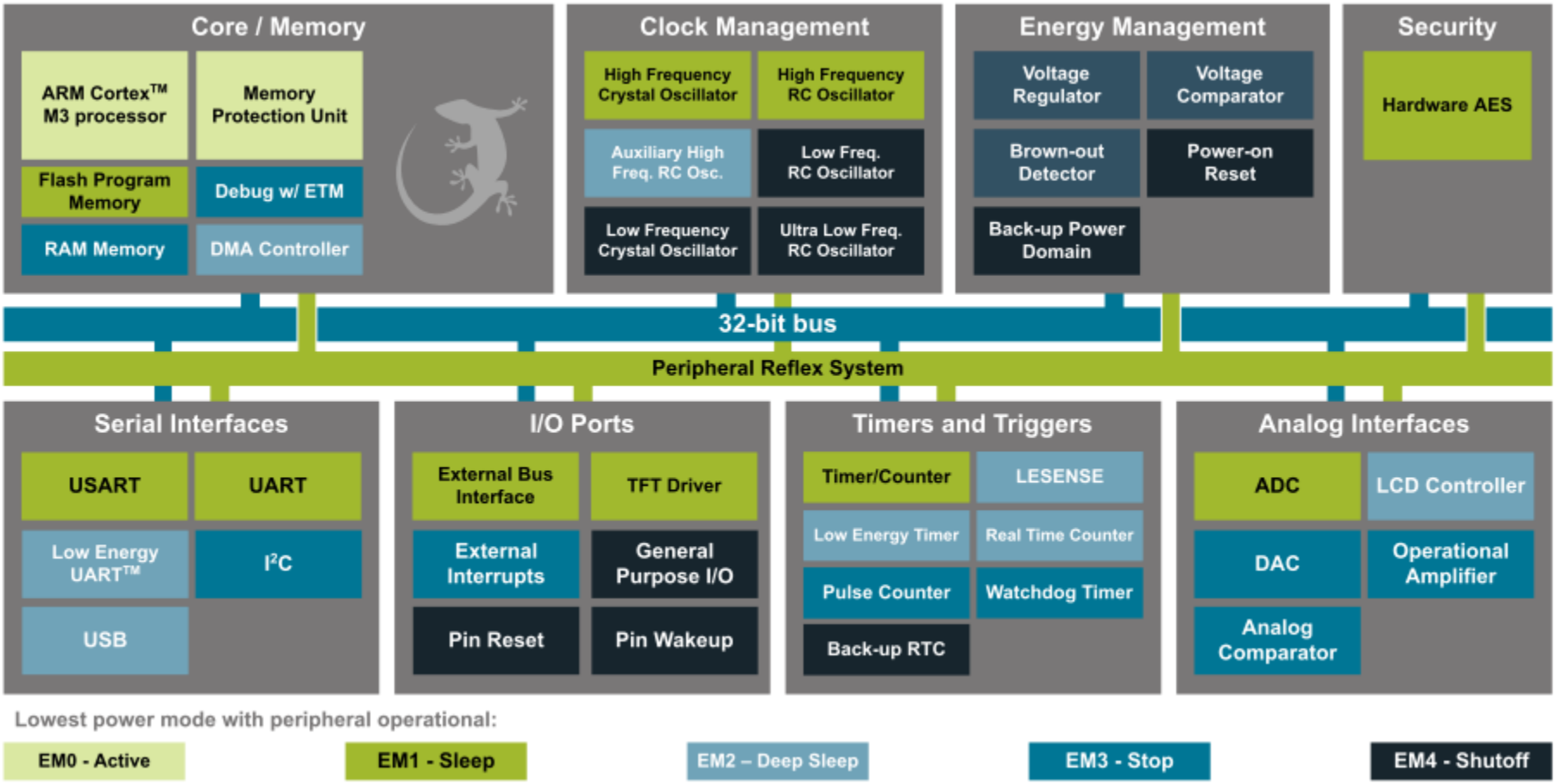
- Program letöltés a mikrovezérlőre, debuggolás
- A mikrovezérlő egyik UART perifériájának kommunikációját ráülteti az USB kapcsolatra (virtuális soros port a számítógépen)
- A mikrovezérlő egyik lehetséges táp forrása. Ha innét tápláljuk, megfigyelhető a felvett áram.

EFM32GG990F1024 mikrovezérlő:

- 32-bites mag (ARM Cortex-M3)
- 1 MiB program memória (flash)
- 128 KiB adat memória (SRAM)

A felhasznált hardver

- A kártyán szereplő mikrovezérlő egyszerűsített blokk diagramja:



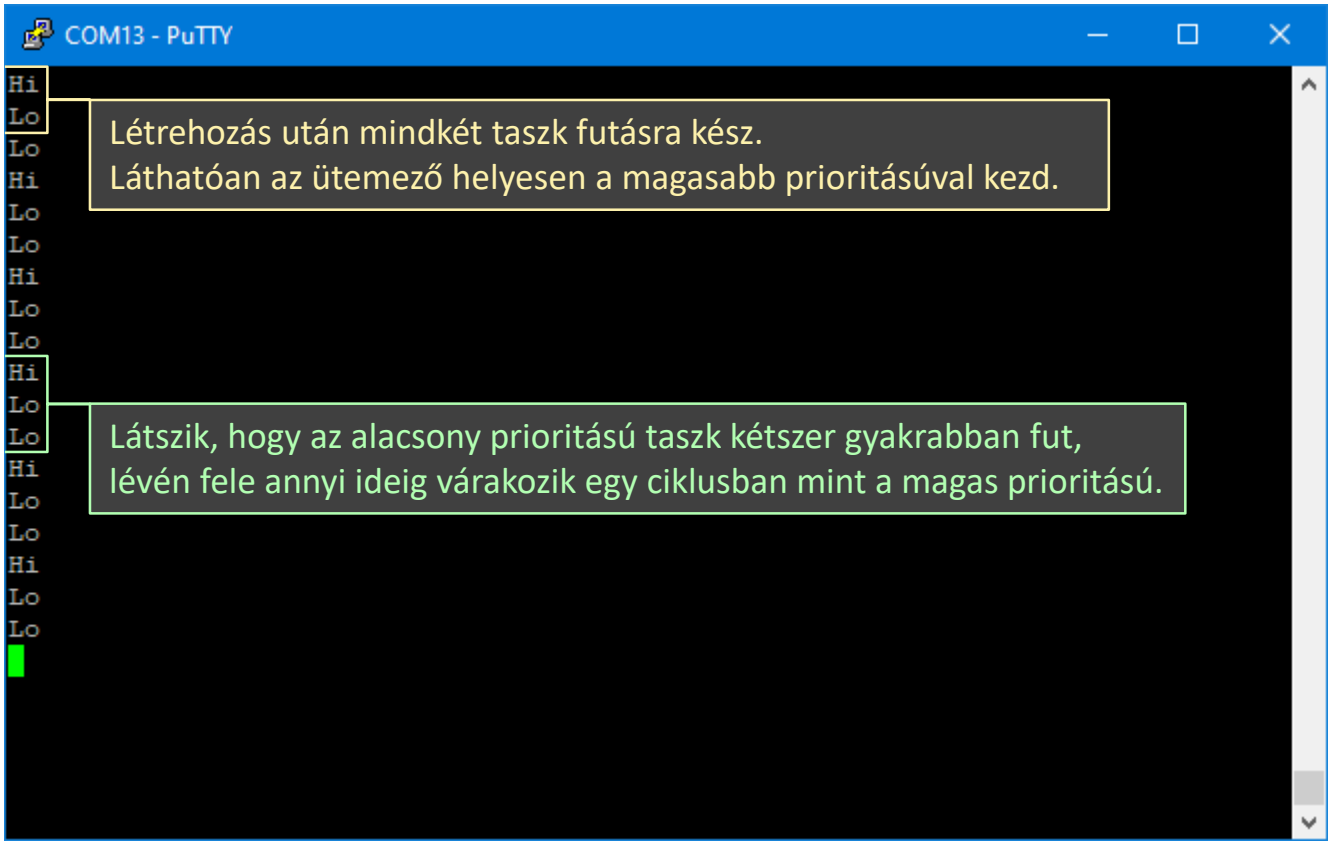
A felhasznált hardver

- A példa alkalmazás az alábbiakat használja a mikrovezérlőből:
  - ARM Cortex-M3 (CPU) → program végrehajtás
  - Flash Memory → program memória a kódnak
  - RAM Memory → adat memória a változóknak
  - UART0 → egyszerű soros kommunikáció a C stdio számára (a `printf()` ide ír ki)
  - System Timer → az idő múlásának követésére (a FreeRTOS kezeli megszakításos módon; ez az egység az előző blokk diagramon közvetlenül nem látszik, az ARM Cortex-M3 beépített időzítője)

A felhasznált hardver

- A printf() útja a számítógép felé:
  - A beágyazott alkalmazás kimenete a mikrovezérlő egyik UART perifériáját használja karakterek küldésére.
  - Ez a debugger áramköri részhez érkezik, ami továbbítja azt a számítógép felé USB kapcsolaton keresztül.
  - A számítógépen egy driver virtuális soros portot hoz létre (Windows alatt COM<#> néven).
  - A COM portokra ún. terminál programok (mint pl. a PuTTY) képesek csatlakozni. Ezáltal a vett karaktereket megjeleníteni, és a legépelt karaktereket elküldeni (ezzel most a példa alkalmazásunk nem kezd semmit).

Az alkalmazás által generált kimenet



Az alkalmazás nyomon követése (trace)

- Számos RTOS (így a FreeRTOS is) lehetőséget biztosít a futásának megfigyelésére → trace (nyomkövetés) szolgáltatások
- Ez a gyakorlatban azt jelenti, hogy az OS számos pontján trace makrók találhatóak, pl.:
  - Amikor egy taszk elhagyja a FUT állapotot
  - Amikor egy taszk FUT állapotba kerül
  - Amikor megtörtént egy óraütés (System Timer megszakítás)
  - Amikor egy taszk elkezd várni egy szemaforra
  - …

Az alkalmazás nyomon követése (trace)

- Ezek a makrók alapértelmezetten üresek:

```
#ifndef traceTASK_SWITCHED_IN
    /* Called after a task has been selected to run.
       pxCurrentTCB holds a pointer to the task control block
       of the selected task. */

    #define traceTASK_SWITCHED_IN()

#endif
```

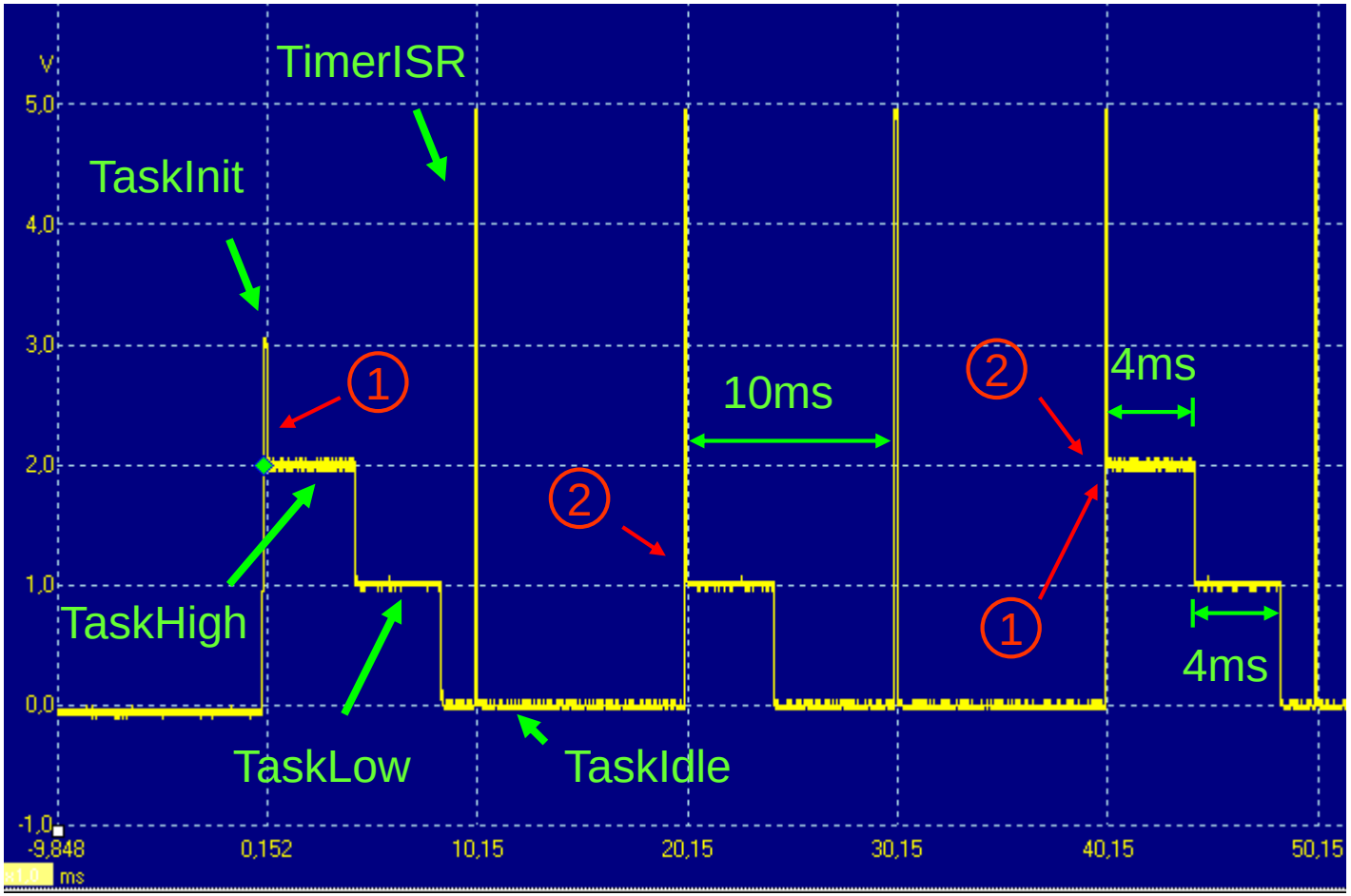
Részlet a FreeRTOS.h fejléc fájlból

- Látszik azonban, hogy felül lehet definiálni őket

Az alkalmazás nyomon követése (trace)

- Az egyik legegyszerűbb esetben a trace makrókat úgy lehet felüldefiniálni, hogy az alkalmazás egy analóg kimenetet mindig egy – az éppen futó taszk prioritásával arányos – feszültség szintre állítson be.
- Kiegészítés gyanánt a megszakításokhoz (mint pl. amilyen a legtöbb RTOS-ben lévő óraütés) is rendelhető egy – a taszkokhoz rendelteknél magasabb – feszültség szint
- A kimenetet egy oszcilloszkóppal megfigyelve követhető az alkalmazás futása

Az alkalmazás nyomon követése (trace)



Ez az ábra régebben készült egy másik (de hasonló) példa alkalmazással egy másik (de hasonló) operációs rendszer alatt (µC/OS-II). Az alkalmazás a mostanitól annyiban különbözik, hogy van benne egy harmadik, egyszeri lefutású taszk is (TaskInit). Ennek van a legnagyobb prioritása, és ez hozza létre a másik kettőt, majd törli magát.

1: olyan időpont, amikor több taszk is futásra kész, és az OS helyesen a legnagyobb prioritásút ütemezi

2: olyan időpont, amikor a megszakítás nem oda tér vissza, ahova beütött, hanem az OS átütemez

Az alkalmazás nyomon követése (trace)

- Léteznek kifinomultabb megoldások is egy analóg kimeneti láb feszültségének állítgatásán felül
- A trace makrókat megvalósíthatjuk úgy is, hogy az egyes eseményeket naplózzák, gyűjtsenek be róluk egyéb hasznos kiegészítő információkat.
- Majd ezeket valamilyen csatornán keresztül juttassák el egy számítógépnek, ahol egy analizátor alkalmazással ezeket jóval hatékonyabb módon elemezni tudjuk

Az alkalmazás nyomon követése (trace)

- Egy lehetséges ilyen tool a SEGGER SystemView

A beágyazott eszközön futó kód:

- Hozzá kell fordítani a saját alkalmazásunkhoz
- Minimális overhead-et jelent
- Feladata az eseményekről az információk begyűjtése, majd eltárolása egy bufferbe a RAM-ban
- A buffer tartalmát debug csatornán keresztül elérhetővé teszi egy számítógép számára

A számítógépen futó alkalmazás:

- Feladata a megfigyelt események megjelenítése

Az alkalmazás nyomon követése (trace)

- Példa egy SystemView által megvalósított FreeRTOS trace makróra:

```
#define traceTASK_SWITCHED_IN()                                          \
    if(prvGetTCBFromHandle(NULL) == xIdleTaskHandle) {                   \
        SEGGER_SYSVIEW_OnIdle();                                         \
    } else {                                                             \
        SEGGER_SYSVIEW_OnTaskStartExec((U32)pxCurrentTCB);               \
    }
```

Részlet a SEGGER_SYSVIEW_FreeRTOS.h fájlból (újra tördelve)

A SystemView által biztosított függvény

A FreeRTOS által biztosított makró, változó

Az alkalmazás nyomon követése (trace)

- Kövessük nyomon a példa alkalmazásunk futását a SEGGER SystemView segítségével!
- A következő diákon a futás különböző fázisait fogjuk megnézni.

Az alkalmazás nyomon követése (trace)

- Az alkalmazásunk az ideje nagy részét az Idle szálban tölti, mivel a saját szálaink 1 ill. fél másodpercenként aktívak csak egy rövid időre.
- Az Idle szálat a System Timer által generált megszakítás kérések hatására periodikusan lefutó megszakítás kezelő rutin (SysTick ISR) szakítja meg
- Az ISR-t az operációs rendszer valósítja meg. Így követi az idő múlását, és ha kell, átütemez.
- A mi esetünkben általában nem lesz átütemezés, az Idle szál megszakítása után oda is térünk vissza.

Az alkalmazás nyomon követése (trace)

108521	08:24.799 958 500	SysTick	ISR Enter	Runs for 36.7 us (514 cycles)	
108522	08:24.799 995 214	SysTick	ISR Exit	Returns to Idle	
108523	08:24.809 958 500	SysTick	ISR Enter	Runs for 36.7 us (514 cycles)	
108524	08:24.809 995 214	SysTick	ISR Exit	Returns to Idle	
108525	08:24.819 958 643	SysTick	ISR Enter	Runs for 36.7 us (515 cycles)	
108526	08:24.819 995 429	SysTick	ISR Exit	Returns to Idle	
108527	08:24.829 958 500	SysTick	ISR Enter	Runs for 36.7 us (514 cycles)	
108528	08:24.829 995 214	SysTick	ISR Exit	Returns to Idle	
108529	08:24.839 958 500	SysTick	ISR Enter	Runs for 36.7 us (514 cycles)	
108530	08:24.839 995 214	SysTick	ISR Exit	Returns to Idle	

Trigger Configuration | SEGGER SystemView V2.52d

Trigger on: Interrupt

SysTick

ISR Enter

Apply | OK | Cancel

Timeline Width: 2.0 ms

0 us -100 us 08:24.839 958 500 +100 us +200 us +300 us +400 us +500 us +600 us +700 us +800 us +900 us +1000

Unified

SysTick

Scheduler

Hi

Lo

Idle

Contexts

Name	Type	Stack Information	Run Count	Frequency	Last Run Time	Min Run Time	Max Run Time	Total Run Time
SysTick	#15		50484	100 Hz	0.0367 ms	0.0365 ms (#28975)	0.8142 ms (#82259)	1981.6580 ms
Scheduler			2525	5 Hz	0.0671 ms	0.0469 ms (#16783)	0.078 ms (#30111)	152.0080 ms
Hi	@2	50 @ 0x200001e8	505	1 Hz	0.3230 ms	0.3223 ms (#50748)	0.6929 ms (#101689)	163.2164 ms
Lo	@1	50 @ 0x20000478	1010	2 Hz	0.3230 ms	0.3223 ms (#9371)	0.3272 ms (#11418)	326.0202 ms
Idle			50484	100 Hz	9.9632 ms	8.6037 ms (#54193)	9.9635 ms (#1296)	502206.404 ms

Az alkalmazás nyomon követése (trace)

- Kicsit nagyobb időszeletet ábrázolva látható, hogy jelenleg a rendszer úgy van felkonfigurálva, hogy 10ms-es periódussal kövessék egymást a SysTick megszakítások

Az alkalmazás nyomon követése (trace)

7352	34.119 960 643	SysTick	ISR Enter	Runs for 36.7 us (514 cycles)
7353	34.119 997 357	SysTick	ISR Exit	Returns to Idle
7354	34.129 960 643	SysTick	ISR Enter	Runs for 36.7 us (514 cycles)
7355	34.129 997 357	SysTick	ISR Exit	Returns to Idle
7356	34.139 960 643	SysTick	ISR Enter	Runs for 36.7 us (514 cycles)
7357	34.139 997 357	SysTick	ISR Exit	Returns to Idle
7358	34.149 960 643	SysTick	ISR Enter	Runs for 36.7 us (514 cycles)
7359	34.149 997 357	SysTick	ISR Exit	Returns to Idle
7360	34.159 960 714	SysTick	ISR Enter	Runs for 36.7 us (514 cycles)
7361	34.159 997 429	SysTick	ISR Exit	Returns to Idle
7362	34.169 960 643	SysTick	ISR Enter	Runs for 36.7 us (514 cycles)

Trigger Configuration | SEGGER SystemView V2.52d

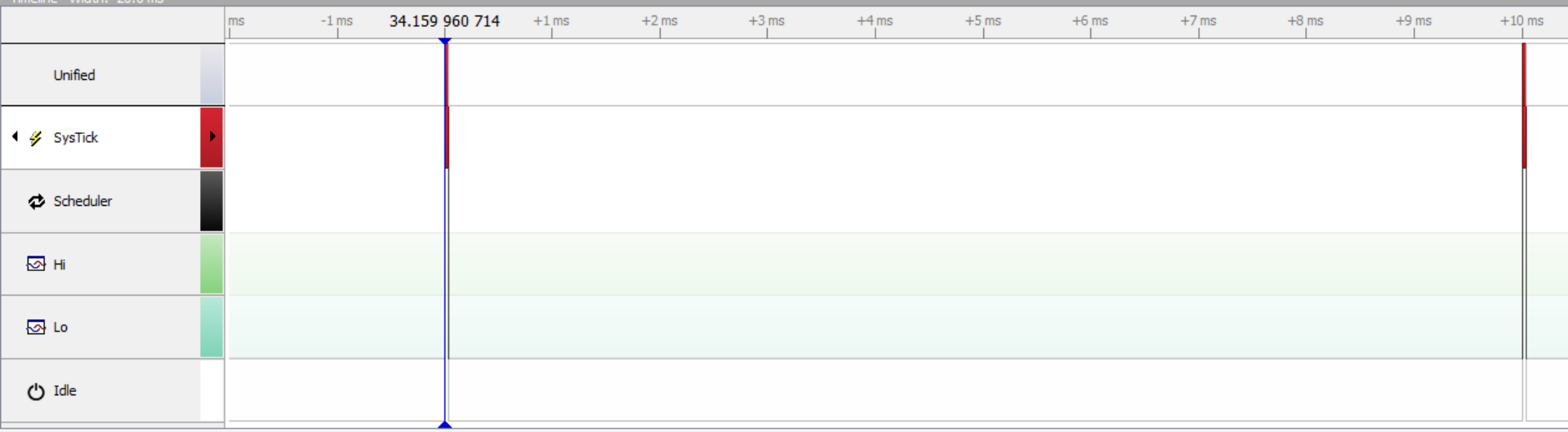
Trigger on: Interrupt

SysTick

ISR Enter

Apply | OK | Cancel

Timeline Width: 20.0 ms



Contexts

Name	Type	Stack Information	Run Count	Frequency	Last Run Time	Min Run Time	Max Run Time	Total Run Time	Run Time/s	
SysTick	#15		3416	100 Hz	0.0367 ms	0.0366 ms (#458)	0.8123 ms (#611)	135.1815 ms	3.8333 ms	0.38%
Scheduler			172	5 Hz	0.0670 ms	0.0469 ms (#3358)	0.0738 ms (#1641)	10.296 ms	0.3007 ms	0.03%
Hi	@2	50 @ 0x200001e8	34	1 Hz	0.3230 ms	0.3224 ms (#1858)	0.3271 ms (#5294)	10.6643 ms	0.3230 ms	0.03%
Lo	@1	50 @ 0x20000478	69	2 Hz	0.3230 ms	0.3223 ms (#6904)	0.3273 ms (#6262)	21.9715 ms	0.6461 ms	0.06%
Idle			3416	100 Hz	9.9633 ms	9.0206 ms (#5299)	9.9635 ms (#4083)	33971.1935 ms	994.8967 ms	99.49%

Az alkalmazás nyomon követése (trace)

- Kicsit érdekesebb az az eset, amikor valamelyik taszk felébred.
- Az egyszerűség kedvéért elsőre most egy olyan időpontot nézünk meg, amikor csak az alacsony prioritású szál ébred fel.
- A SysTick ISR-ben az OS most futásra kész állapotba teszi az alacsony prioritású szálat (ezt jelzi a szürke sáv, ekkor még nem fut).
- Az ISR után most az ütemező (fekete) fog futni, ami az Idle szálról átvált az alacsony prioritásúra

Az alkalmazás nyomon követése (trace)

- Miután az alacsony prioritású szál kiírta az üzenetét, ismét elteszi magát időre való várakozó állapotba (ezt az OS hívást szemlélteti a pici függőleges zöld vonalka)
- Az OS hívás hatására tehát az éppen futó szál várakozó állapotba kerül. Ezért az OS ebből a hívásból nem tér vissza azonnal az alacsony prioritású szálba, hanem az ütemező algoritmus fog futni, ami visszaadja a vezérlést az immár egyedüli futásra kész szálnak (Idle)

Az alkalmazás nyomon követése (trace)

65896	05:06.519 958 500	SysTick	ISR Enter	Runs for 91.7 us (1 285 cycles)
65897	05:06.520 006 857	SysTick	Task Ready	Lo, runs after 90.4 us (1 266 cycles)
65898	05:06.520 050 286	SysTick	ISR Exit	Returns to Scheduler
65899	05:06.520 097 286	Lo	Task Run	Runs for 323.0 us (4 523 cycles)
65900	05:06.520 372 214	Lo	vTaskDelay	xTicksToDelay=50
65901	05:06.520 420 357	Lo	Task Block	Lo, Wait for timeout
65902	05:06.520 487 500	Scheduler	System Idle	Idle for 9.471 ms (132 594 cycles)
65903	05:06.529 958 500	SysTick	ISR Enter	Runs for 36.7 us (514 cycles)
65904	05:06.529 995 214	SysTick	ISR Exit	Returns to Idle
65905	05:06.539 958 500	SysTick	ISR Enter	Runs for 36.7 us (514 cycles)
65906	05:06.539 995 214	SysTick	ISR Exit	Returns to Idle

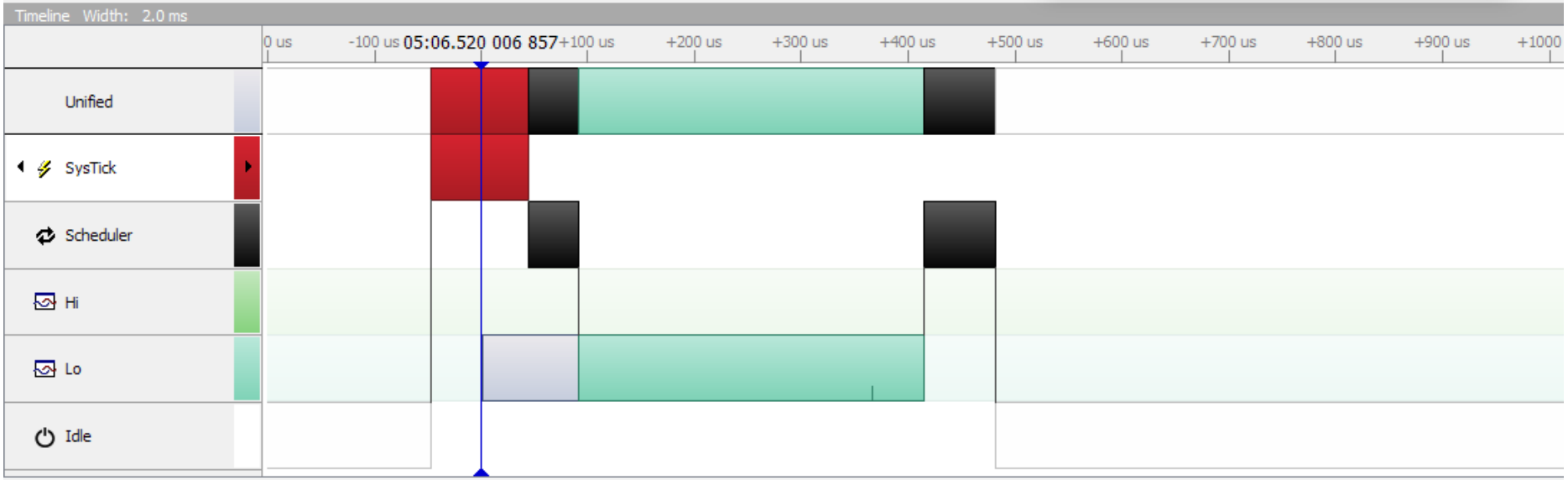
Trigger Configuration | SEGGER SystemView V2.52d

Trigger on: Task

Lo

Task Ready

Apply | OK | Cancel



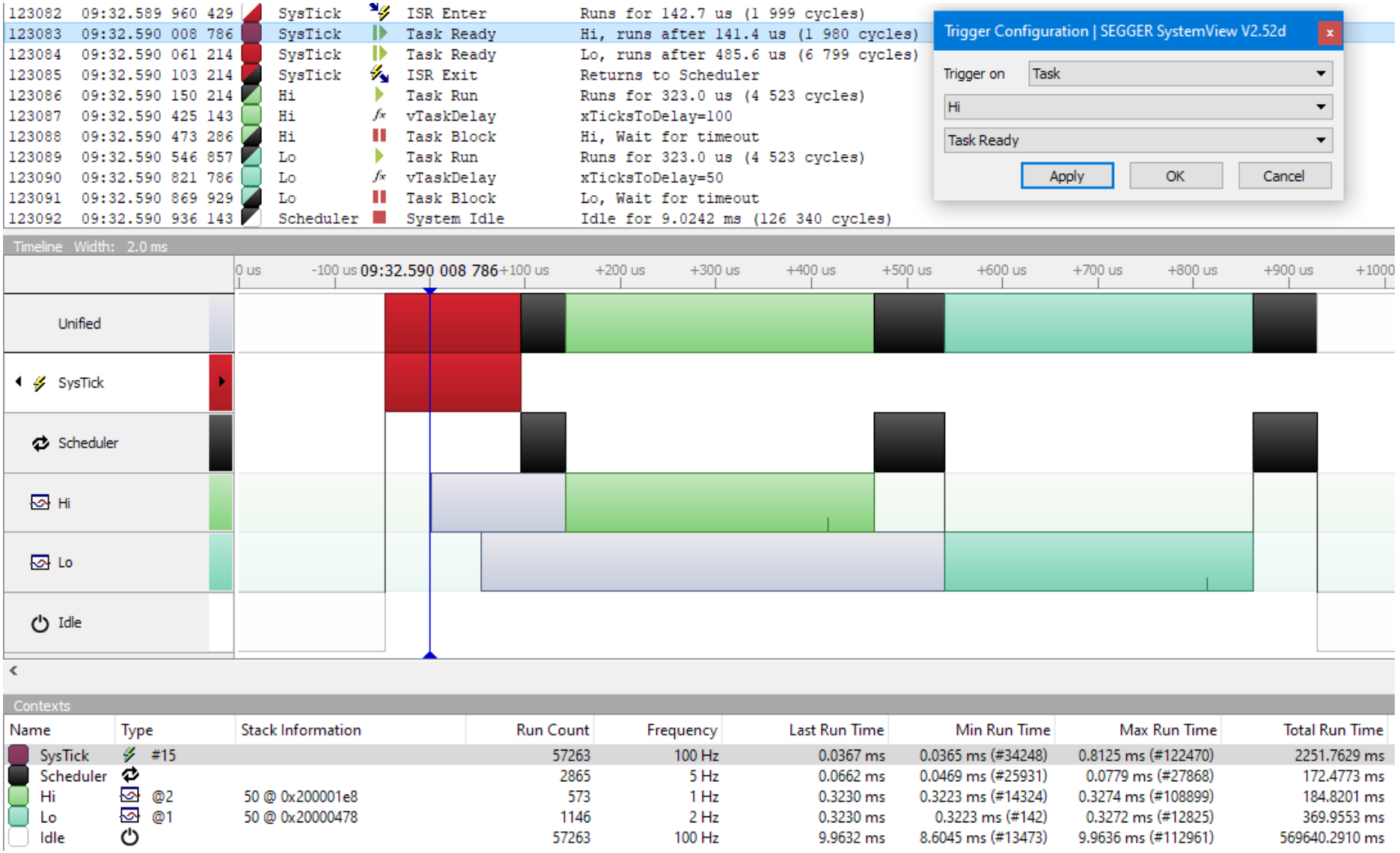
Contexts

Name	Type	Stack Information	Run Count	Frequency	Last Run Time	Min Run Time	Max Run Time	Total Run Time
SysTick	#15		30657	100 Hz	0.0367 ms	0.0365 ms (#28975)	0.8122 ms (#56368)	1203.0479 ms
Scheduler			1535	5 Hz	0.0671 ms	0.0469 ms (#16783)	0.078 ms (#30111)	92.3841 ms
Hi	@2	50 @ 0x200001e8	307	1 Hz	0.323 ms	0.3223 ms (#50748)	0.3272 ms (#33983)	98.867 ms
Lo	@1	50 @ 0x20000478	614	2 Hz	0.3230 ms	0.3223 ms (#9371)	0.3272 ms (#11418)	198.063 ms
Idle			30657	100 Hz	9.9632 ms	8.6037 ms (#54193)	9.9635 ms (#1296)	304966.9447 ms

Az alkalmazás nyomon követése (trace)

- A következő dia az egyszerű példa alkalmazásunk legbonyolultabb futási szekvenciáját mutatja. Azt, amikor mindkét saját szálunk egyszerre ébred fel.
- Látható, hogy most az ISR mindkét saját szálunkat futásra kész állapotba helyezi, majd az ISR-ből történő kilépés után az ütemező fut, ami helyesen a magas prioritású szálat fogja elsőként ütemezni.
- Ezt követi később az alacsony prioritású szál, miután a magas elment várakozni, majd az Idle szál, miután az alacsony is elment várakozó állapotba.

Az alkalmazás nyomon követése (trace)



Az alkalmazás nyomon követése (trace)

- Végezetül az időzítési viszonyok szemléltetésére nézzük meg, hogy viszonyul az előbbi eset futási ideje a SysTick ISR-ek gyakoriságához
- A taszkjaink futási idejét leginkább az UART kommunikáció szabja meg. A beállított sebesség 115200 b/s (tekintsük az egyszerűség kedvéért 100000-nek). A 8 bites karakterek 2 segéd bittel egészülnek ki az átvitel során (Start ill. Stop), így egy karakter átviteléhez 10 bit kell, tehát a sebesség 10000 char/s. Ezerrel egyszerűsítve ez: 10 char/ms

Az alkalmazás nyomon követése (trace)

- Mindkét taszk egy két betűből álló szót küld el („Hi” ill. „Lo”) plusz az „új sor” karaktert (\n), amihez az API automatikusan beszúrja a másik sorvég karaktert („kocsi vissza”, \r).
- Így tehát, ha mindkét taszkunk egyszerre ébred fel, összesen 8 karaktert fognak elküldeni. Ez a 10 char/ms sebességgel számolva kicsit kevesebb mint 1 ms időt vesz igénybe.
- Az OS hívások (vTaskDelay()), a SysTick ISR és az ütemező futási ideje ehhez képest rövidebb, de nem nulla.

Az alkalmazás nyomon követése (trace)