µC/OS-II

Micro-Controller Operating System II

Core: v2.52

Port: Atmel AVR-GCC v270603 (Julius Luukko)

Naszály Gábor
BME – MIT

ver.: 2010a.opre3

1. A µC/OS története

- **Fejlesztője:** Jean J. Labrosse
- **Motiváció:** egyik alkalmazáshoz RT kernel kellett
  - „A” kernel: már ismerték, megbízható, de túl drága volt
  - „B” kernel: ismeretlen, de olcsóbb volt
    - végül ezt vették meg
    - 2 hónapjuk ment rá, hogy pár egyszerű taszkot el tudjanak indítani
    - mint utóbb kiderült, ők voltak az egyik első vevő, és a kernel még nem volt normálisan tesztelve

1. A µC/OS története

- ezek után megvették az “A” kernelt is
  - ezt 2 nap alatt működésre bírták
  - azonban 3 hónap után találtak egy bug-ot
  - viszont a 90 napos garancia lejárt
  - ennek ellenére próbálták rávenni a gyártót, hogy javítsa ki ingyen, hisz voltaképp a hiba bejelentésével szívességet tesznek nekik
  - de a gyártó hajthatatlan volt, így fizettek a hibajavításért, ami 6 hónapig tartott!
- természetesen iszonyatosan idegesek lettek, ráadásul a terméket is nagy késéssel szállították le

1. A µC/OS története

- Jean J. Labrosse

”Well, it can’t be that difficult to write a kernel. All it needs to do is save and restore processor registers.”

- esténként és hétvégenként dolgozva elkészült egy új kernel
- kb. egy év alatt ért el az „A” kernel szintjére
- új céget azonban nem akart alapítani, mert már volt vagy 50 kernel a piacon akkoriban

1. A µC/OS története

- helyette inkább meg szeretett volna jelentetni egy cikket a C User’s Journal-ban. De elutasították az alábbi indokokkal:
  - túl hosszú volt a cikk
  - ”Another kernel article?”
- próbálkozott az Embedded Systems Programming magazinnál is:
  - az elején hasonló indokokkal utasították itt is el
  - azonban heti háromszori gyakorisággal rágva a szerkesztő fülét, végül sikerült megjelentetnie a cikket
  - akkoriban (1992) állítólag ez volt a magazin legolvasottabb cikke

1. A µC/OS története

- nem sokra rá Dr. Bernard Williams, aki többek között a C Journal kiadója is, felhívta, hogy érdekelné a cikk:

> **Jean J. Labrosse:** “Don’t you think you are a little bit late with this? The article is being published in ESP.”
>
> **Dr. Bernard Williams:** “No, No, you don’t understand, because the article is so long, I want to make a book out of it.”

- → könyv: *µC/OS, The Real-Time Kernel*
- → konferenciák → siker → cég

2. A µC/OS tulajdonságai

- forráskódban rendelkezésre áll
- hordozható (processzor függő részek külön)
- skálázható
- multi-tasking
- preemptív ütemező
- determinisztikus futási idő
- minden taszknak különböző méretű lehet a stack-je
- rendszer szolgáltatások: mailbox, queue, semaphore, fix méretű memória partíció, idő szolgáltatások stb.
- interrupt management (255 szintű egymásbaágyazhatóság)
- robusztus és megbízható

2. A µC/OS tulajdonságai

- nagyon jól dokumentált (µC/OS-II, The Real-Time Kernel könyv 300 oldalon elemzi a kódot)
- oktatási célra a kernel ingyenesen hozzáférhető
- kiegészítő csomagok:
  - TCP-IP (Protocol Stack)
  - FS (Embedded File System)
  - GUI (Embedded Graphical User Interface)
  - USB Device (Universal Serial Bus Device Stack)
  - USB Host (Universal Serial Bus Host Stack)
  - FL (Flash Loader)
  - Modbus (Embedded Modbus Stack)
  - CAN (CAN Protocol Stack)
  - BuildingBlocks (Embedded Software Components)
  - Probe (Real-Time Monitoring)

3. A µC/OS felépítése

Application software

µC/OS-II
(Processor Independent Code)

OS_CORE.C
OS_MBOX.C
OS_MEM.C
OS_Q.C
OS_SEM.C

OS_TASK.C
OS_TIME.C

uCOS_II.C
uCOS_II.H

µC/OS-II Configuration
(Application Specific)

OS_CFG.H
INCLUDES.H

µC/OS-II Port
(Processor Specific Code)

OS_CPU.H OS_CPU.C OS_CPU_A.ASM

Software

Hardware

CPU

Timer

4. A µC/OS konfigurálása (OS_CFG.H)

A skálázás **#define** sorokkal történik a konfigurációs header fájlban.

```
/* --------------------- Miscellaneous ---------------------- */

#Define OS_ARG_CHK_EN          1    /* enable (1) or disable (0) argument checking                 */
#Define OS_CPU_HOOKS_EN        0    /* uc/os-ii hooks are found in the processor port files        */
#Define OS_LOWEST_PRIO         63   /* defines the lowest priority that can be assigned            */
#Define OS_MAX_EVENTS          20   /* max. Number of event control blocks in your application     */
#Define OS_MAX_FLAGS           5    /* max. Number of event flag groups in your Application        */
#Define OS_MAX_MEM_PART        10   /* max. Number of memory partitions                            */
#Define OS_MAX_QS              5    /* max. Number of queue control blocks in your Application     */
#Define OS_MAX_TASKS           32   /* max. Number of tasks in your application                    */
#Define OS_SCHED_LOCK_EN       1    /*  include code for osschedlock() and Osschedunlock()         */
#Define OS_TASK_IDLE_STK_SIZE  512  /* idle task stack size                                         */
#Define OS_TASK_STAT_EN        1    /* enable (1) or disable(0) the statistics task                */
#Define OS_TASK_STAT_STK_SIZE  512  /* statistics task stack size                                   */
#Define OS_TICKS_PER_SEC       200  /* set the number of ticks in one second                       */
```

4. A µC/OS konfigurálása (OS_CFG.H)

```
/* ----------------------- EVENT FLAGS ------------------------ *
#define OS_FLAG_EN              0   /* Enable (1) or Disable (0) code generation for EVENT FLAGS */
#define OS_FLAG_WAIT_CLR_EN     1   /* Include code for Wait on Clear EVENT LAGS    */
#define OS_FLAG_ACCEPT_EN       1   /*    Include code for OSFlagAccept()           */
#define OS_FLAG_DEL_EN          1   /*    Include code for OSFlagDel()              */
#define OS_FLAG_QUERY_EN        1   /*    Include code for OSFlagQuery()            */

/* ----------------------- SEMAPHORES ------------------------ */
#define OS_SEM_EN               0   /* Enable (1) or Disable (0) code generation for SEMAPHORES */
#define OS_SEM_ACCEPT_EN        1   /*    Include code for OSSemAccept()            */
#define OS_SEM_DEL_EN           1   /*    Include code for OSSemDel()               */
#define OS_SEM_QUERY_EN         1   /*    Include code for OSSemQuery()             */

/* ---------------- MUTUAL EXCLUSION SEMAPHORES --------------- */
#define OS_MUTEX_EN             0   /* Enable (1) or Disable (0) code generation for MUTEX    */
#define OS_MUTEX_ACCEPT_EN      1   /*    Include code for OSMutexAccept()          */
#define OS_MUTEX_DEL_EN         1   /*    Include code for OSMutexDel()             */
#define OS_MUTEX_QUERY_EN       1   /*    Include code for OSMutexQuery()           */
```

4. A µC/OS konfigurálása (OS_CFG.H)

```
/* -------------------- MESSAGE MAILBOXES --------------------- */
#define OS_MBOX_EN              1    /* Enable (1) or Disable (0) code generation for MAILBOXES */
#define OS_MBOX_ACCEPT_EN       1    /* Include code for OSMboxAccept()       */
#define OS_MBOX_DEL_EN          0    /* Include code for OSMboxDel()          */
#define OS_MBOX_POST_EN         1    /* Include code for OSMboxPost()         */
#define OS_MBOX_POST_OPT_EN     0    /* Include code for OSMboxPostOpt()      */
#define OS_MBOX_QUERY_EN        0    /* Include code for OSMboxQuery()        */

/* ---------------------- MESSAGE QUEUES ---------------------- */
#define OS_Q_EN                 1    /* Enable (1) or Disable (0) code generation for QUEUES */
#define OS_Q_ACCEPT_EN          1    /* Include code for OSQAccept()          */
#define OS_Q_DEL_EN             1    /* Include code for OSQDel()             */
#define OS_Q_FLUSH_EN           1    /* Include code for OSQFlush()           */
#define OS_Q_POST_EN            1    /* Include code for OSQPost()            */
#define OS_Q_POST_FRONT_EN      1    /* Include code for OSQPostFront()       */
#define OS_Q_POST_OPT_EN        1    /* Include code for OSQPostOpt()         */
#define OS_Q_QUERY_EN           1    /* Include code for OSQQuery()           */

/* -------------------- MEMORY MANAGEMENT -------------------- */
#define OS_MEM_EN               0    /* Enable (1) or Disable (0) code gen.for MEM.MANAGER */
#define OS_MEM_QUERY_EN         1    /* Include code for OSMemQuery()         */
```

4. A µC/OS konfigurálása (OS_CFG.H)

```
                          /* --------------------- TASK MANAGEMENT ---------------------- */
#define OS_TASK_CHANGE_PRIO_EN  0  /* Include code for OSTaskChangePrio()                   */
#define OS_TASK_CREATE_EN       0  /* Include code for OSTaskCreate()                       */
#define OS_TASK_CREATE_EXT_EN   1  /* Include code for OSTaskCreateExt()                    */
#define OS_TASK_DEL_EN          0  /* Include code for OSTaskDel()                          */
#define OS_TASK_SUSPEND_EN      0  /* Include code for OSTaskSuspend() and OSTaskResume()   */
#define OS_TASK_QUERY_EN        0  /* Include code for OSTaskQuery()                        */

                          /* --------------------- TIME MANAGEMENT ---------------------- */
#define OS_TIME_DLY_HMSM_EN     1  /* Include code for OSTimeDlyHMSM()                      */
#define OS_TIME_DLY_RESUME_EN   0  /* Include code for OSTimeDlyResume()                    */
#define OS_TIME_GET_SET_EN      0  /* Include code for OSTimeGet() and OSTimeSet()          */

typedef INT16U OS_FLAGS;           /* Date type for event flag bits (8, 16 or 32 bits)      */
```

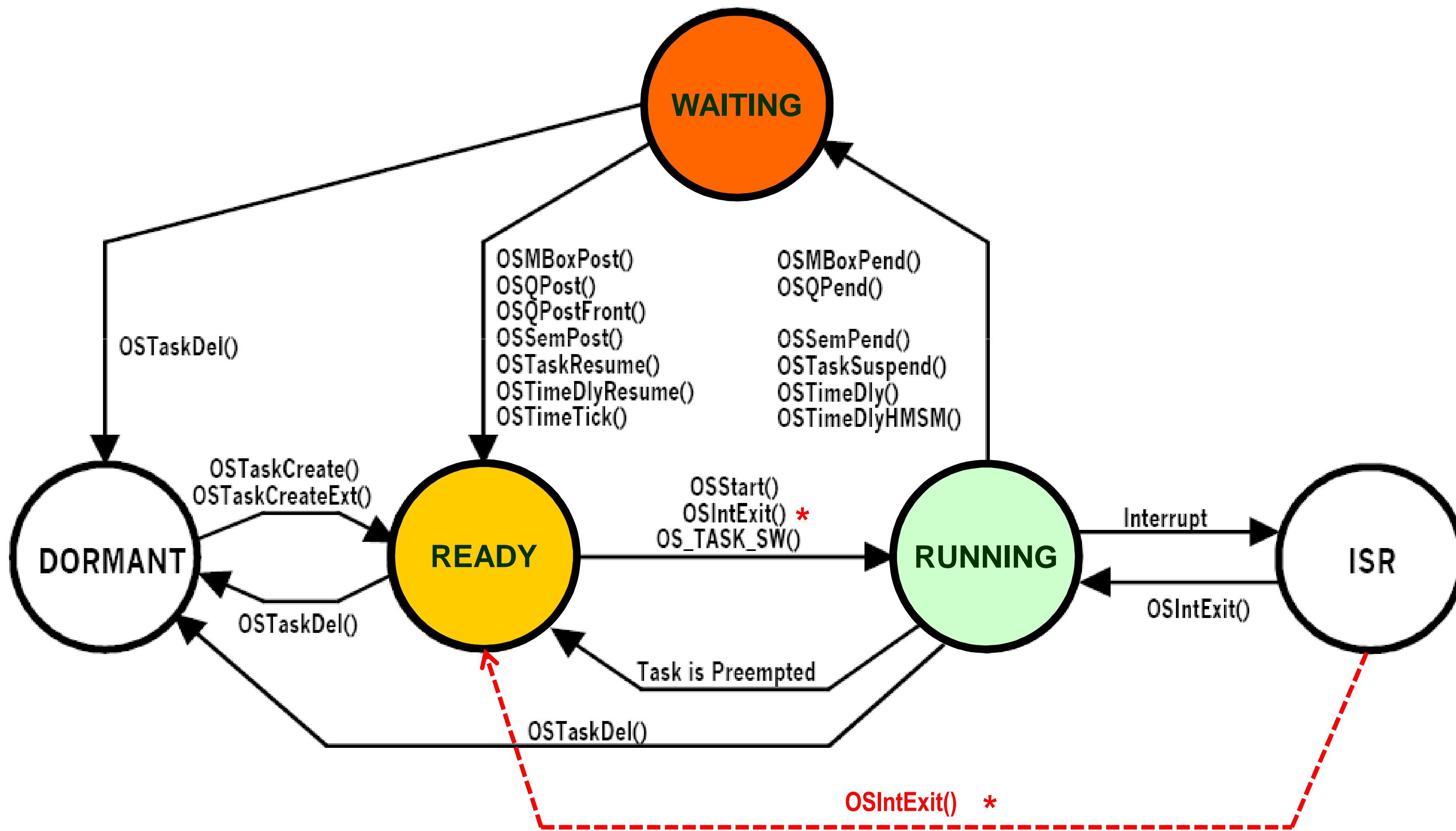
4. A µC/OS konfigurálása (OS_CFG.H)

A forráskód a **#define** sorokra épülten tartalmaz feltételes fordítási direktívákat.

```
#if OS_SEM_ACCEPT_EN > 0
INT16U  OSSemAccept (OS_EVENT *pevent) {
   INT16U     cnt;
#if OS_CRITICAL_METHOD == 3        /* Allocate storage for CPU status register  */
   OS_CPU_SR  cpu_sr = 0;
#endif

#if OS_ARG_CHK_EN > 0
   if (pevent == (OS_EVENT *)0) {      /* Validate 'pevent'                        */
      return (0);
   }
#endif
   …
}
#endif
```

5. A µC/OS taszk állapotai



5. A µC/OS taszk állapotai

- Két speciális állapot:
  - **DORMANT:** „szunnyadó”, akkor van ebben az állapotban a taszk, amikor a memóriában ugyan megtalálható, de az ütemező hatáskörében nincs benne (nem hozták még létre `OSTaskCreate()`, vagy törölték `OSTaskDel()`)
  - **ISR:** a taszkot megszakította egy interrupt rutin. Ha a rutin mellékhatásaként egy magasabb prioritású taszk válik futásra késszé, akkor a rutin végeztével az kerül a RUNNIG állapotba, és a megszakított taszk pedig a READY-be
- Ha nincs egy futásra kész taszk sem, akkor egy speciális taszk, az `OSTaskIdle()` fut.

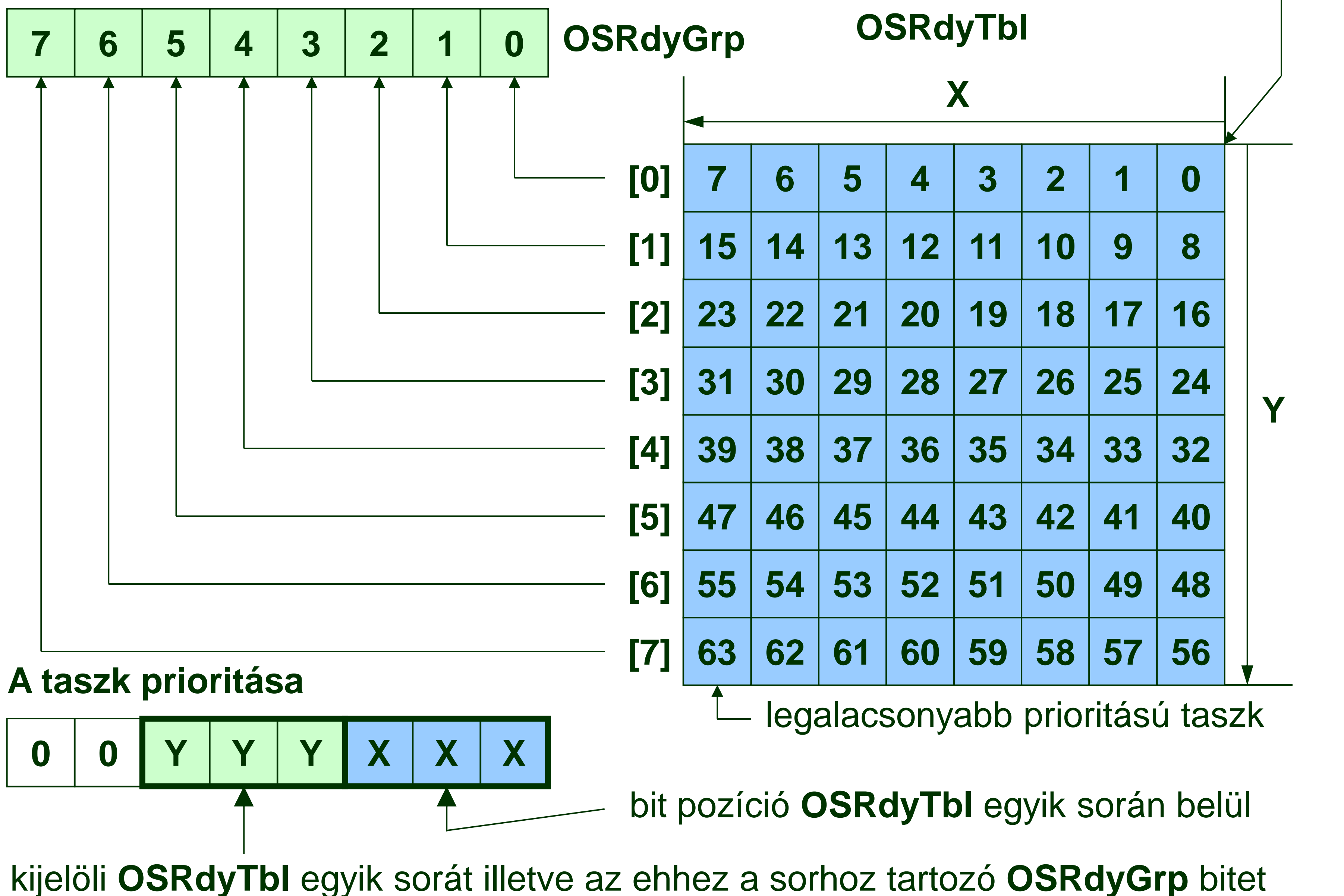
6. A µC/OS ütemezője

- A futásra kész taszkok egy **2D bitmap** struktúrában tárolódnak, ahol az egyes bitek reprezentálják a taszkokat. Ha egy bit 1, a hozzá tartozó taszk futásra kész. A bitek prioritást is jelentenek (az egyik sarokban a legmagasabb, a másikban a legalacsonyabb prioritású taszkhoz tartozó bit található).

6. A µC/OS ütemezője

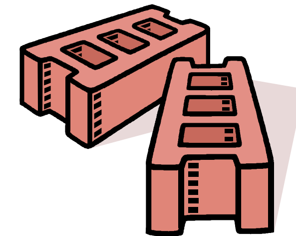
- + Gyors beszúrás, ami független a futásra kész taszkok számától. Csupán pár egyszerű művelet kell a megfelelő bit beállításához. Továbbá a legmagasabb futásra kész taszkot is gyorsan meg lehet találni lookup-table segítségével.
- – A taszkoknak egyedi prioritással kell rendelkezniük (azaz a round-robin / time slice ütemezés nem lehetséges)
- – Szükség van egy lookup táblázatra előre számított értékekkel a *bitminta* → *legmagasabb prioritású futásra kész taszk* megfeleltetéshez

6. A µC/OS ütemezője



6. A μC/OS ütemezője

- Építőkockák ütemezéshez:



  1. taszk kivétele a futásra készek közül
  2. taszk futásra késszé tétele
  3. a legmagasabb prioritású, futásra kész taszk megtalálása
  4. kontextus váltás

6. A µC/OS ütemezője

- Kik használják az imént ismertetett építőkockákat?
  1. **Taszk kivétele a futásra készek közül:** várakozó rendszerhívások (pl. `OSSemPend()`, `OSMBoxPend(), OSTimeDly()`). Először elvégzik a rájuk jellemző műveletet (pl. szemafor lefoglalása), majd kiveszik a futásra kész taszkok halmazából az őket meghívó taszkot. A végén pedig meghívják az ütemezőt.

6. A µC/OS ütemezője

2. **Taszk futásra késszé tétele:** elengedő rendszerhívások (pl. `OSSemPost()`, `OSMBoxPost()`) vagy adott idő letelte. Először elvégzik a rájuk jellemző műveletet (pl. szemafor felszabadítása), majd futásra késszé teszik a megfelelő taszkot, taszkokat. A végén pedig meghívják az ütemezőt.

6. A µC/OS ütemezője

3. **A legmagasabb prioritású, futásra kész taszk megtalálása:** az ütemező feladata. Ha a legmagasabb prioritású, futásra kész taszk prioritása nem egyezik meg az éppen futóéval, elvégzi a taszk váltást.
4. **Kontextus váltás:** szintén az ütemező feladata.

6. A µC/OS ütemezője

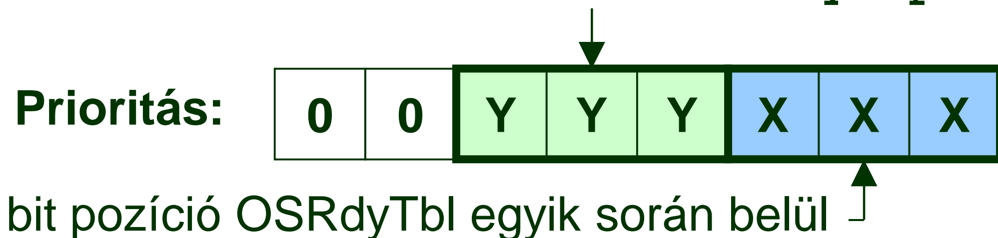
2. Taszk futásra késszé tétele

A taszk prioritása által kijelölt biteket 1-be kell billenteni az `OSRdyGrp`-ben és az `OSRdyTbl`-ben.

```
OSRdyGrp             |= OSMapTbl[prio >> 3];
OSRdyTbl[prio >> 3]  |= OSMapTbl[prio & 0x07];
```

`OSMapTbl:`

Index	Bit mask (binary)
0	00000001
1	00000010
2	00000100
3	00001000
4	00010000
5	00100000
6	01000000
7	10000000



prio:	a taszk prioritása
prio >> 3:	YYY
prio & 0x07:	XXX

6. A µC/OS ütemezője

1. Taszk kivétele a futásra készek közül

A taszk prioritása által kijelölt bitet törölni kell a megfelelő **`OSRdyTbl`** sorból. Ha ezzel nem marad egy futásra kész taszk sem az adott sorban, a sornak megfelelő bitet az **`OSRdyGrp`**-ból is törölni kell.

```
if ((OSRdyTbl[prio >> 3] &= ~OSMapTbl[prio & 0x07]) == 0)
OSRdyGrp &= ~OSMapTbl[prio >> 3];
```

6. A µC/OS ütemezője

3. A legmagasabb prioritású futásra kész taszk megtalálása

Erre a célra egy segédtáblázatot használnak, aminek az a feladata, hogy egy adott 8 bites számról megmondja, melyik az a legalacsonyabb bitpozíció (azaz legmagasabb prioritás), ahol 1-es található.

Például:

Hexa	Bináris								Legalacsonyabb 1-es
	7.	6.	5.	4.	3.	2.	1.	0.	
0x2C	0	0	1	0	1	1	0	0	2.

6. A µC/OS ütemezője

```
INT8U  const  OSUnMapTbl[256] = {
    0,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,  /* 0x00 to 0x0F */
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,  /* 0x10 to 0x1F */
    5,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,  /* 0x20 to 0x2F */
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,  /* 0x30 to 0x3F */
    6,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,  /* 0x40 to 0x4F */
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,  /* 0x50 to 0x5F */
    5,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,  /* 0x60 to 0x6F */
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,  /* 0x70 to 0x7F */
    7,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,  /* 0x80 to 0x8F */
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,  /* 0x90 to 0x9F */
    5,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,  /* 0xA0 to 0xAF */
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,  /* 0xB0 to 0xBF */
    6,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,  /* 0xC0 to 0xCF */
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,  /* 0xD0 to 0xDF */
    5,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0,  /* 0xE0 to 0xEF */
    4,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0   /* 0xF0 to 0xFF */
};
```

6. A µC/OS ütemezője

Először meg kell határozni azt a sort (csoportot, YYY értéket), ami a legmagasabb prioritású 1-es tartalmazza, majd a soron belül is meg kell keresni a legmagasabb prioritást azonosító 1-est (XXX). Ezt követően a legmagasabb prioritás már számolható: YYY * 8 + XXX.

```
y = OSUnMapTbl[OSRdyGrp];
x = OSUnMapTbl[OSRdyTbl[y]];
prio = (y << 3) + x;
```

6. A µC/OS ütemezője

4. Kontextus váltás

1. az aktuális környezet mentése:
   - *regiszterek mentése*
   - veremmutató mentése

2. az új környezet visszaállítása:
   - veremmutató visszaállítása
   - *regiszterek visszaállítása*

A kontextus váltás platform függő!

6. A µC/OS ütemezője

- Az ütemező egy függvény, amelyet más OS függvények hívhatnak meg. Pl.: ha le akarunk foglalni egy szemafort, de az még nem szabad, akkor az **`OSSemPend()`** hívás kiveszi a futásra készek közül az aktuális taszkot, és meghívja az ütemezőt, ami eldönti, hogy a megmaradt futásra kész taszkok közül melyiknek van a legmagasabb a prioritása, és végrehajtja a kontextus váltást.

6. A µC/OS ütemezője

```
void OSSched (void)
{
    INT8U y;

    OS_ENTER_CRITICAL();
    if ((OSLockNesting | OSIntNesting) == 0) {
        y = OSUnMapTbl[OSRdyGrp];
        OSPrioHighRdy = (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl[y]]);
        if (OSPrioHighRdy != OSPrioCur) {
            OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
            OSCtxSwCtr++;
            OS_TASK_SW();
        }
    }
    OS_EXIT_CRITICAL();
}
```

Az ütemező algoritmus kritikus szakasznak minősül: azaz előtte tiltjuk, utána engedélyezzük a megszakításokat (ez CPU függő kód). Ezek után csak akkor fut le az ütemező, ha nincs tiltva, és nem ISR-ből hívták meg. Maga az ütemező pedig először kikeresi a legmagasabb prioritású futásra kész taszkhoz tartozó prioritást. Kontextusváltás csak akkor történik, ha ez nem egyezik meg az éppen futó taszk prioritásával.

6. A µC/OS ütemezője

```
if (OSPrioHighRdy != OSPrioCur) {
    OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
    OSCtxSwCtr++;
    OS_TASK_SW();
}
```

A kontextusváltás azzal kezdődik, hogy kikeressük a legmagasabb prioritású futásra kész taszk TCB-jére (Task Control Block) mutató pointert. (A TCB tárolja a taszkra jellemző adatokat mint prioritás, státusz, verem mutató stb. Más környezetben PCB-nek nevezik (Process Control Block).) Ezután egy, a kontextusváltásokat statisztikai okokból számoló számláló értékét növeljük eggyel. A tényleges környezetváltást (regiszterek, stack pointer mentése, visszaállítása) pedig processzor függő assembly kód végzi.

6. A µC/OS ütemezője

Fontosabb mezők a Task Control Block struktúrában:

```
typedef struct os_tcb {
    OS_STK   *OSTCBStkPtr; // Pointer to top of stack
    ...
    INT16U   OSTCBDly; // Ticks to delay task or timeout wait
    INT8U    OSTCBStat; // Task status
    INT8U    OSTCBPrio; // Task priority
     ...
} OS_TCB;
```

6. A µC/OS ütemezője

OS_TASK_SW()

1. az aktuális környezet mentése:
   - *regiszterek mentése*
   - veremmutató mentése
2. az új környezet visszaállítása:
   - veremmutató visszaállítása
   - *regiszterek visszaállítása*

A kontextus váltás **platform függő!**
A bemutatott kódrészlet **AVR ATmega128** mikrovezérlőre vonatkozik.

```
PUSHRS
PUSHSREG

LDS     R30,OSTCBCur
LDS     R31,OSTCBCur+1
in      r28,_SFR_IO_ADDR(SPL)
ST      Z+,R28
in      r29,_SFR_IO_ADDR(SPH)
ST      Z+,R29

CALL    OSTaskSwHook
LDS     R16,OSPrioHighRdy
STS     OSPrioCur,R16

LDS     R30,OSTCBHighRdy
LDS     R31,OSTCBHighRdy+1
STS     OSTCBCur,R30
STS     OSTCBCur+1,R31
LD      R28,Z+
out     _SFR_IO_ADDR(SPL),R28
LD      R29,Z+
out     _SFR_IO_ADDR(SPH),R29

POPSREG
POPRS
```

7. Időzítő (timer)

- különféle okokból szükség lehet az idő múlásának követésére, például:
  - time-slicing ütemezéshez (µC/OS-II alatt nincs)
  - taszk futásának adott ideig történő felfüggesztése
  - timeoutos OS hívások
  - függvény adott idő múlva történő meghívásához
  - függvény adott időközönként történő periodikus meghívásához
  - az idő mérésére

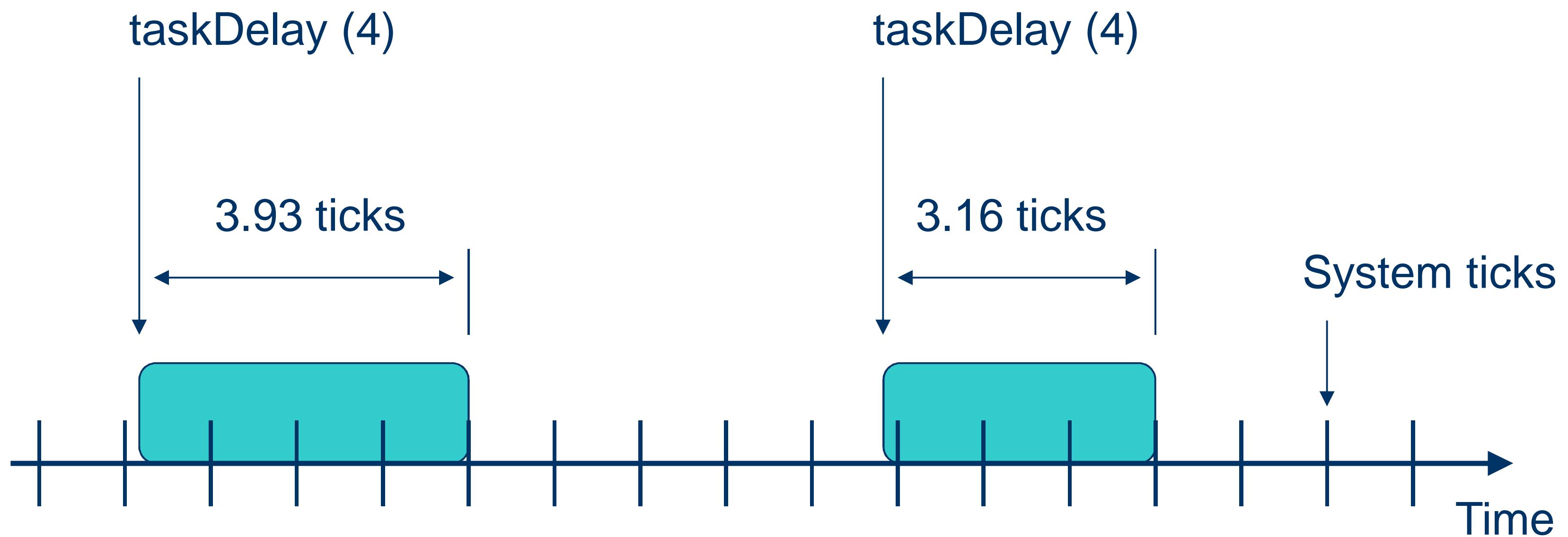
7. Időzítő (timer)

- ezen a célokra az OS „felhúz” egy hardver timert (**heartbeat timer**), ami aztán periodikus megszakításokat generál (**system ticks**)
- Mekkora legyen egy óraütés?
  - minél kisebb, annál pontosabbak lesznek az időzítőn alapuló szolgáltatások
  - minél nagyobb, annál kevesebb overheadet jelent a timer megszakítás

7. Időzítő (timer)

- Erre a megszakításra az OS „felfűz” némi kódot saját magából. Ez biztosítja például azt, hogy ha egy taszk adott időre elküldte magát várakozó állapotba, akkor az idő letelte után futásra kész lehessen.
- Továbbá ha felébredése esetén ő a legnagyobb prioritású, akkor a timer ISR után az OS erre a taszkra adja a futás jogát. Megvalósítva a preempciót.

7. Időzítő (timer)



- Pontosság: 1 óraütés
- Minimum **n** óraütés késleltetéshez **n+1**-et kell megadni

8. Egyéb OS szolgáltatások

- Taszk kezelés
  - Létrehozás
  - Megszüntetés
- Idő kezelés
  - Taszk futásának adott idővel történő késleltetése
  - Rendszer idő lekérdezés
- Memória partíciók kezelése
  - Lefoglalás
  - Felszabadítás

8. Egyéb OS szolgáltatások

- Szinkronizáció / kommunikáció
  - Szemafor (számláló)
  - Mailbox (~ szemafor + void*)
  - Queue (~ n*mailbox)
  - Mutex (szemafor prioritás inverzió elleni védelemmel: prioritás öröklés protokoll)
  - Event flags (8, 16, 32 bites változó; bitek tetszőleges ÉS, VAGY kombinációjára lehet várni)

9. A µC/OS alkalmazás tipikus szerkezete

```
void YourTask (void *pdata){
 for (;;) {
  /* USER CODE */
  !! Call one of uC/OS-II’s
  !! services: OSMboxPend(),
  !! OSQPend(),OSSemPend(),
  !! OSTaskDel(OS_PRIO_SELF),
  !! OSTaskSuspend(OS_PRIO_SELF),
  !! OSTimeDly(),OSTimeDlyHMSM()
  /* USER CODE */
 }
}
```

← **Végtelen ciklus** taszk szervezés

vagy

„**Single shot**” taszk szervezés

```
void YourTask (void *pdata)
{
 /* USER CODE */
 OSTaskDel(OS_PRIO_SELF);
}
```

```
void main (void){
 OSInit(); /* Initialize uC/OS-II */
 ...
 !! Create at least 1 task using either OSTaskCreate() or
 !! OSTaskCreateExt(). And maybe other OS objects (MBox, ...).
 ...
 OSStart(); /* Start multitasking! OSStart() will not return */
}
```

10. Egyszerű példaprogram

- A következőkben egy egyszerű példaprogram segítségével szemléltetjük a μC/OS működését.
- A program tartalmaz egy inicializáló taszkot, melynek feladata az időzítő megszakítás beállítása (ami az OS időkezelő szolgáltatásaihoz kell), valamint létrehoz másik két taszkot, és a végén törli magát (ezzel példát mutatva a hagyományos végtelen ciklusú taszk szervezéstől eltérő, egyszeri lefutású taszk struktúrára).
- A másik két taszk mindegyike 4ms hosszú hasznos kód futását szimulálja (szoftveres várakozó hurkokkal). Ezt követően mindkettő elküldi magát OS értelemben vett (időre) várakozó állapotba. A magasabb prioritású 4, az alacsonyabb 2 óra ütésig (1 óra ütés = 10 ms). (Mindkét taszk végtelen ciklus szervezésű.)

10. Egyszerű példaprogram (kód 1/4)

```
#include "includes.h"

// Prioritások (kisebb szám -> nagyobb prioritás):
#define TASK_INIT_PRIO  24
#define TASK_HIGH_PRIO  37
#define TASK_LOW_PRIO   50

// Vermek lefoglalása:
#define STACK_SIZE 128
OS_STK  TaskInitStk[STACK_SIZE];
OS_STK  TaskHighStk[STACK_SIZE];
OS_STK  TaskLowStk[STACK_SIZE];

// Függvény deklarációk:
void  TaskInit(void *data);
void  TaskHigh(void *data);
void  TaskLow(void *data);
...
```

10. Egyszerű példaprogram (kód 2/4)

```
...
void main (void)
{
    OSInit(); // Az OS inicializációja

    // TaskInit létrehozása:
    OSTaskCreate(
        // A taszk kódját tartalmazó függvény:
            TaskInit,
        // Opcionális paraméter:
            NULL,
        // A taszk vermének teteje:
            &TaskInitStk[STACK_SIZE - 1],
        // A taszkhoz rendelt prioritás:
            TASK_INIT_PRIO);

    OSStart(); // Multitaszking indítása
}
...
```

10. Egyszerű példaprogram (kód 3/4)

```
...
void TaskInit (void *data) {

    // Timer interrupt inicializálás (1 OS tick = 10ms):
    INIT_TIMER();

    // TaskHigh és TaskLow létrehozása:
    OSTaskCreate(
      TaskHigh, NULL, &TaskHighStk[STACK_SIZE - 1],
      TASK_HIGH_PRIO);

    OSTaskCreate(
      TaskLow, NULL, &TaskLowStk[STACK_SIZE - 1],
      TASK_LOW_PRIO);

    // Saját magunk törlése -> példa egyszer lefutó taszkra:
    OSTaskDel(OS_PRIO_SELF);
}
...
```

10. Egyszerű példaprogram (kód 4/4)

```
...
void TaskHigh(void *data) {
   while(1) {
      // 4ms hasznos kód szimulálása:
         _delay_ms(4);
      // Várakozó állapotba küldés, ébredés 4 OS tick múlva:
         OSTimeDly(4);
   }
}

void TaskLow(void *data) {
   while(1) {
      // 4ms hasznos kód szimulálása:
         _delay_ms(4);
      // Várakozó állapotba küldés, ébredés 2 OS tick múlva:
         OSTimeDly(2);
   }
}
```

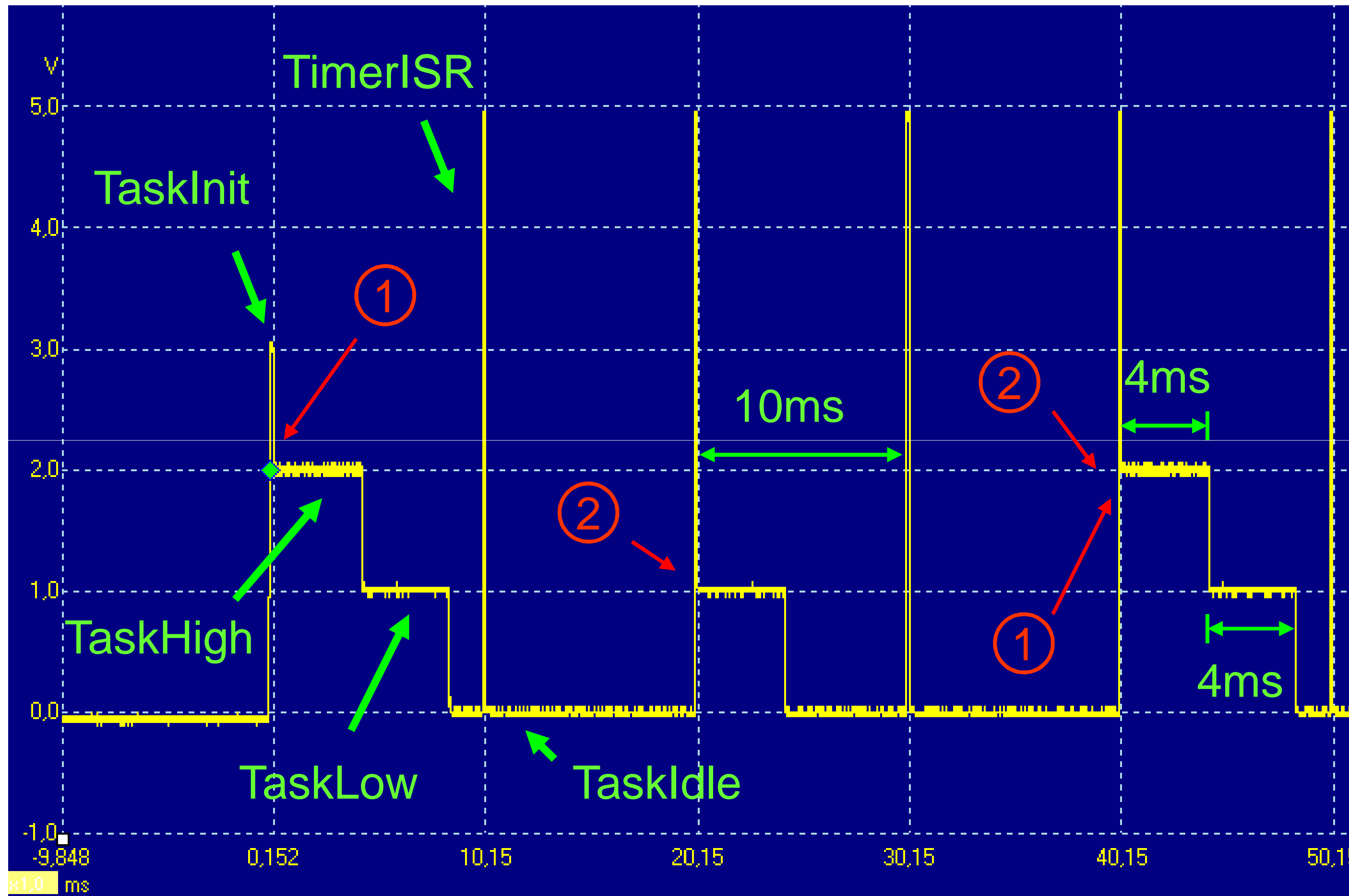
10. Egyszerű példaprogram (futás)

- Amennyiben az adott fejlesztőpanel rendelkezik analóg kimenettel, a program futásának szemléltetésére egy bevált gyakorlat, hogy az éppen futó taszk prioritásával arányos feszültség szintet adunk ki. (Továbbá érdemes lehet még az időzítő megszakításhoz is rendelni egy szintet.)

10. Egyszerű példaprogram (futás)

- Bizonyos operációs rendszerekhez (pl. FreeRTOS) létezik ilyen kiegészítő modul. A µC/OS itt alkalmazott verzióját némi kiegészítés, módosítás segítségével lehetett alkalmassá tenni a feladatra.

10. Egyszerű példaprogram (futás)



A futás szemléltetése a taszkokhoz és az időzítő megszakításhoz rendelt feszültség szintek kiadásával.

10. Egyszerű példaprogram (futás)

- A program futása az előző ábrán jól követhető. A teljességre történő magyarázat helyett két fontosabb észrevétel a számokkal jelölt időpontokra:
  - (1): ezekben a pontokban mind a magas, mind az alacsony prioritású taszk futásra kész. Látható, hogy az OS ütemezője ilyen esetben először a magas prioritású taszkot ütemezi.
  - (2): ezen esetekben pedig azt érdemes észrevenni, hogy egy megszakítás után NEM feltétlen a megszakított taszk folytatja a futását!

10. Újdonságok v2.52 óta

- **Várakozás egyszerre több eseményre**: szemafor, mailbox, queue – mutex, event flag még nem támogatott (v2.86)
- **Időzítők**: függvény adott idő múlva vagy periodikus hívására (v2.81)
- **255** taszk támogatása (v2.80)
- **µC/OS-III**
  - **Tetszőleges számú** taszk és egyéb OS objektum
  - Egy prioritási szinten lehet több taszk is (→ **round-robin** vagy **time-slicing** ütemezés)
  - Nulla közeli megszakítás tiltás idő