Nagyteljesítményű mikrovezérlők
10. RTOS alapok

Scherer Balázs

Méréstechnika és
Információs Rendszerek
Tanszék

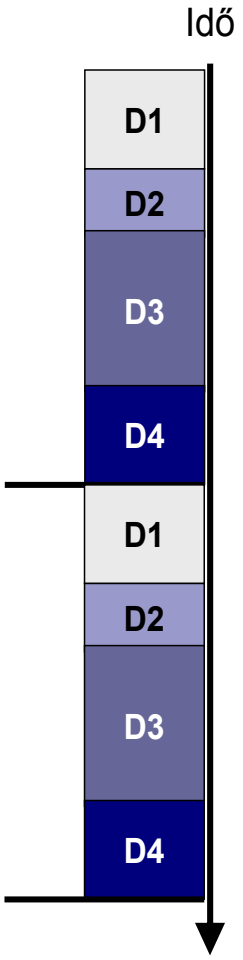
Budapest University of Technology and Economics
Department of Measurement and Information Systems

MŰEGYETEM 1782

Alap beágyazott szoftver architektúrák I.

- Round-Robin

```
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
        }
        ...
  }
}
```



Alap beágyazott szoftver architektúrák II.

- Round-Robin
  - Nagyon egyszerű
  - Nincs interrupt, a főciklus végzi az „ütemezést”
  - Nincs közös erőforrás probléma

  - Worst case válaszidő = a job-ok össz válaszideje
  - A Worst Case válaszidő lineárisan nő a job-ok számával
  - A válaszidőnek rendkívül nagy a jitter-e
  - Ha gyors válasz kell, akkor annak a kiszolgálási pontjait meg lehet többszörözni, de ez rontja az egész rendszer válaszidejét
  - Egy új job felborítja az eddigi időzítést

Alap beágyazott szoftver architektúrák III.

- Megszakításokkal kiegészített Round-Robin

```
BOOL Device1_flag = 0;
BOOL Device2_flag = 0;
BOOL Device3_flag = 0;

void interrupt vDevice1(void)
{
   // Handle Device 1 time critical part
   Device1_flag = 1;
}
void interrupt vDevice2(void)
{
   // Handle Device 2 time critical part
   Device1_flag = 2;
}
void interrupt vDevice3(void)
{
   // Handle Device 3 time critical part
   Device3_flag = 1;
}
```

```
void main(void)
{
  while(1)
  {
     if ( Device1_flag )
       {
          // Handle Device 1 and its data
       }
      if (Device2_flag )
       {
          // Handle Device 2 and its data
       }
      if (Device3_flag )
       {
          // Handle Device 3 and its data
       }
      …
  }
}
```

Alap beágyazott szoftver architektúrák IV.

- Megszakításokkal kiegészített Round-Robin
  - Picit jobban kezeli az időkritikus részeket
  - Jelentkezhet az osztott változó probléma az IT és a főprogram között
  - Esetleg a flag-ek helyett használható számláló is
  - Worst case válaszidő = a job-ok össz válaszideje + IT
  - A Worst Case válaszidő lineárisan nő a job-ok számával
  - A válaszidőnek rendkívül nagy a jitter-e
  - Ha gyors válasz kell, akkor annak a kiszolgálási pontjait meg lehet többszörözni, de ez rontja az egész rendszer válaszidejét
  - Egy új job felborítja az eddigi időzítést

Lehetséges problémák I.: osztott változók

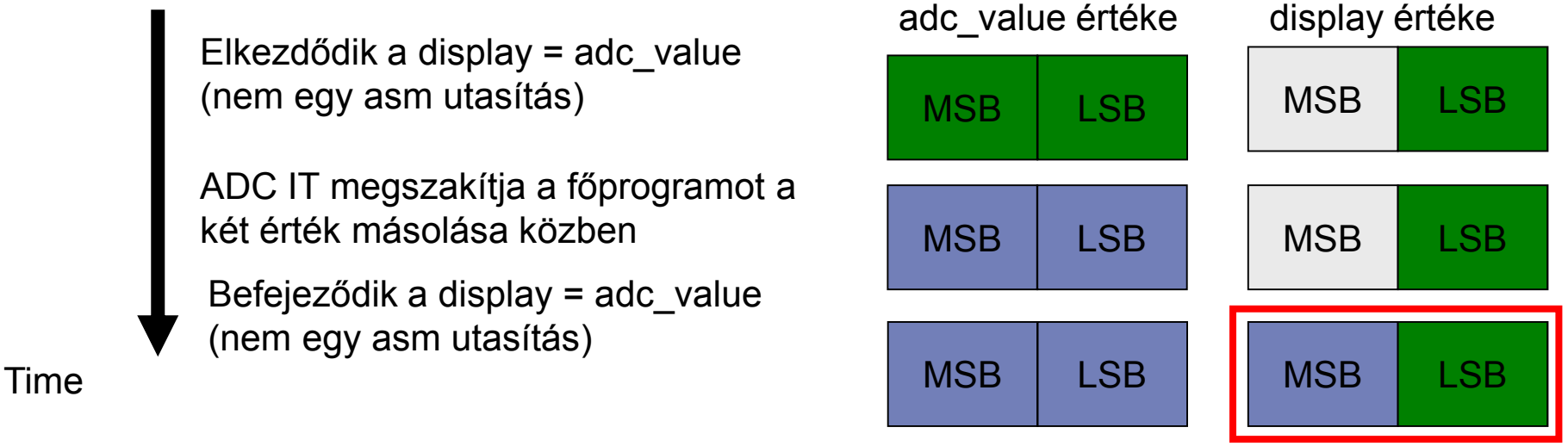
- Nem atomikus módon kezelt változók főprogramban és interruptban történő használata problémához vezethet.

főprogram

```
unsigned short adc_value,display;
main()
{
  while(1) { display = adc_value }
}
```

Interrupt

```
external unsigned short adc_value;
INTERRUPT(SIG_ADC )
{
  // Az AD kiolvasása
   adc_value = read_adc();
}
```



Problémák II.: függvény újrahívatóság

- Az előző eset kiterjesztése és egyik leggyakoribb megjelenési formája.
- Olyan függvények nem használhatóak interrupt-ból és főprogramból is egyszerre, amelyek globális változókat, static kulcsszóval ellátott változókat vagy közös erőforrást használnak
- A fordító általában figyelmeztet erre

Alap beágyazott szoftver architektúrák V.

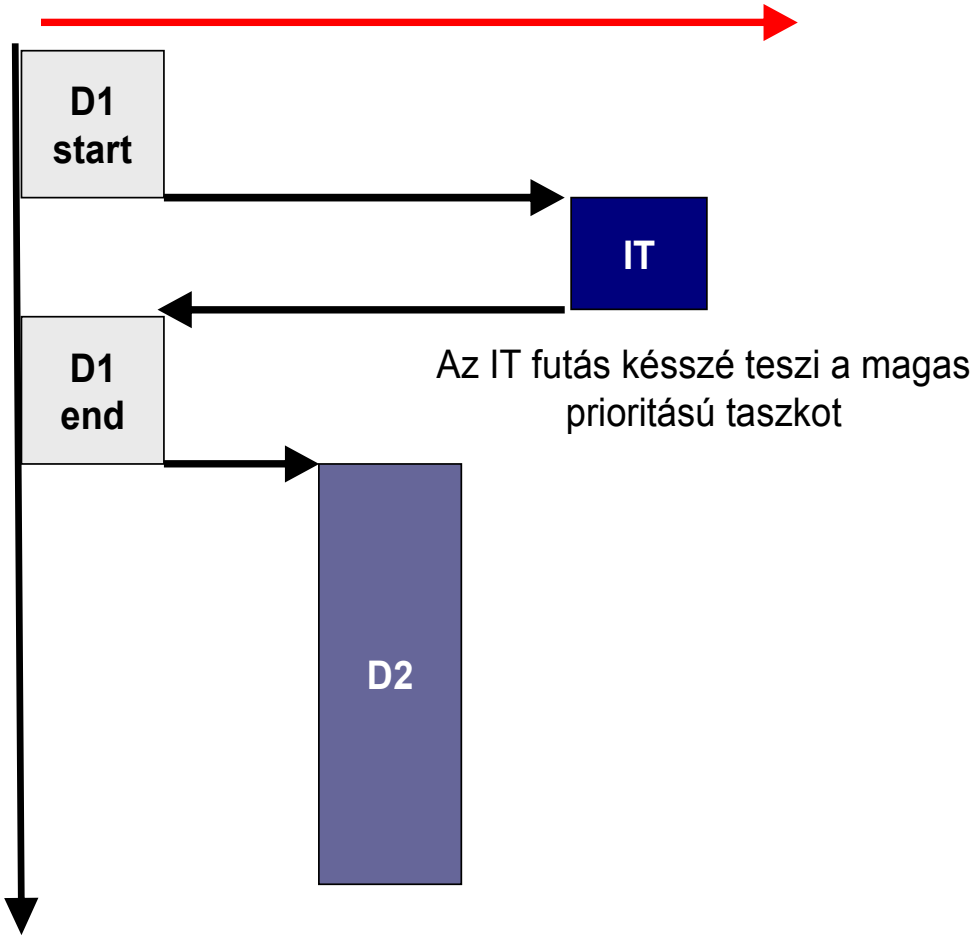
- Függvénysor alapú nem preemptív ütemezés

```
void interrupt vDevice1(void)
{
   // Handle Device 1 time critical part
   // Put Device1_func to call queue
}
void interrupt vDevice2(void)
{
   // Handle Device 2 time critical part
   // Put Device2_func to call queue
}
void interrupt vDevice3(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  }
```

Alap beágyazott szoftver architektúrák VI.

- Függvénysor alapú nem preemptív ütemezés



Alap beágyazott szoftver architektúrák VII.

- Függvénysor alapú nem preemptív ütemezés
  - Képes a prioritások kezelésére.
  - Jelentkezhet az osztott változó probléma az IT és a főprogram között.
  - Worst case válaszidő = a leghosszabb job válaszideje + IT
  - A Worst Case válaszidő nem nő lineárisan a job-ok számával.
  - A válaszidő jitter jóval kézbenntarthatóbb
  - Egy új job nem borítja fel az eddigi időzítést

Alap beágyazott szoftver architektúrák VIII.

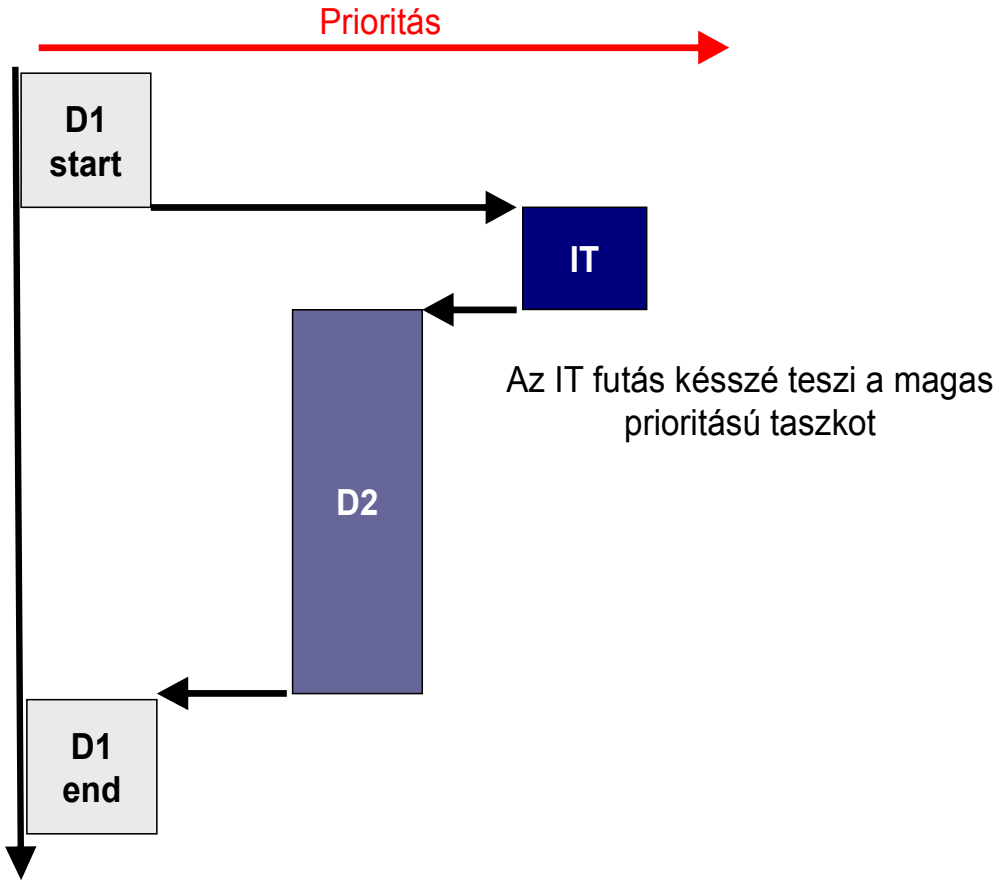
- Real Time OS, preemptív ütemezés

```
void interrupt vDevice1(void)
{
   // Handle Device 1 time critical part
   // Set signal to Device1_task
}
void interrupt vDevice2(void)
{
   // Handle Device 2 time critical part
   // Set signal to Device2_task
}
void interrupt vDevice3(void)
{
   // Handle Device 3 time critical part
   // Set signal to Device3_task
}
```

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

void Device2_task (void)
{
   // Wait for signal to Device2_task
   // Handle Device 2
}

void Device3_task (void)
{
   // Wait for signal to Device3_task
   // Handle Device 3
}
```

Alap beágyazott szoftver architektúrák IX.

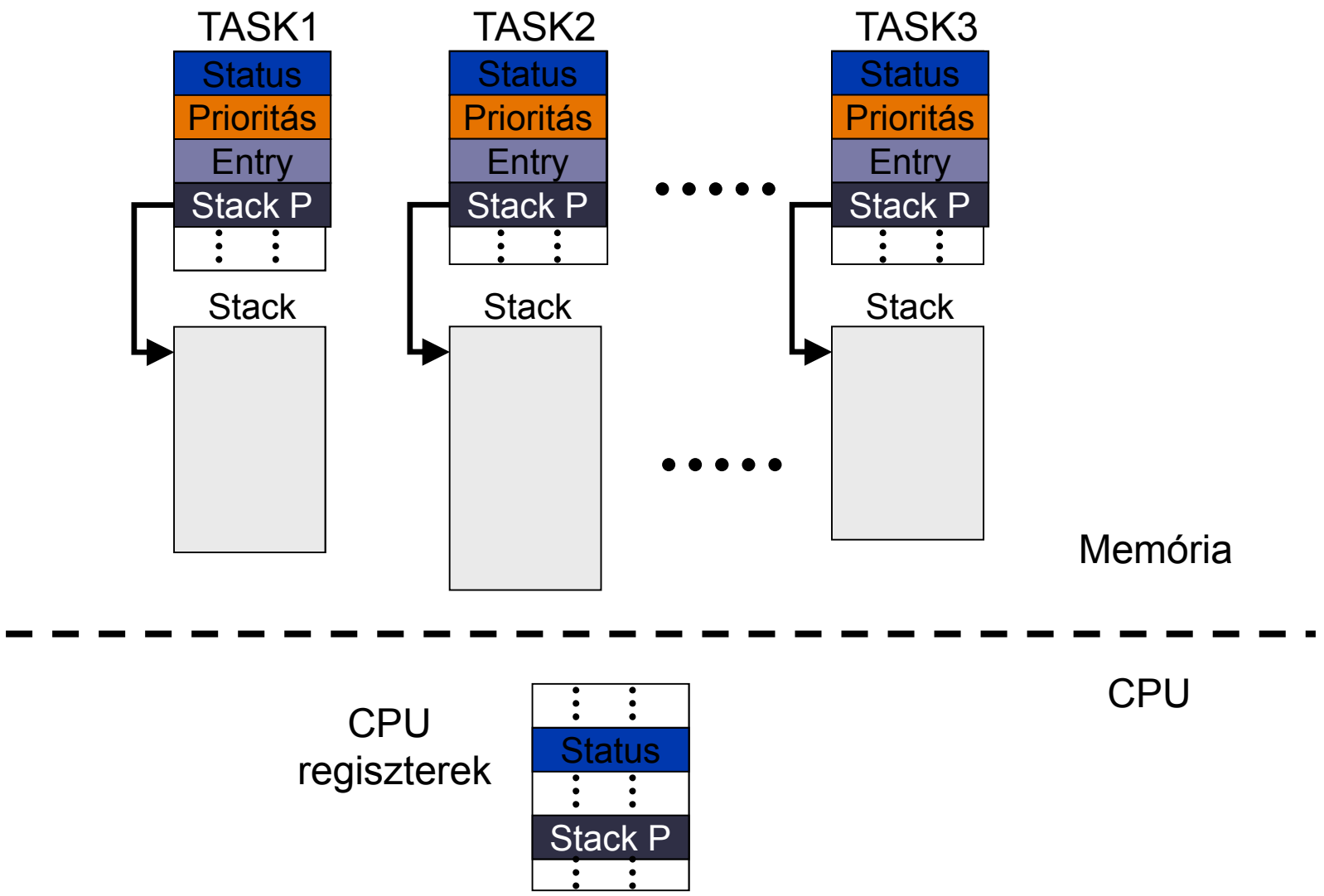
- Real Time OS, preemptív ütemezés



Alap beágyazott szoftver architektúrák X.

- Real Time OS, preemptív ütemezés
  - Erősen prioritásos
  - Jelentkezhet az osztott változó probléma az IT és a főprogram között, valamint az egyes task-ok között is.
  - Worst case válaszidő = a task váltási idő + IT
  - A Worst Case válaszidő nem nő az új job –ok hozzáadásával
  - A válaszidő jitter nagyon alacsony a magas prioritású szálakra
  - Jelentős kód overhead

A Task-ok felépítése és a taszkváltás



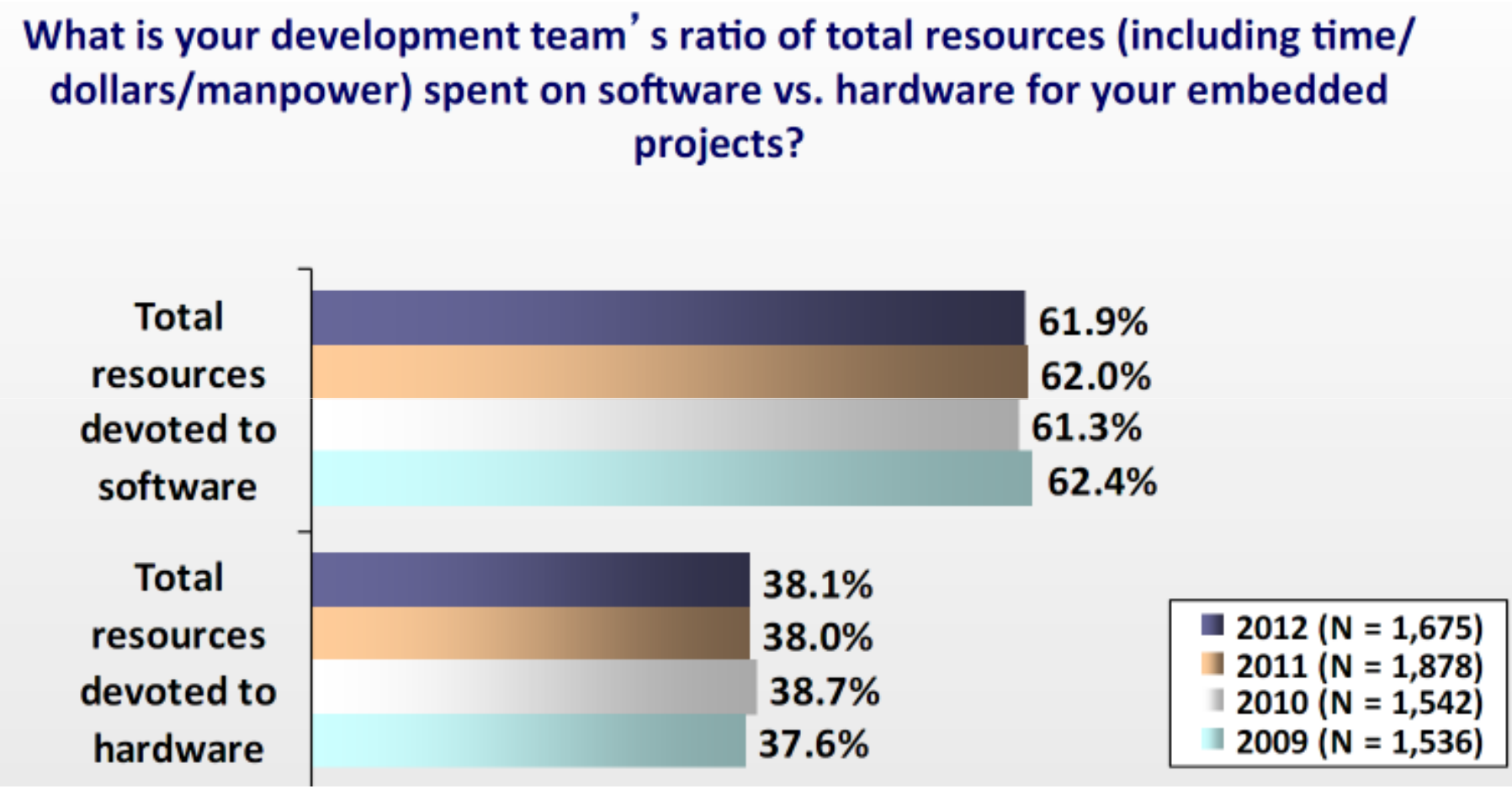
Beágyazott OS-ek és a normál OS-ek közötti különbségek

- Footprint
- Konfigurálhatóság
- Real-time viselkedés
- Nem az OS indítja az alkalmazást, hanem az alkalmazás az OS-t
- Nincs memória védelem

Beágyazott operációs rendszerek piackép

- Miért kell nekünk OS-ekkel foglalkozni
  - Beágyazott OS-ek elterjedtsége
  - Milyen RTOS-ek léteznek a piacon
  - Beágyazott OS és a normál OS-ek közötti különbség
  - Csoportosítás
  - Statisztikák

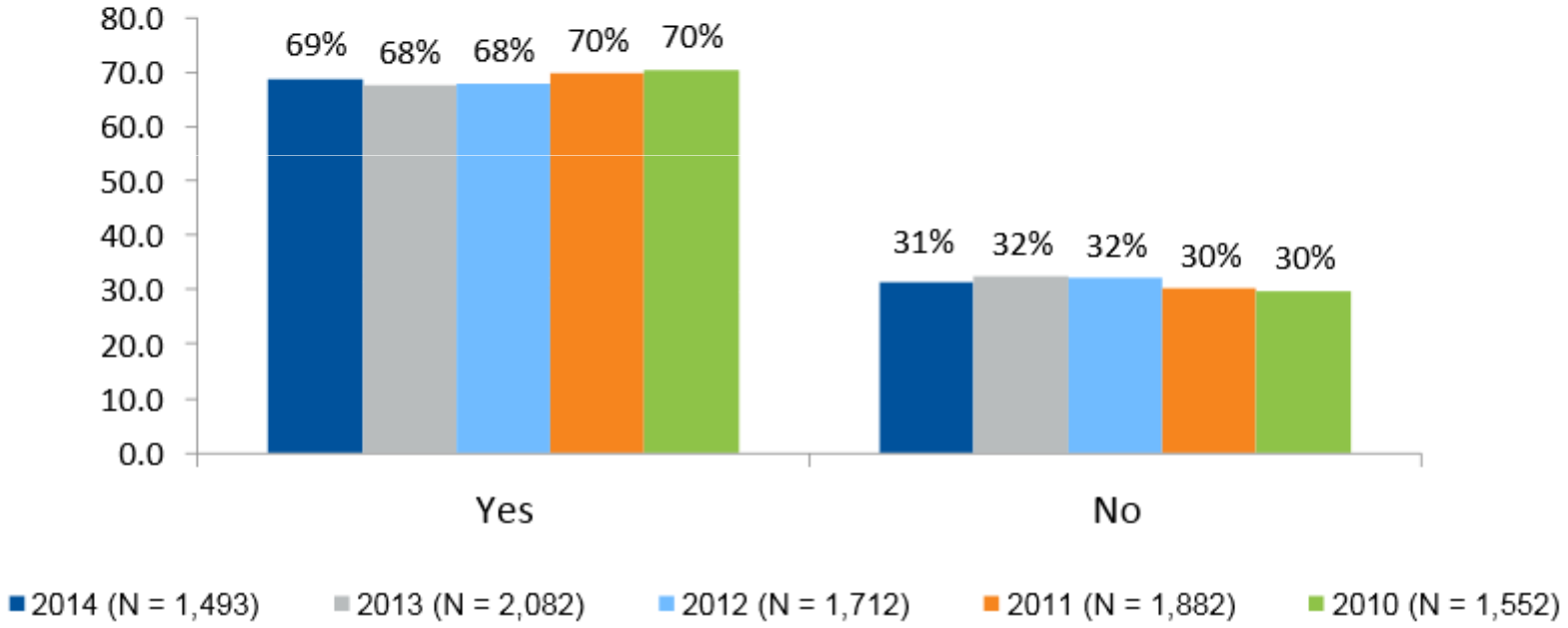
Miért fontos az operációs rendszer és egyéb szoftver tool-ok



A beágyazott OS-ek elterjedtsége

Does your current embedded project use an operating system, RTOS, kernel, software executive, or scheduler of any kind?

Hardly any change in usage of RTOS, kernels, execs, schedulers over past 5 years

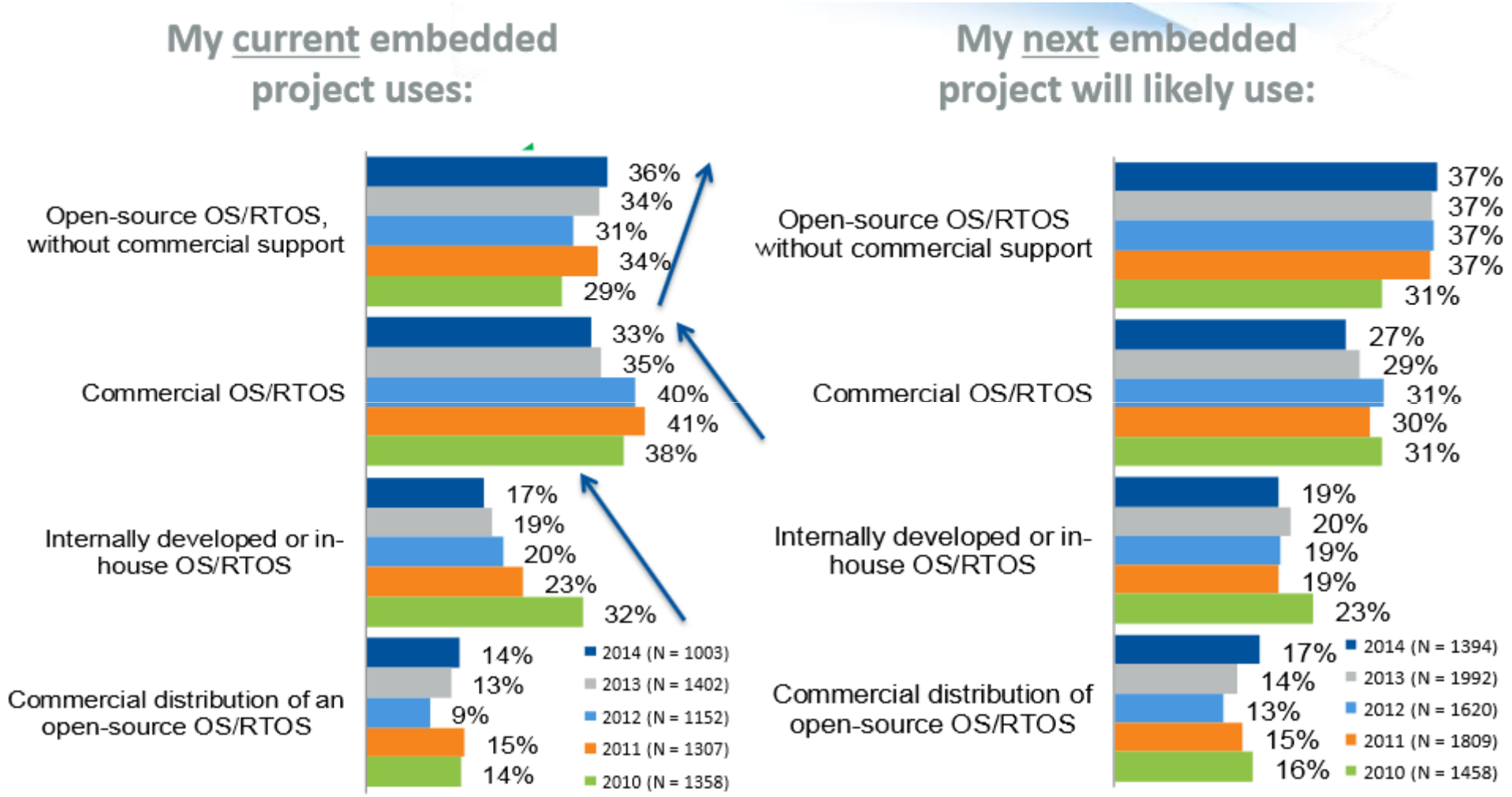


Beágyazott operációs rendszerek csoportosítása

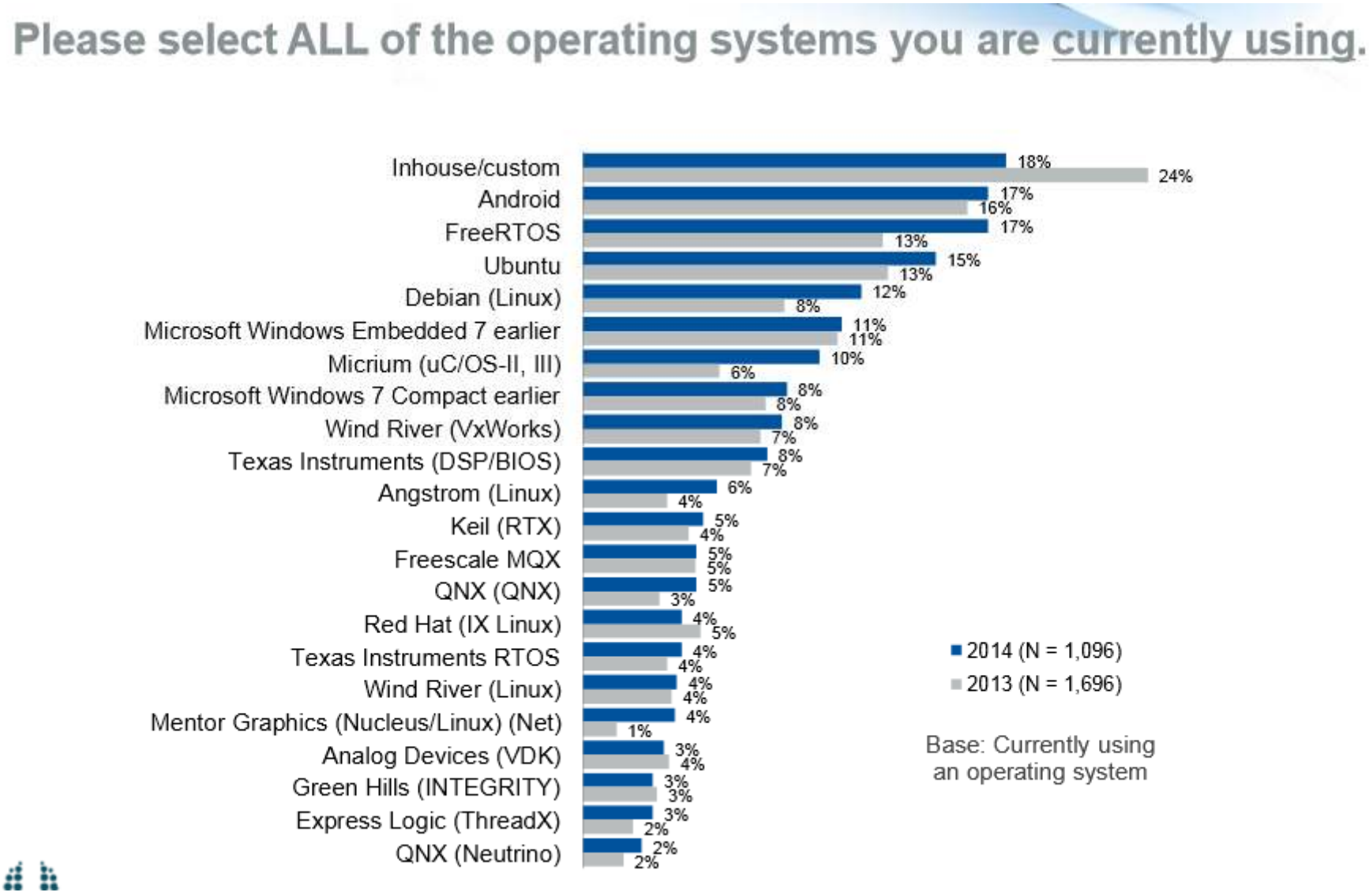
- Komplexitás szerint
  - Egyszerű kernelek 5-100k (uC-OS, FreeRTOS, CMX, eCos …)
  - Közepes komplexitású 100k-1M (eCos, Nucleus, QNX, Rtems, VxWorks …)
  - Nagy komplexitású 1M + (Linux, WinCe, WinXP Embedded …)

- Fizetős, vagy ingyenes
  - Ingyenes (FreeRTOS, eCos, Rtems, Linux)
  - Fizetős (VxWorks, uC-OS, Nucleus, QNX, WinCe)

Milyen OS-t használtak az elmúlt években?



Milyen OS-t használtak az elmúlt években?



µC-OS

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
- Publikálja: Embedded Systems Programming magazinnál 1992 legolvasottabb cikke

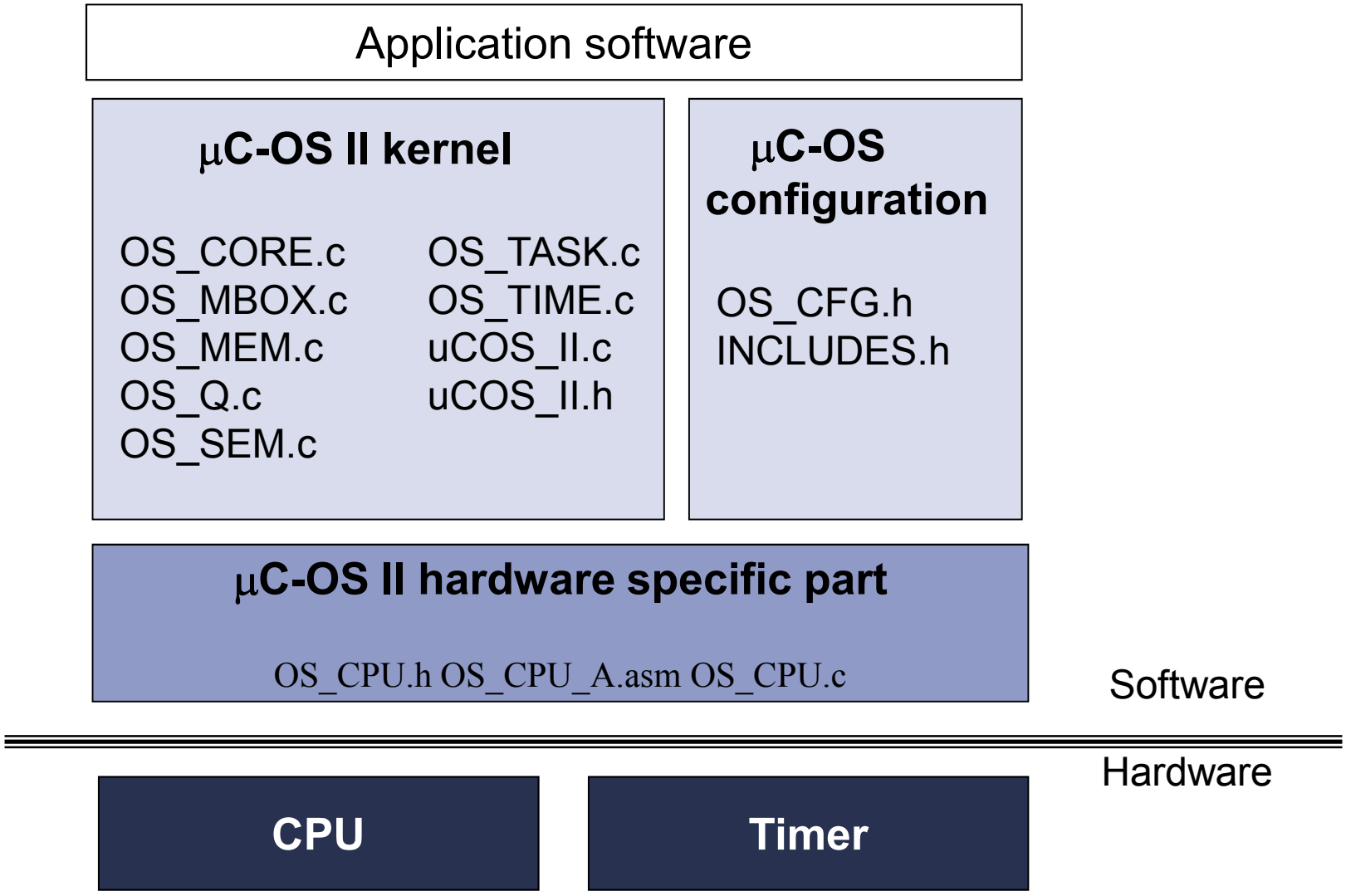
A µC/OS tulajdonságai

- 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

A µC/OS tulajdonságai

- nagyon jól dokumentált (µC/OS-III, 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)

A µC/OS architektúrája



A µC/OS konfigurálása

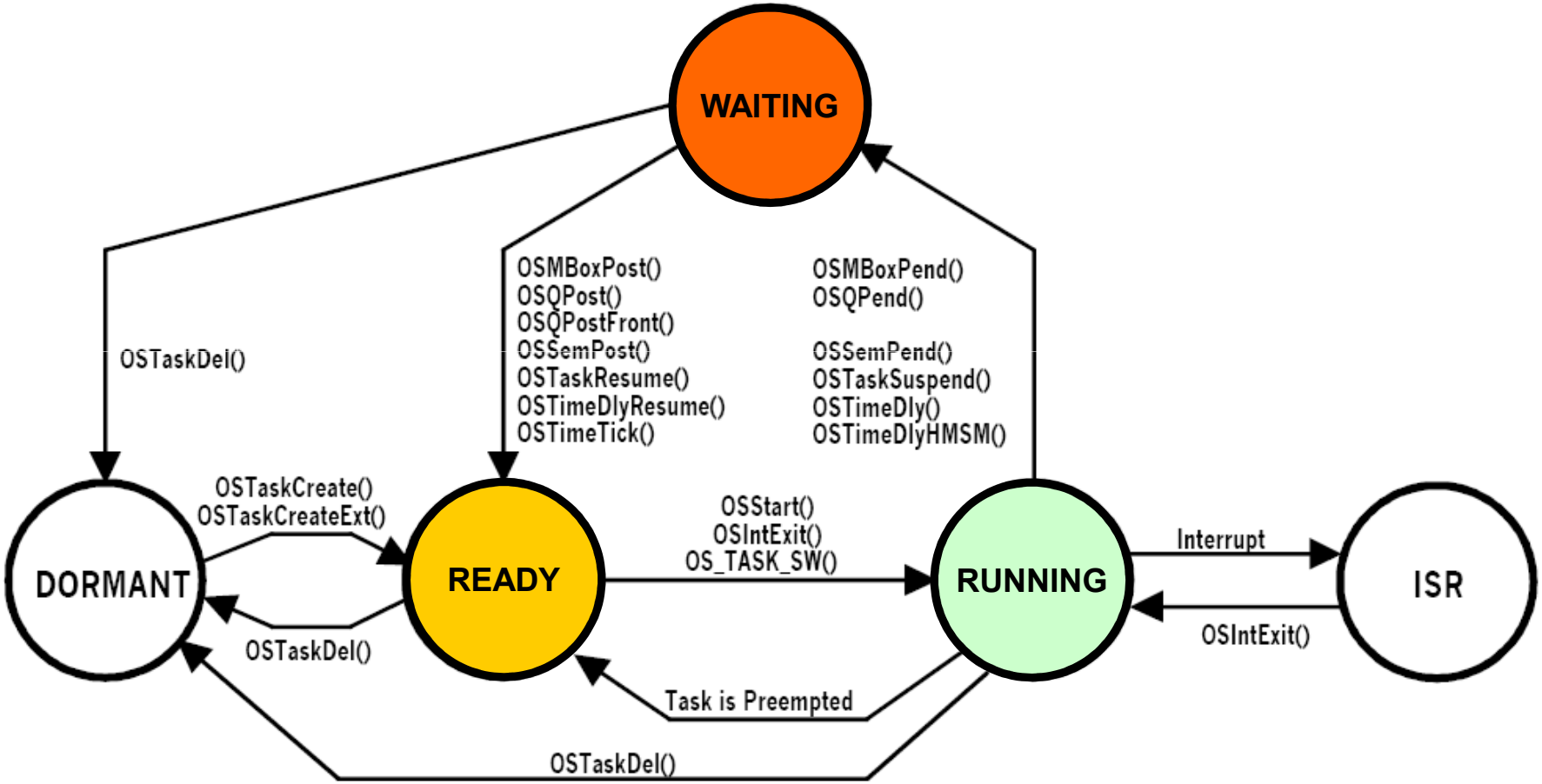
OS_CFG.h

```
/* -------------------- 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            1    /*     Include code for OSMboxDel()                        */
#define OS_MBOX_POST_EN           1    /*     Include code for OSMboxPost()                       */
#define OS_MBOX_POST_OPT_EN       1    /*     Include code for OSMboxPostOpt()                    */
#define OS_MBOX_QUERY_EN          1    /*     Include code for OSMboxQuery()                      */
```

OS_MBOX.c

```
#if OS_MBOX_EN > 0
        .........
        #if OS_MBOX_ACCEPT_EN > 0
        .........
        #endif
        .........
        #if OS_MBOX_DEL_EN > 0
        .........
        #endif
#endif
```

A µC/OS taszk állapotai



FreeRTOS

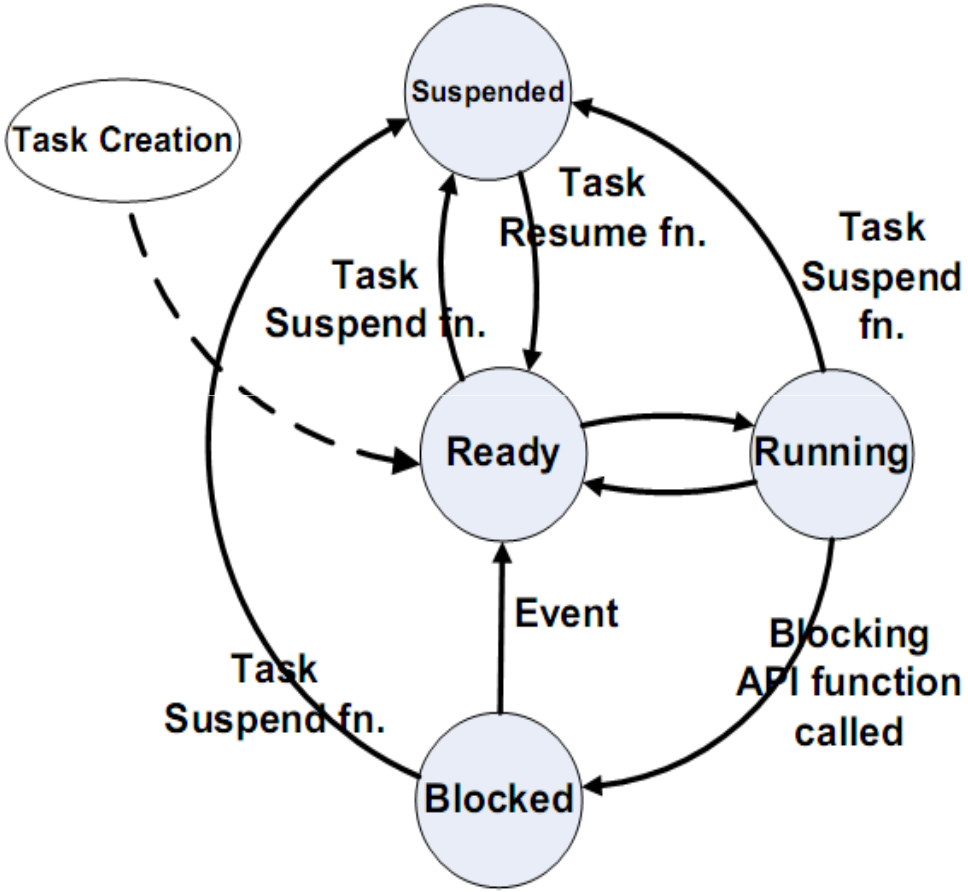
FreeRTOS

- Nyílt forráskódú egyszerű kernel
  - www.freertos.org

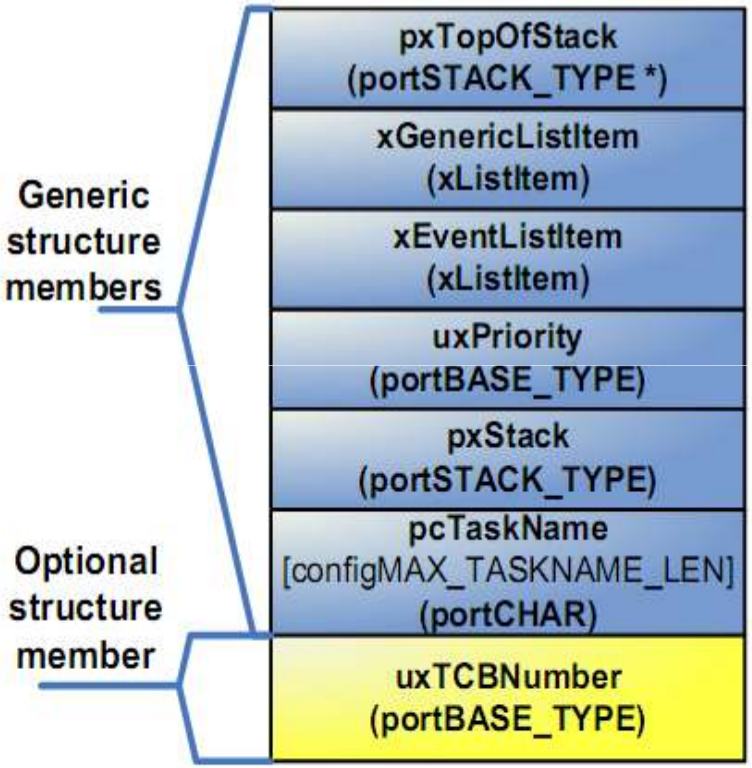
- Az elmúlt időszak legdinamikusabban fejlődő könnyű kategóriájú kernelje
- 2008-ban több mint 77,500 letöltés.
- Portok:
  - ARM7, ARM9, CortexM3
  - Atmel AVR, AVR32
  - PIC18, PIC24, dsPIC, PIC32
  - Microblase…

FreeRTOS taszkok

- Taszkok
  - Saját stack
  - Konfigurálni kell hogy mennyit használunk
  - Magas prioritás szám magas prioritás
  - Idle task 0-s prioritás



FreeRTOS taszk control block

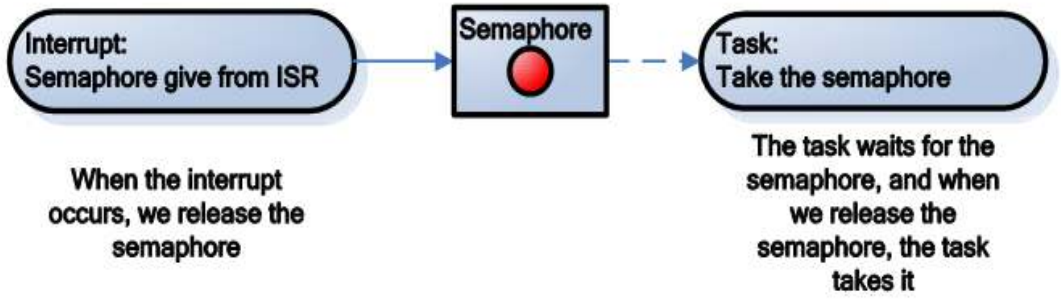


FreeRTOS taszkok kezelése

```
void vOtherFunction( void )
{
xTaskHandle xHandle;

// Create the task, storing the handle.
xTaskCreate( vTaskCode, "NAME", STACK_SIZE, NULL,
    tskIDLE_PRIORITY, &xHandle );
// Use the handle to delete the task.
vTaskDelete( xHandle );
}
```

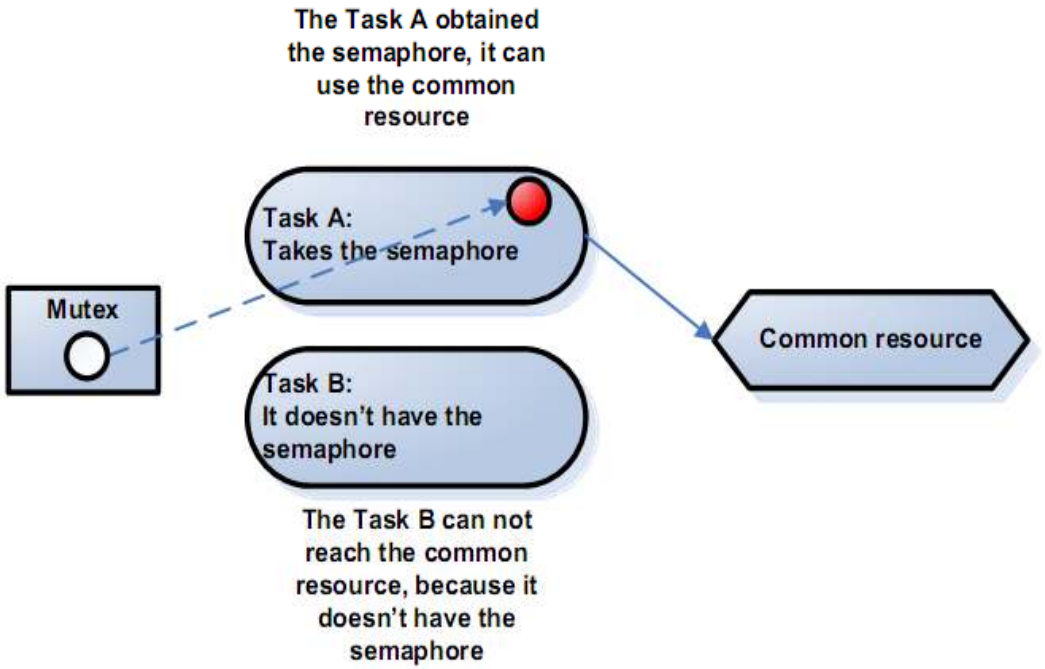
FreeRTOS taszk szinkronizáció

- Bináris szemaforok
  - vSemaphoreCreateBinary
  - xSemaphoreTake
  - xSemaphoreGive
  - xSemaphoreGiveFromISR
- Számláló szemaforok
  - Nem csak 0-1 lehet az értéke, hanem egy egész szám.
  - Erőforrás menedzsment ahol egyszerre többen férhetnek hozzá.
  - Esemény számolás.



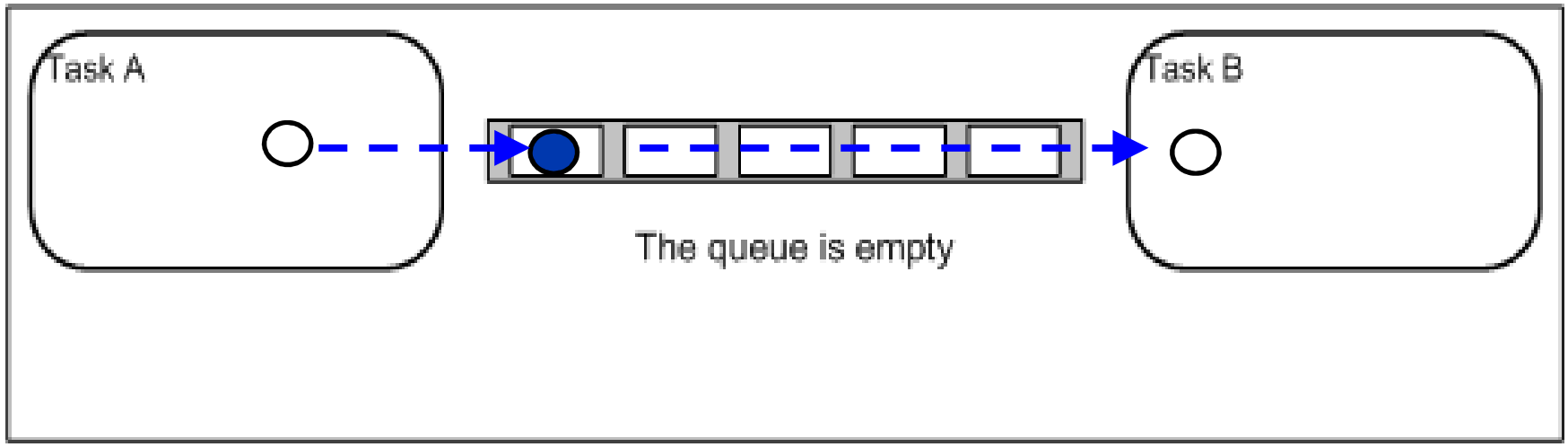
FreeRTOS taszk szinkronizáció mutex

- Mutexek
  - Prioritás inverzió ellen védettek
  - Ne használjuk megszakításból



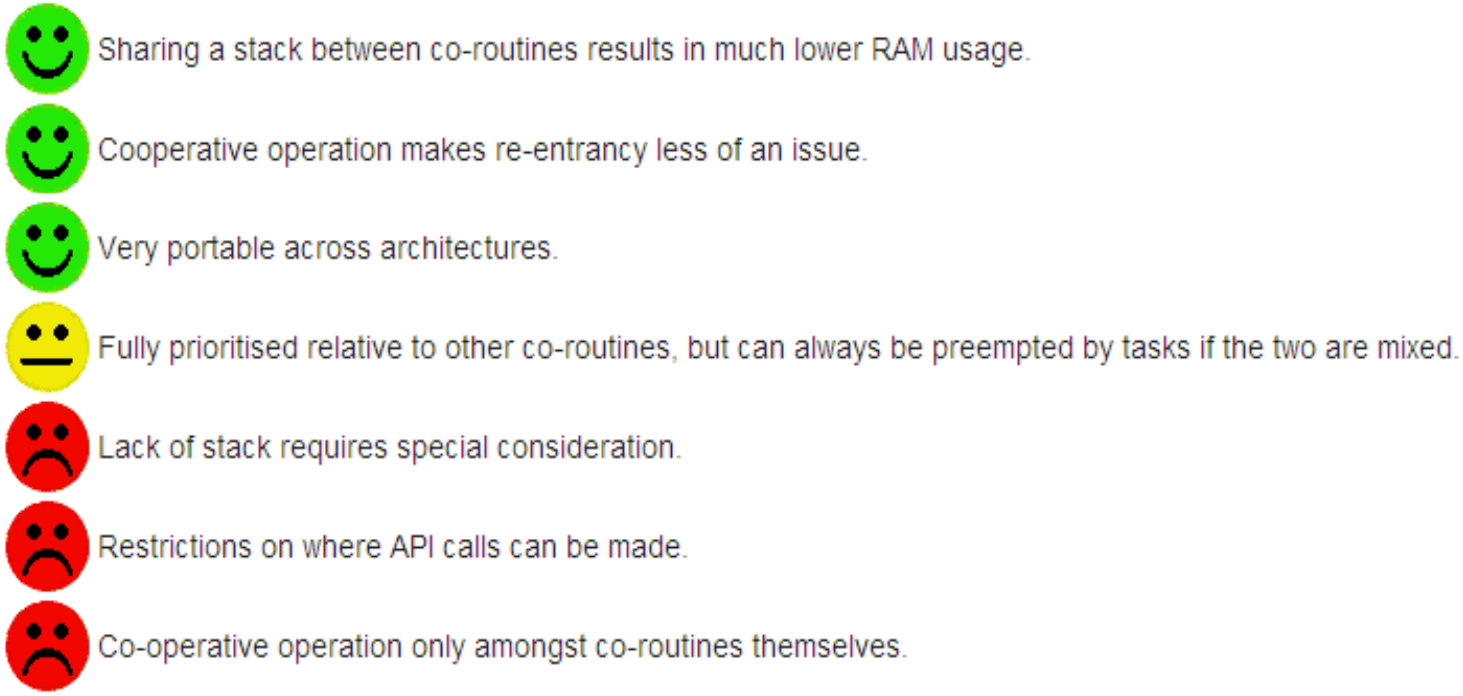
FreeRTOS Queue

- Queue
  - Üzenetek küldése folyamatok között
  - xQueueCreate
  - xQueueSend
  - xQueueReceive
  - xQueueSendFromISR



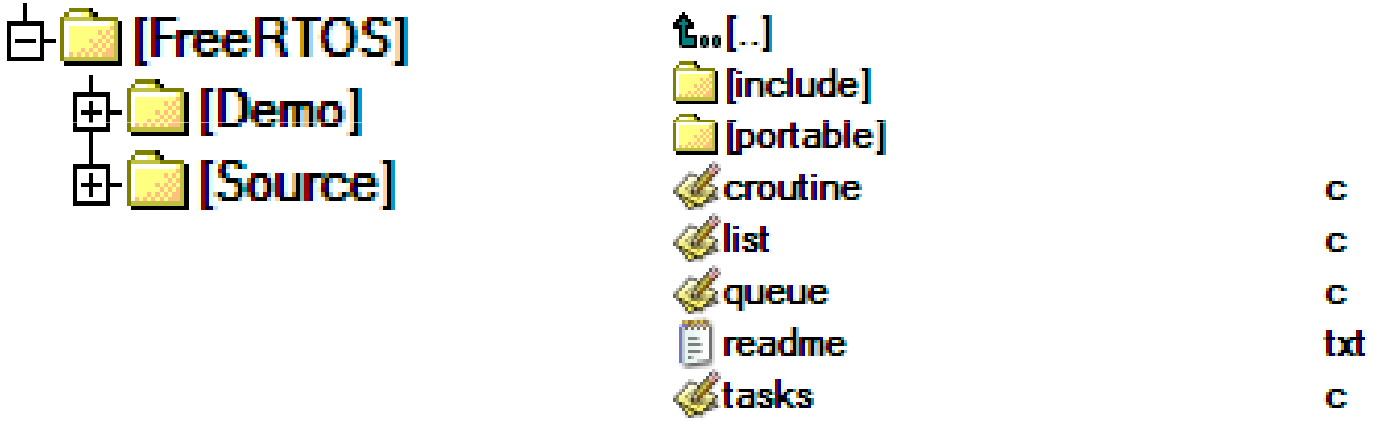
FreeRTOS CoRutin

- Egyszerűbb mint egy taszk
- Függvény sor alapú, nem preemtív ütemető



FreeRTOS forráskód elrendezés

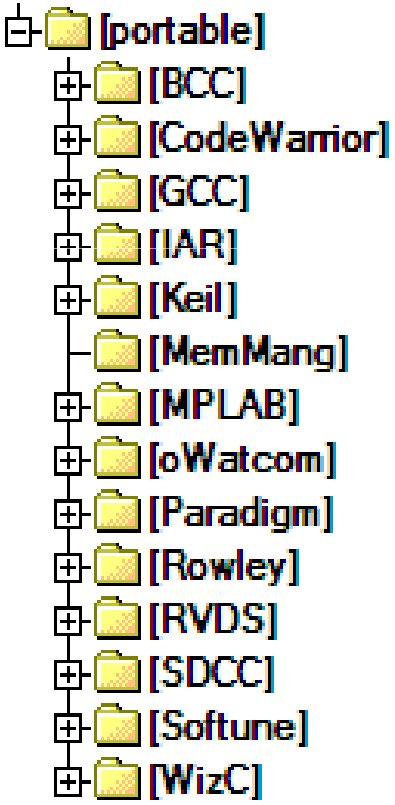
- Nagyon egyszerű alap kernel
  - tasks.c, queue.c, list.c. File-ok az Source könyvtárban



  - Ezek a file-ok tartalmazzák az alap taszk létrehozást és szinkronizációt.

FreeRTOS portolás

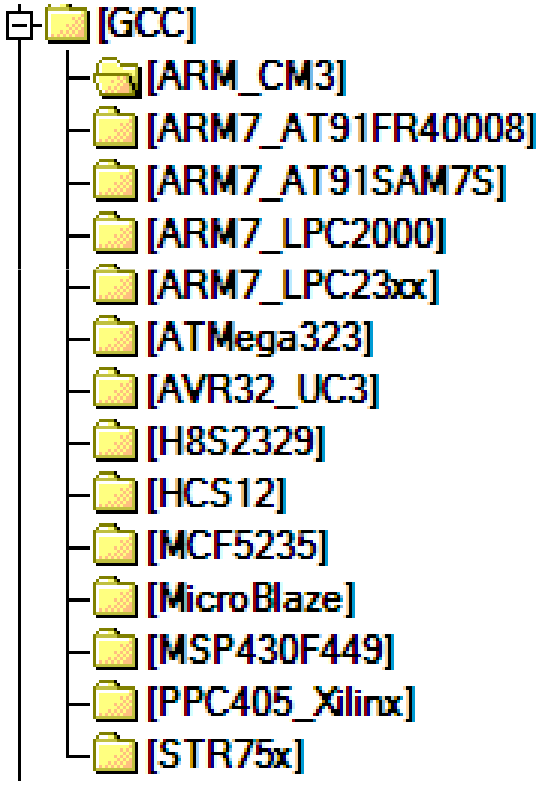
- A port specifikus kód részek a portable directoryban



- Task váltás, Sys tick timer, Critical szekcióba lépés és elhagyás
- Toolchain szerint rendezve

GCC specifikus részek

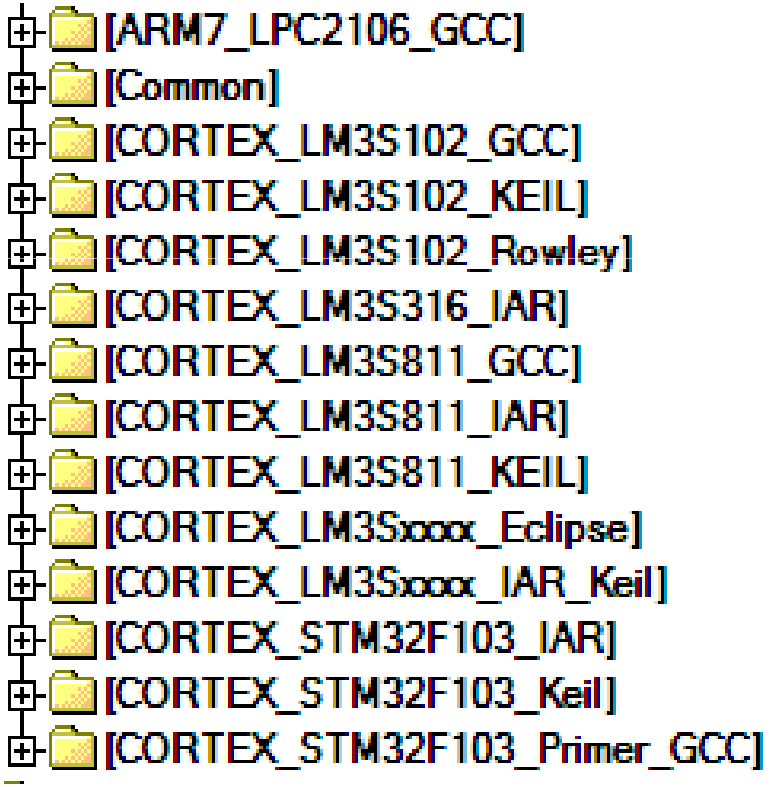
- GCC specifikus részek



- Egy port file

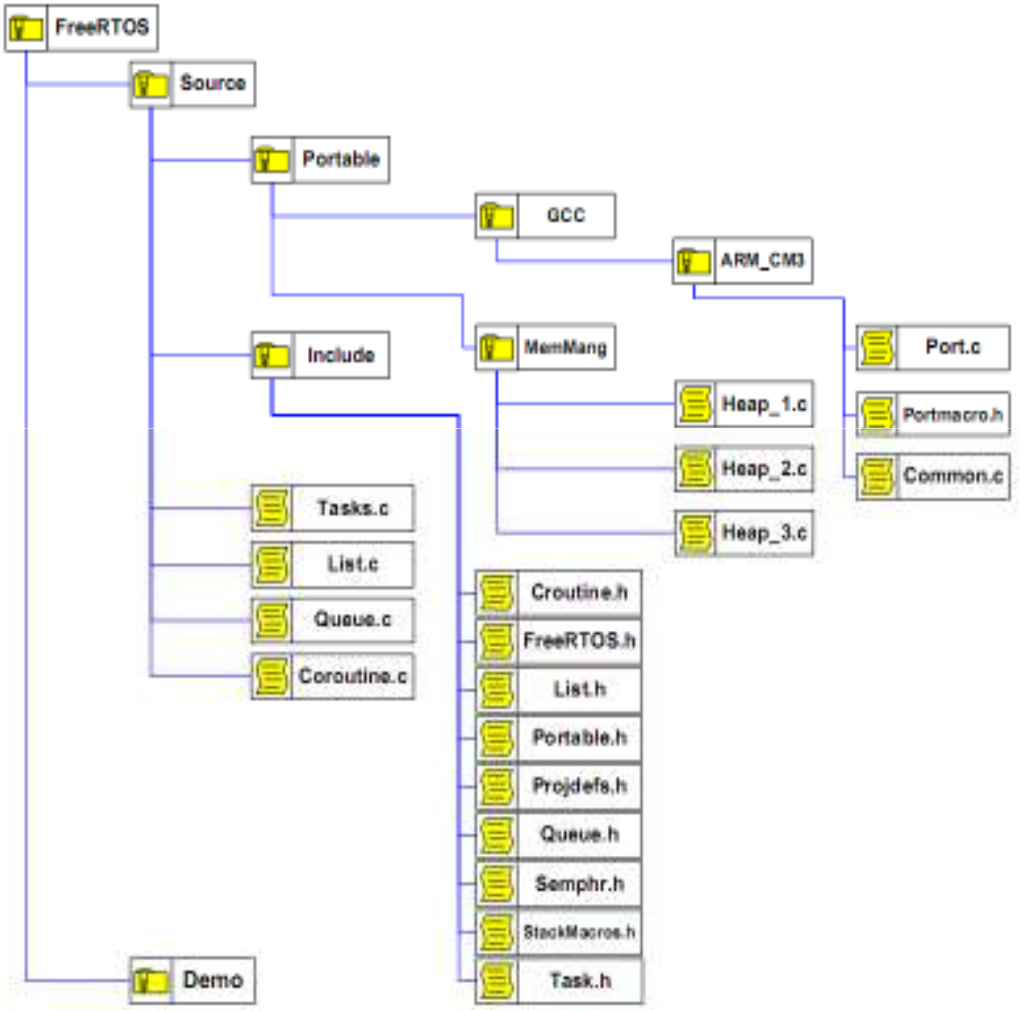
GCC demó projectek

- Kártya és fordító specifikus részek



- Startup kód
- Kártya specifikus kód

A FreeRTOS könyvtárszerkezete *áttekintés*



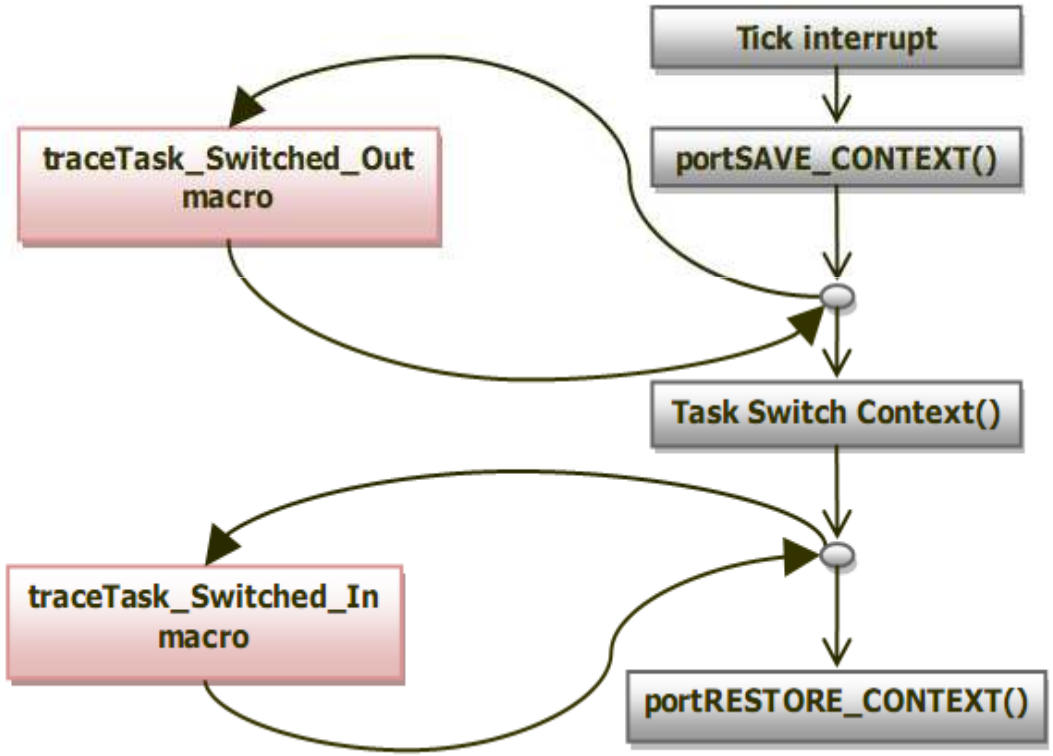
FreeRTOS konfiguráció

- FreeRTOS_Config.h

```
/*-----------------------------------------------------------
 * Application specific definitions.
 *
 * These definitions should be adjusted for your particular hardware and
 * application requirements.
 *
 * THESE PARAMETERS ARE DESCRIBED WITHIN THE 'CONFIGURATION' SECTION OF THE
 * FreeRTOS API DOCUMENTATION AVAILABLE ON THE FreeRTOS.org WEB SITE.
 *-----------------------------------------------------------*/

#define configUSE_PREEMPTION		1
#define configUSE_IDLE_HOOK		0
#define configUSE_TICK_HOOK		0
#define configCPU_CLOCK_HZ		( ( unsigned portLONG ) 20000000 )
#define configTICK_RATE_HZ		( ( portTickType ) 1000 )
#define configMINIMAL_STACK_SIZE	( ( unsigned portSHORT ) 70 )
#define configTOTAL_HEAP_SIZE		( ( size_t ) ( 7000 ) )
#define configMAX_TASK_NAME_LEN		( 10 )
#define configUSE_TRACE_FACILITY	0
#define configUSE_16_BIT_TICKS		0
#define configIDLE_SHOULD_YIELD		0
#define configUSE_CO_ROUTINES		0

#define configMAX_PRIORITIES		( ( unsigned portBASE_TYPE ) 5 )
#define configMAX_CO_ROUTINE_PRIORITIES ( 2 )
```

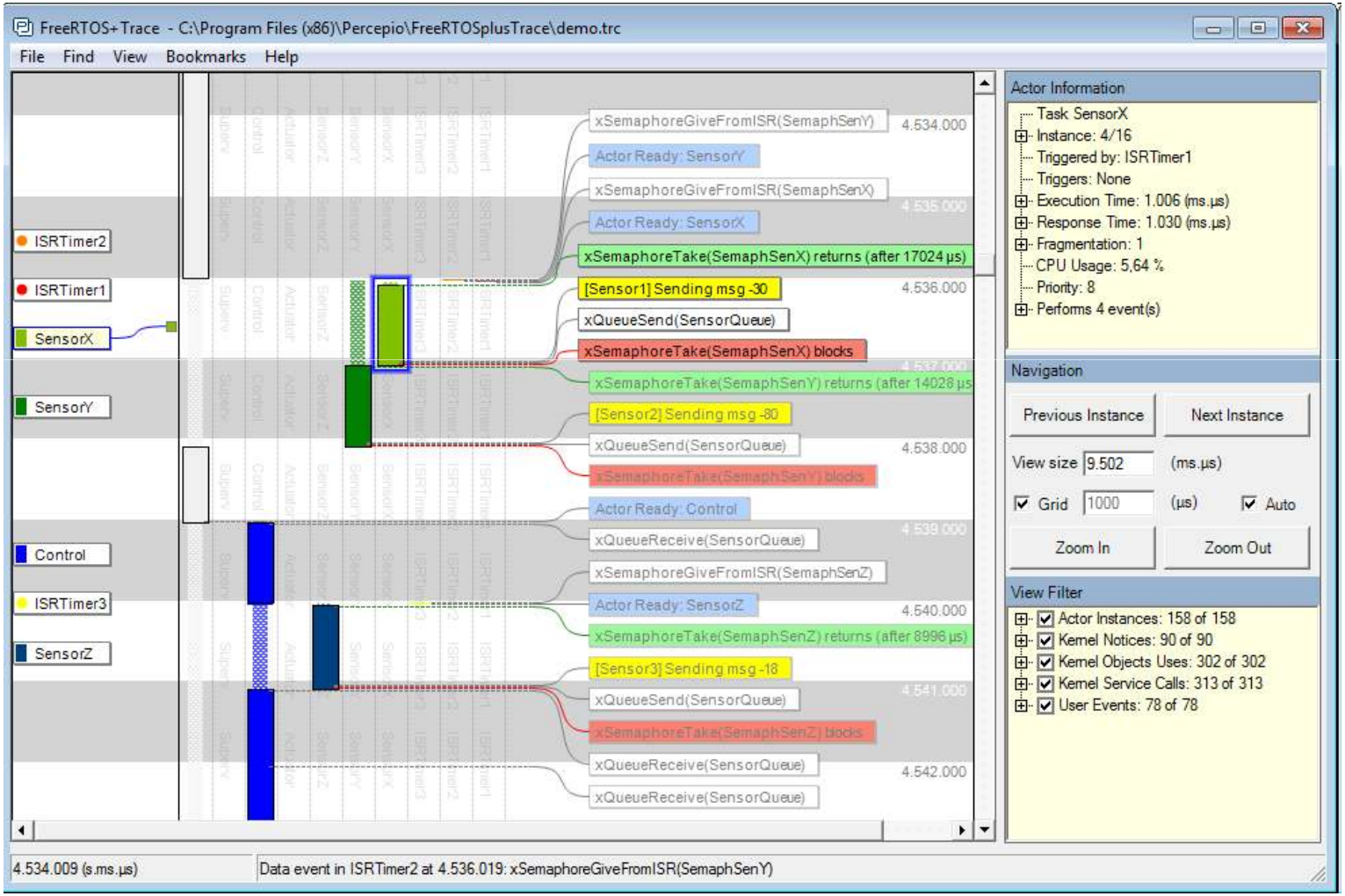
Trace hookok

- Gyakorlatilag minden fontosabb belső lépéshez tartozik



Trace alkalmazás

- Teljes szinkronizációs történet megjelenés



FreeRTOS kereskedelmi változatok

- OpenRTOS
  - Kereskedelmi szinten támogatott verzió
  - USB, File rendszer, TCP-IP támogatás
- SafeRTOS
  - SIL3 szintű tanúsítvány
  - Stellaris LM3S9B96 Microcontrollerbe ROM szinten beépítve

eCos
(Embedded Configurable Operating System)

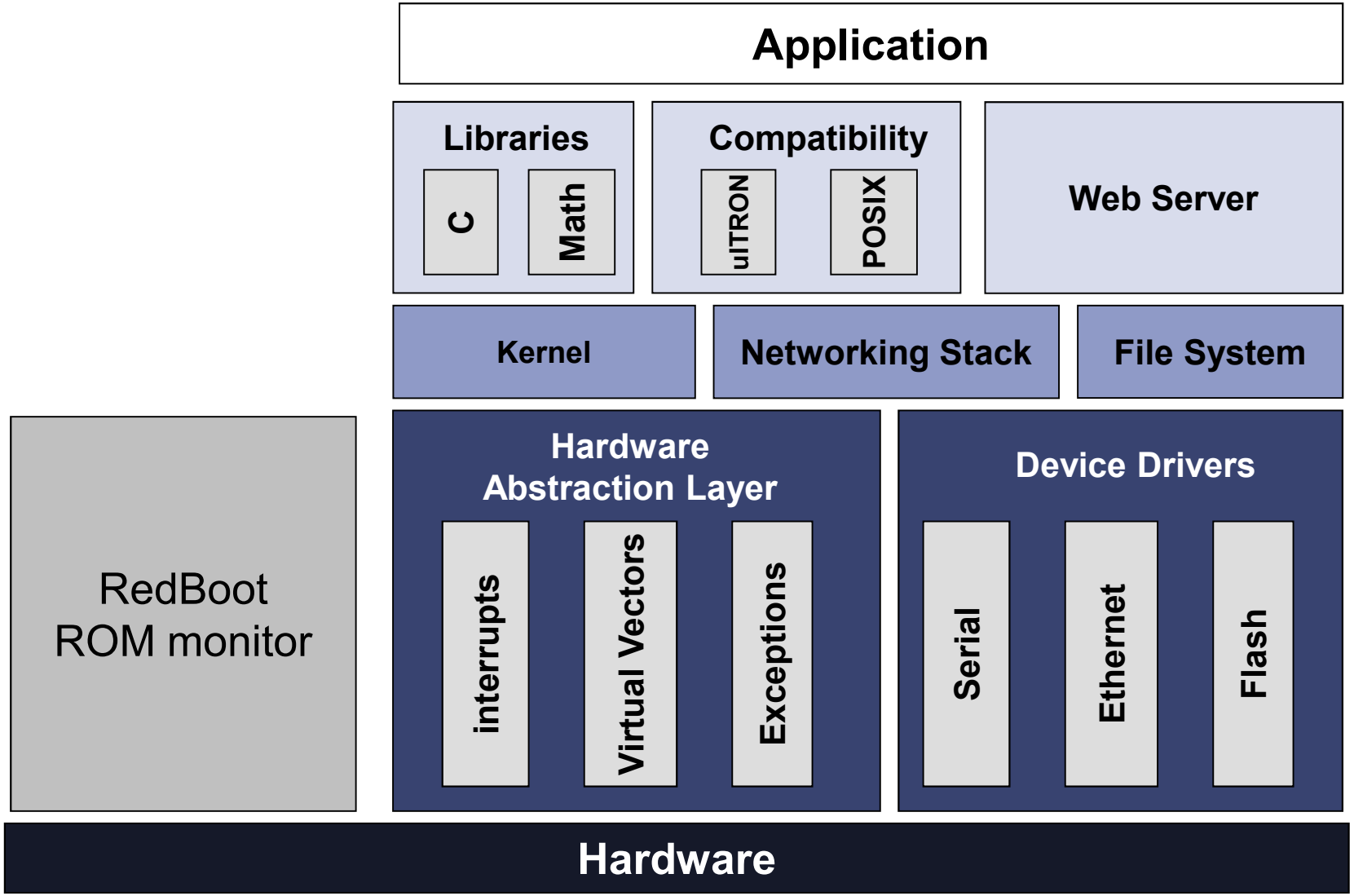
Bevezetés

Az 1998-ban kifejlesztett eCos létrehozóinak filozófiája az volt, hogy egy olyan nyílt forráskódú Real-Time Operációs rendszert készítsenek, amely széleskörű konfigurációs lehetőséget nyújt a felhasználónak anélkül, hogy az operációs rendszer forráskódjának akár csak egy sorát meg kéne változtatni.

- 200+ konfigurálási opció
- Kompatibilitási lehetőségek
  - POSIX, EL/IX, µltron
- Előnyök a Linux-hoz képest
  - Kisebb overhead (40kByte körül már futásképes)
  - Real-Time
  - A RedBoot bootmonitor alapja

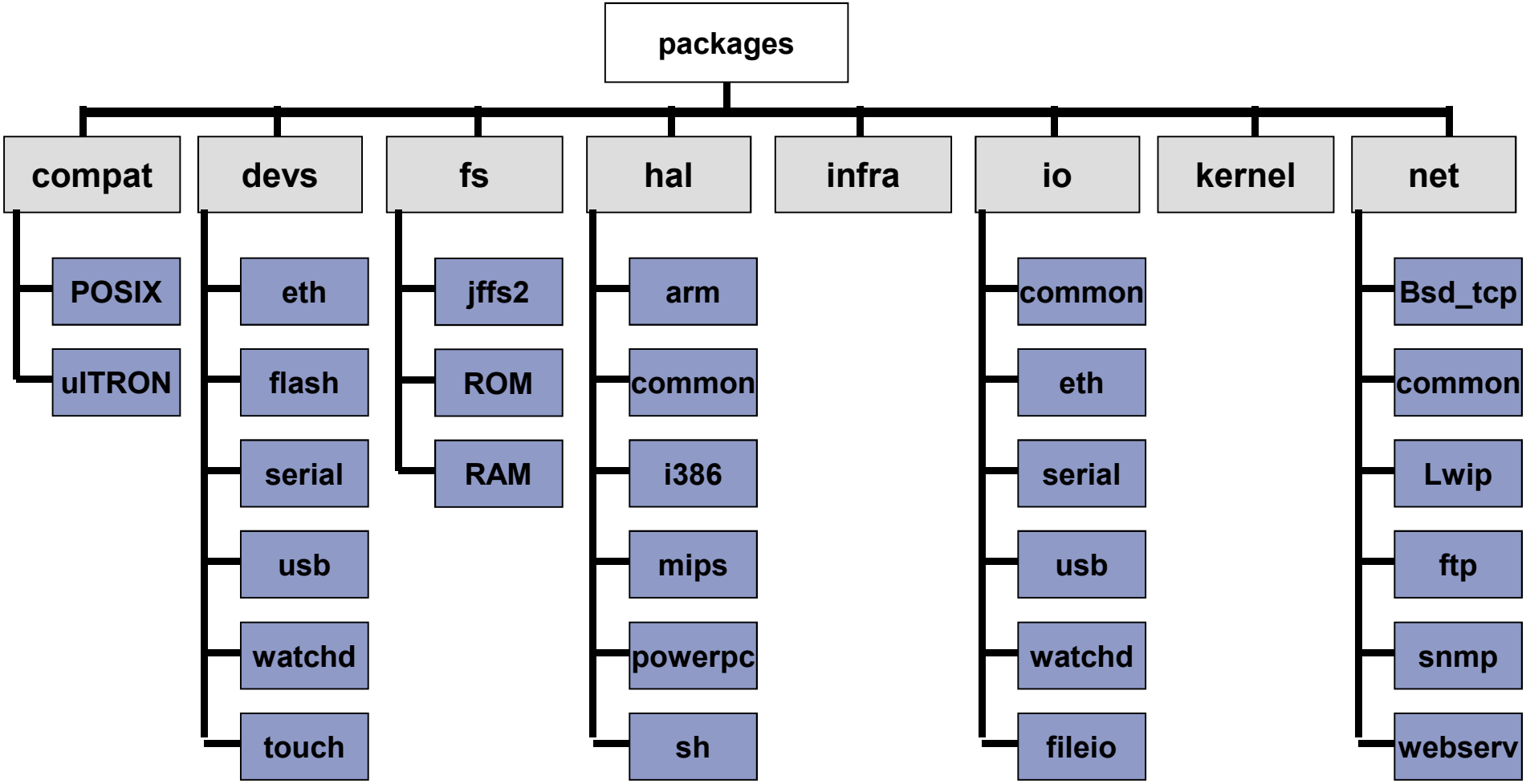
eCos architectúra

Széleskörű hardware platform támogatás



eCos Repository *(A komponens raktár)*

A Component Repository egy könyvtárstruktúra ami tartalmazza az összes csomagot



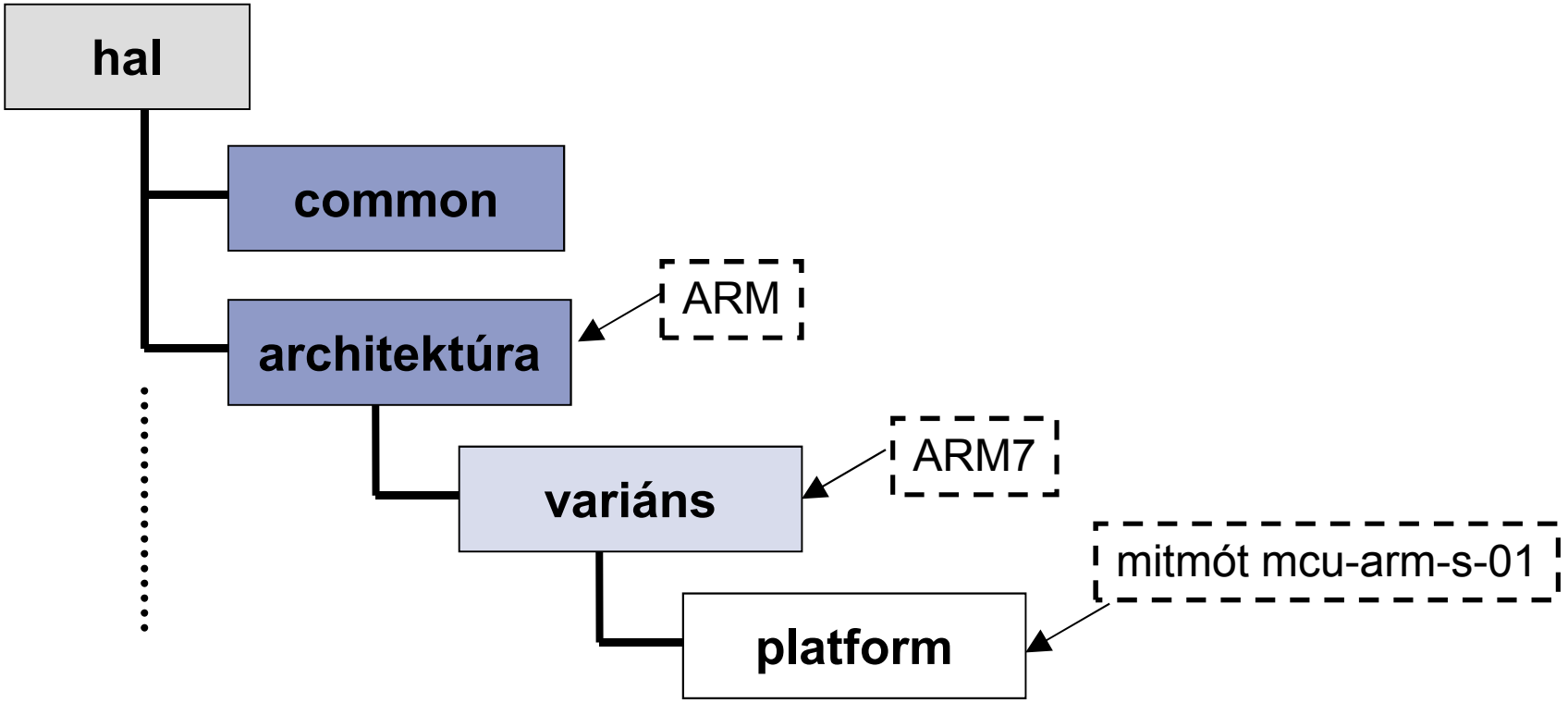
eCos HAL

A HAL (Hardware Abstraction Layer) szeparálja el egymástól a platform architektúrájától függő részeket az általános hardware független részektől.

A HAL Gyakorlatilag egy szoftverréteg általánosított API-val (Application Programing Interface) ami magába zárja a működéshez szükséges hardware műveleteket.

A HAL könyvtár struktúrája

A HAL könyvtár struktúrája egy hierarchikus elrendezés, ami architektúra, variáns, target alapelemekre bontja a hardware függő kódot.



Az eCos component framework *(keretrendszer)*

Ez gyakorlatilag egy olyan eszközgyűjtemény, ami lehetővé teszi a felhasználónak, hogy menedzselje, illetve konfigurálja az eCos rendszer különféle csomagjait

- Configuration Tool
  - Command line / graphical
- Package Administration Tool
  - Komplett csomagok (esetleg új) hozzáadása eltávolítása az eCos komponens raktárhoz/ból
- Memory Layout Tool
  - Új platformra való portolásnál használják a memória kiosztás megadására.

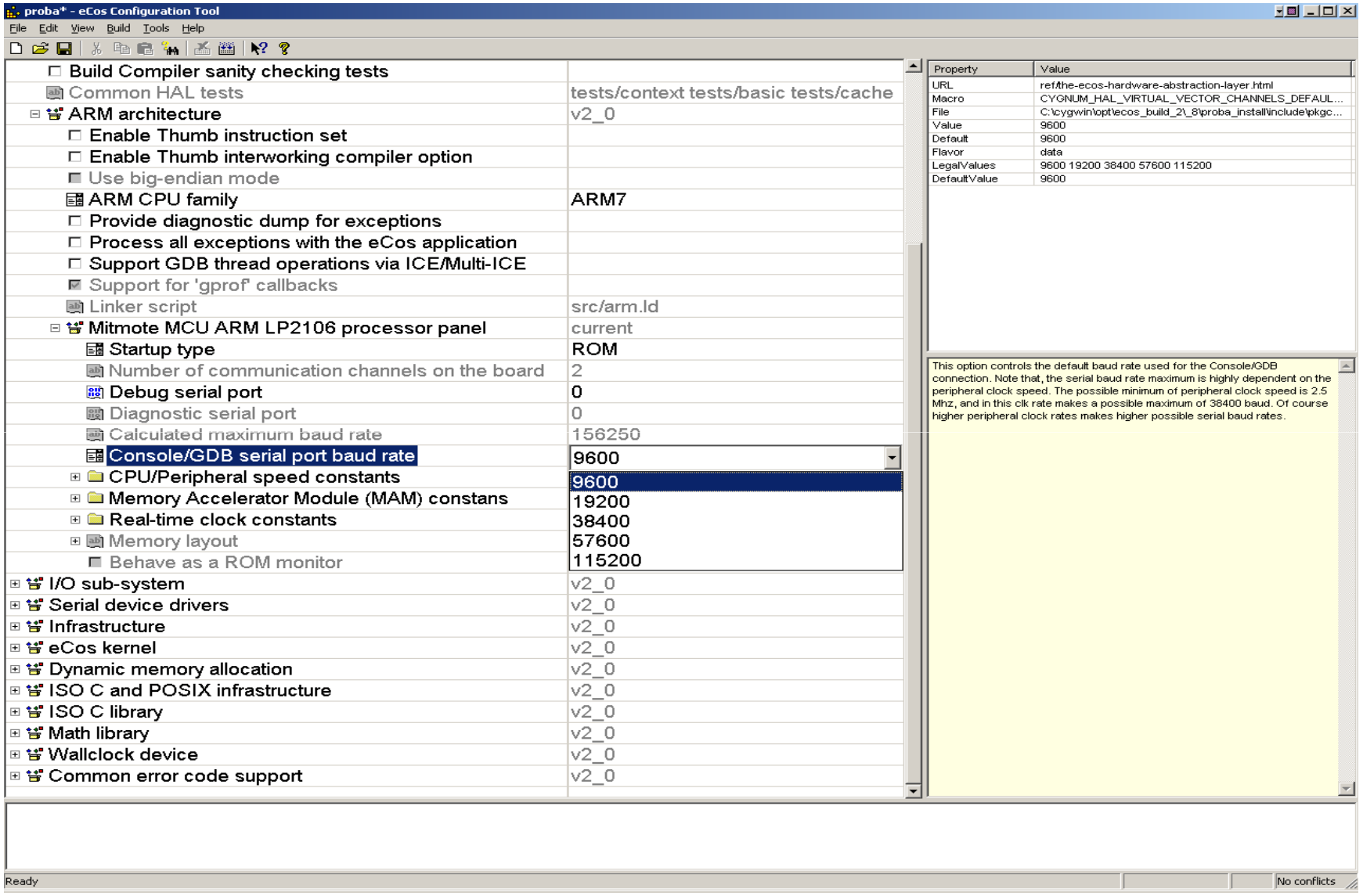
A Configuration Tool

Ezzel lehet finoman hangolt konfigurációkat készíteni, a Configuration Tool gyakorlatilag a CDL (Component Description Language) file-ok Grafikus megjelenítése

- CDL (Component Description Language)
  - Az eCos talán legfontosabb része, egy olyan script nyelv, ami leírja egy csomag tartalmát illetve konfigurálási lehetőségeit
  - Minden csomaghoz tartozik legalább egy cdl file
  - A CDL file-ok definició jelennek meg később a C file-ok #ifdef részeiben

```
cdl_option CYGNUM_HAL_VIRTUAL_VECTOR_CHANNELS_DEFAULT_BAUD {
    display       "Console/GDB serial port baud rate"
    flavor        data
    legal_values  9600 19200 38400 57600 115200
    default_value 38400
    description   "
        This option controls the default baud rate used for the
        Console/GDB connection."
}
```

A Configuration Tool



Mit hoz magával egy RTOS?

- Általában célszerű egy demókártya, processzor használatánál ezzel kezdeni
  - Összeállított fejlesztőkörnyezet
    - GCC fordítások, startup file-ok, make file-ok
  - Egyes külső programcsomagok integrálva vannak
    - TCP/IP
    - Flash file rendszer
  - Mellesleg még párhuzamos programozást is alkalmazhatunk

Újdonságok CMSIS RTOS?

- Általános felület RTOS absztrakcióra