ARM Cortex magú mikrovezérlők
9. RTOS alapok

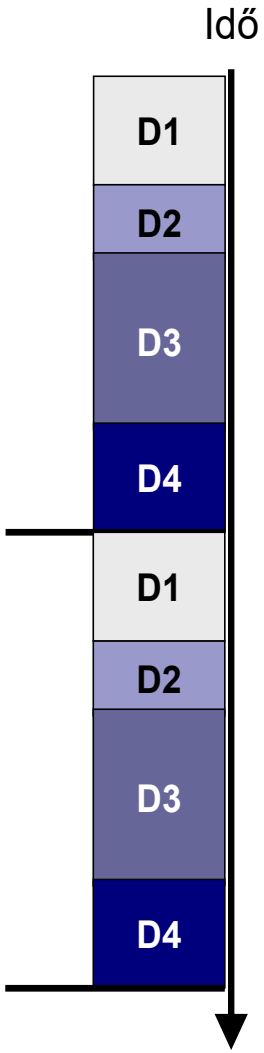
Scherer Balázs

Méréstechnika és
Információs Rendszerek
Tanszék

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

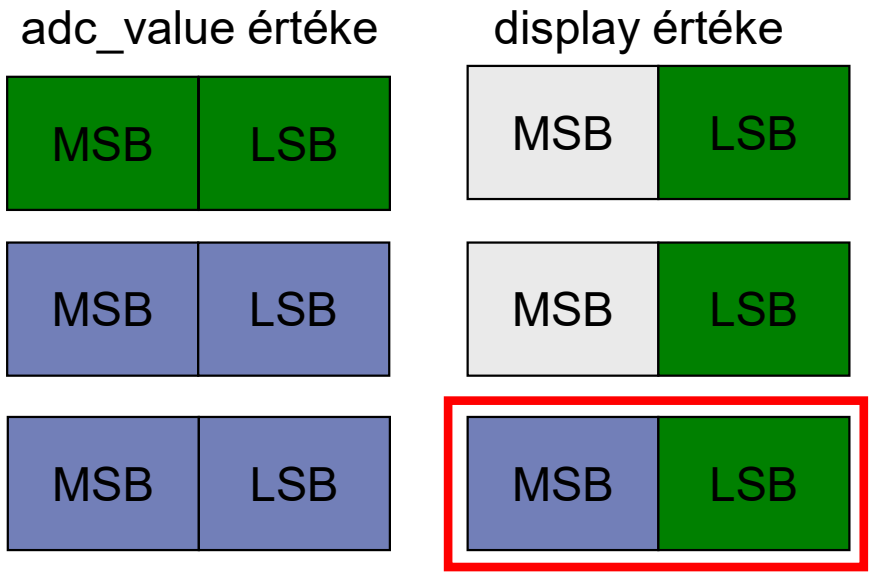
```
external unsigned short adc_value;
INTERRUPT(SIG_ADC )
{
  // Az AD kiolvasása
   adc_value = read_adc();
}
```

Elkezdődik a display = adc_value (nem egy asm utasítás)

ADC IT megszakítja a főprogramot a két érték másolása közben

Befejeződik a display = adc_value (nem egy asm utasítás)

Time



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.

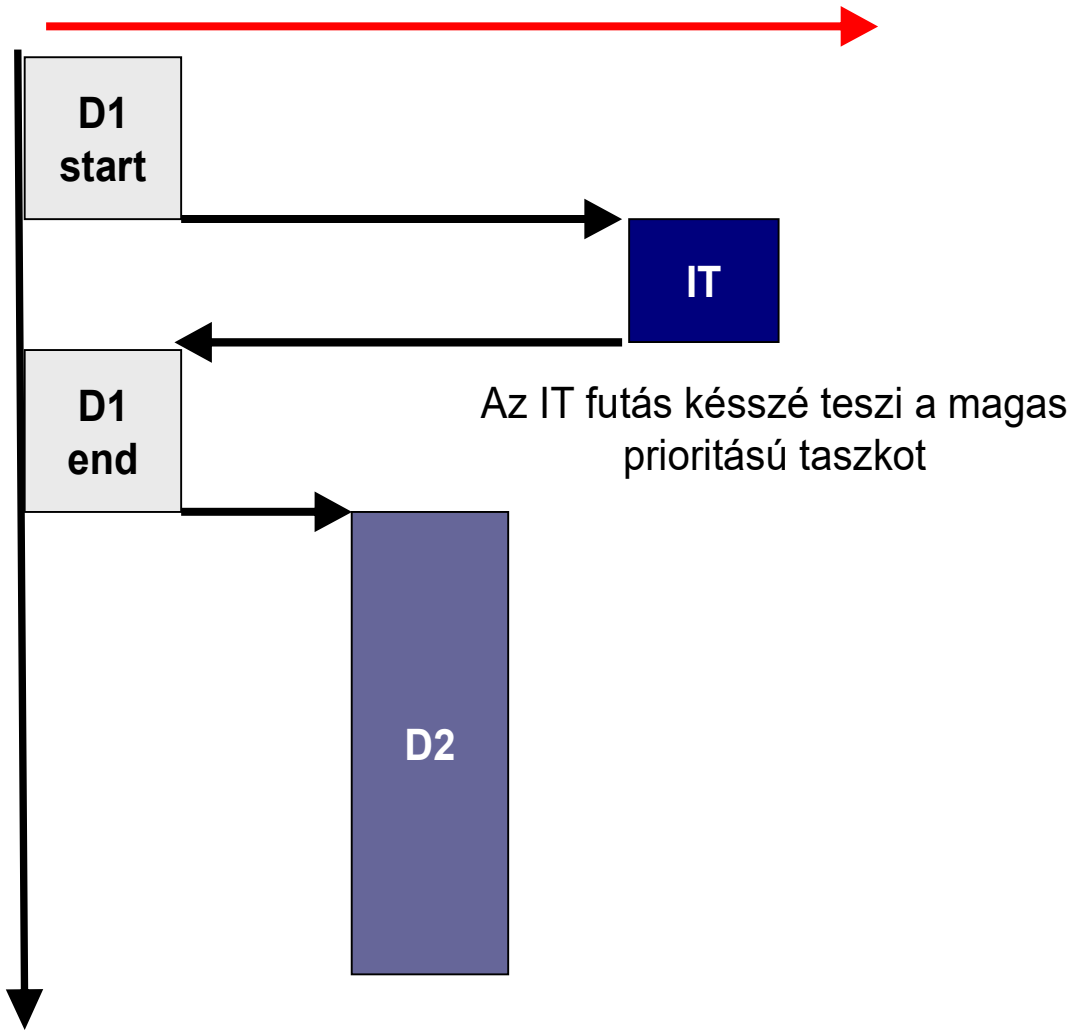
- Függvénysor alapú nem preemptív ütemezés

```
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.

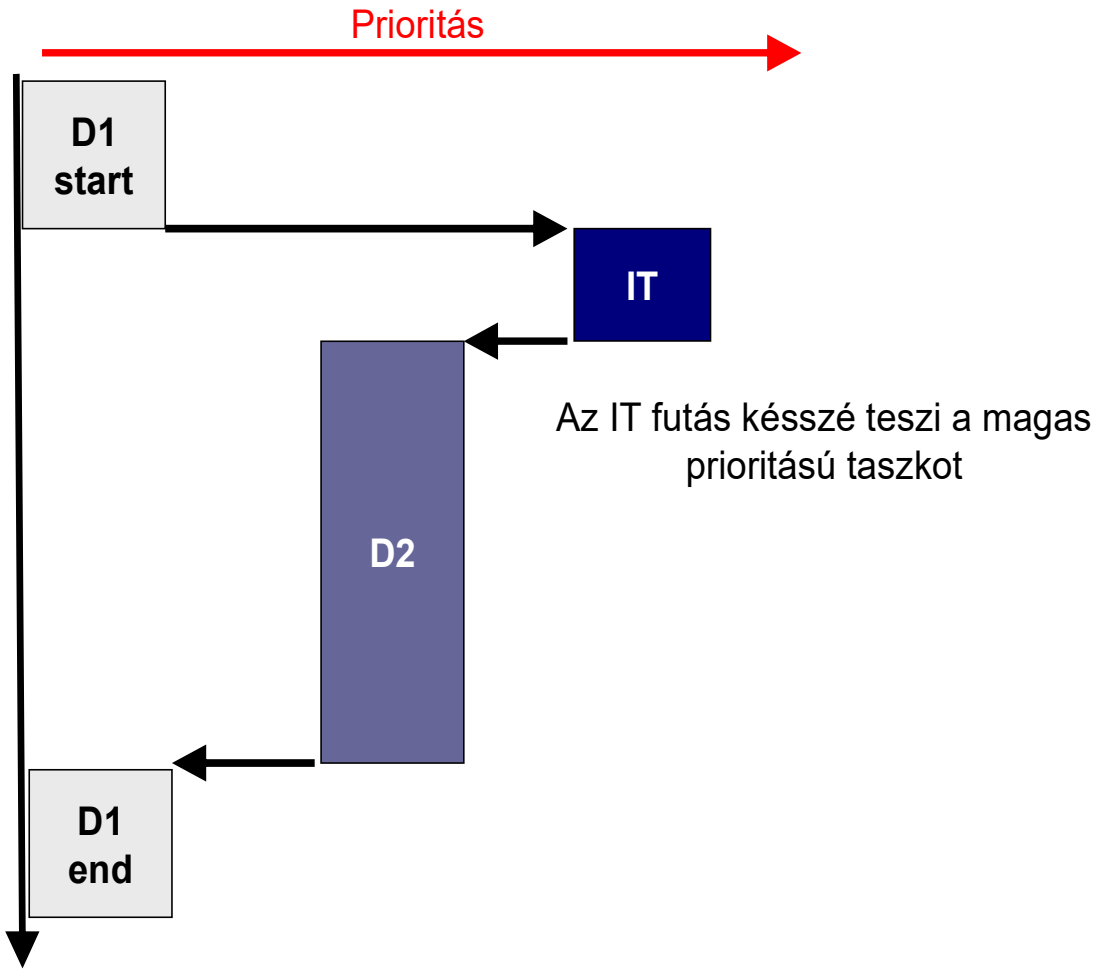
- Real Time OS, preemptív ütemezés

```
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.

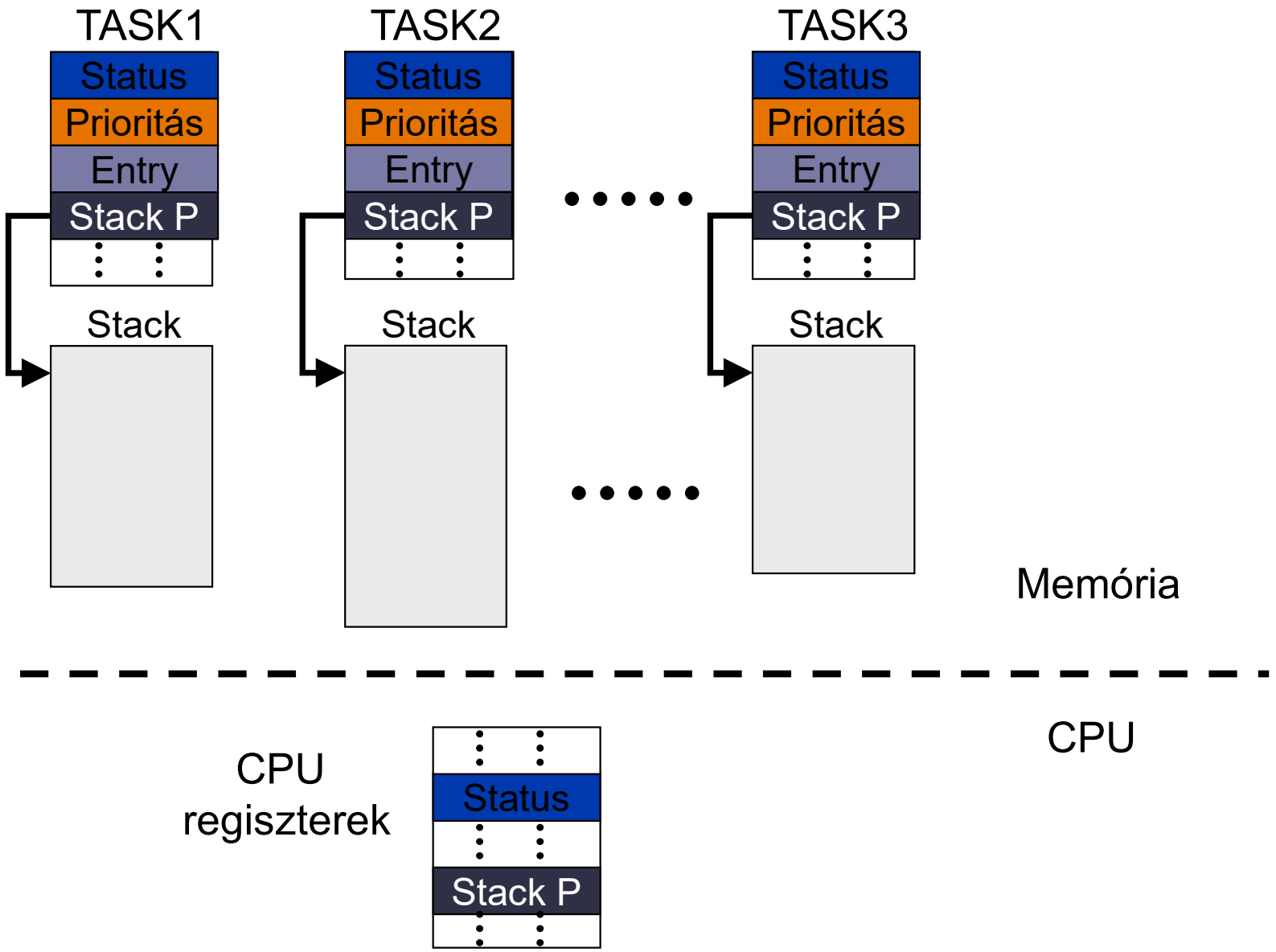
- Real Time OS, preemptív ütemezés



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

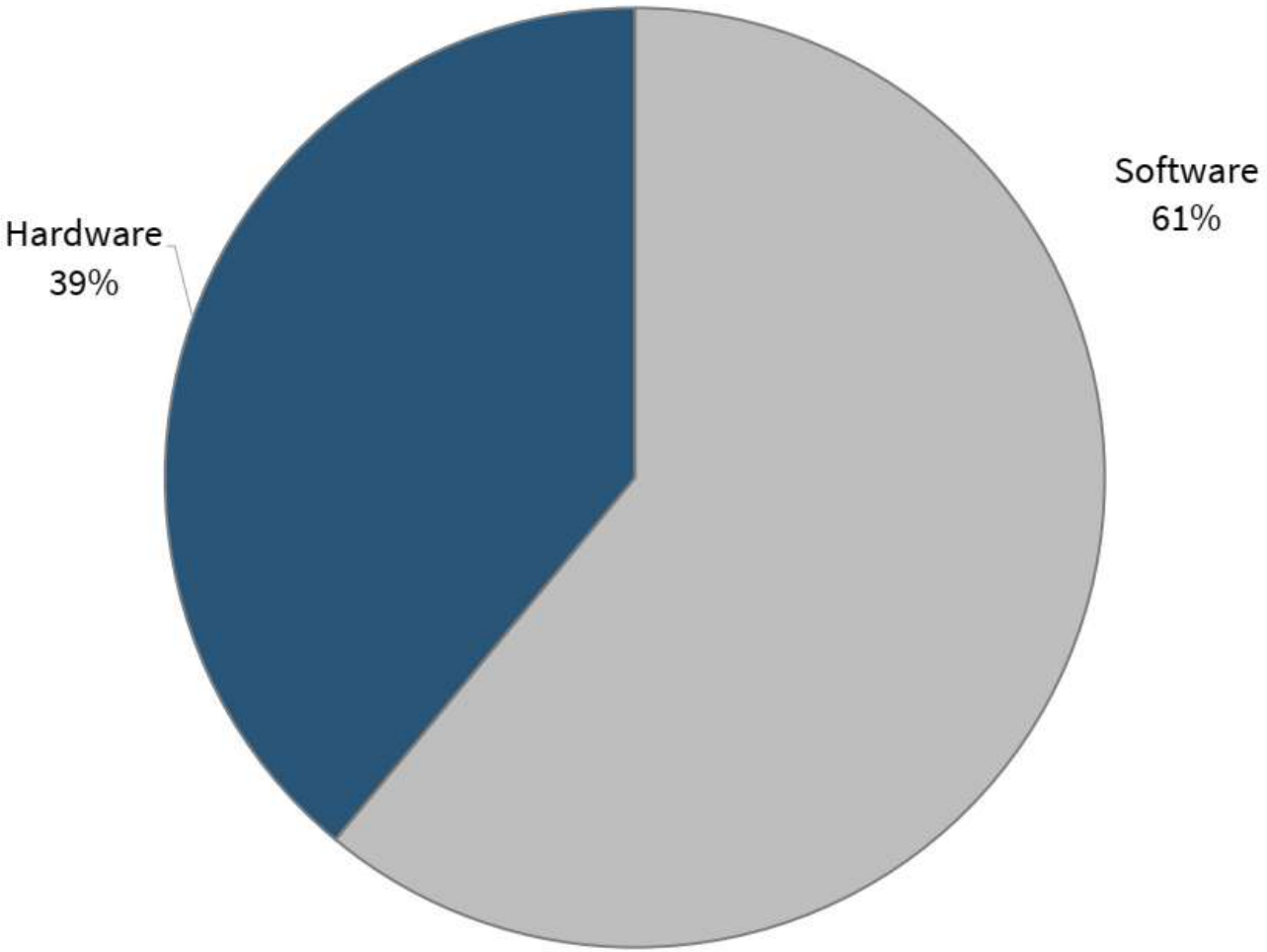
- Footprint
- Konfigurálhatóság
- Real-time viselkedés
- Nem az OS indítja az alkalmazást, hanem az alkalmazás az OS-t
- Nincs memória védelem

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

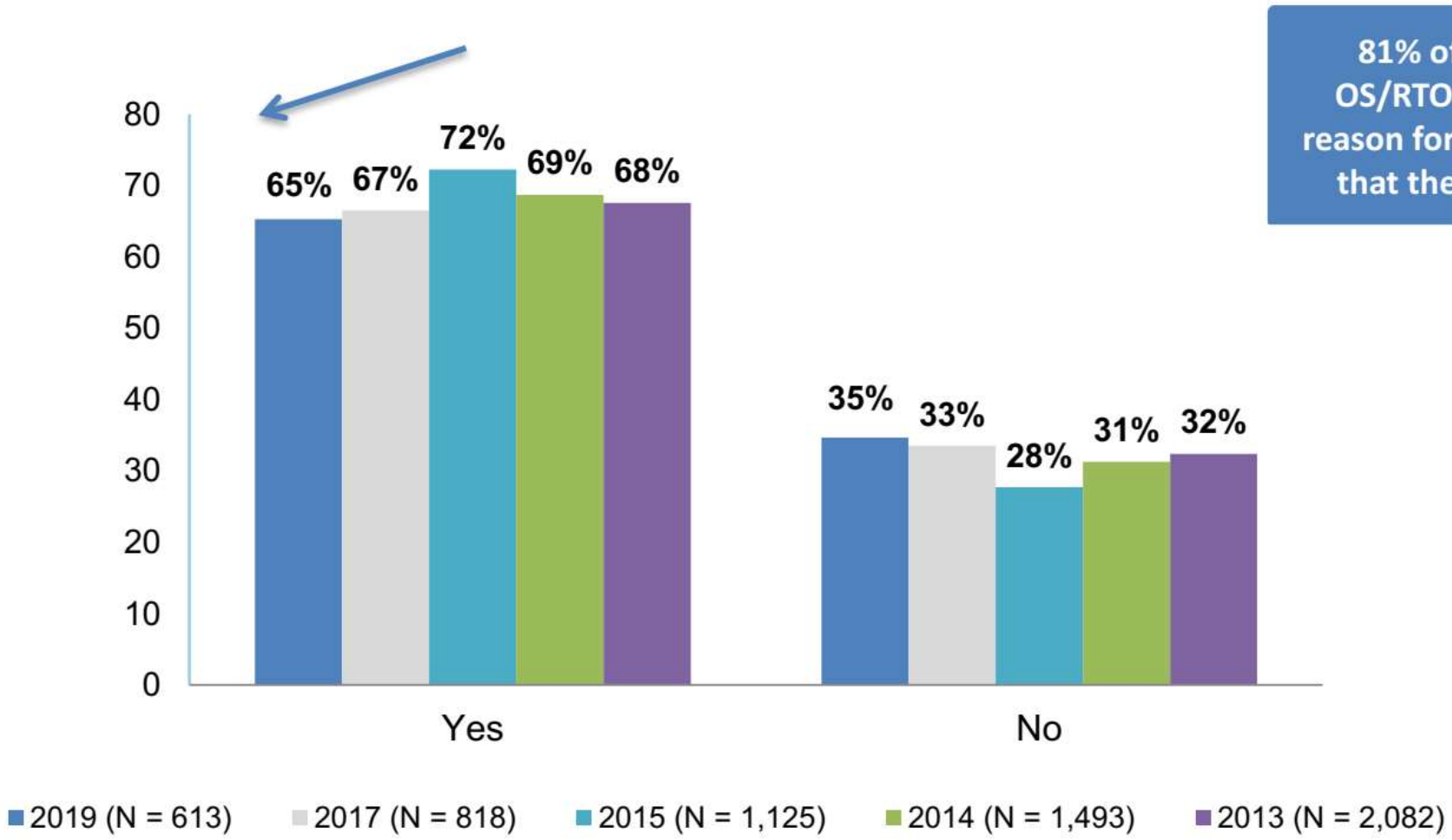
Miért fontos az operációs rendszer és egyéb szoftver tool-ok

- Project erőforrás allokáció (több mint 5 éve változatlan)

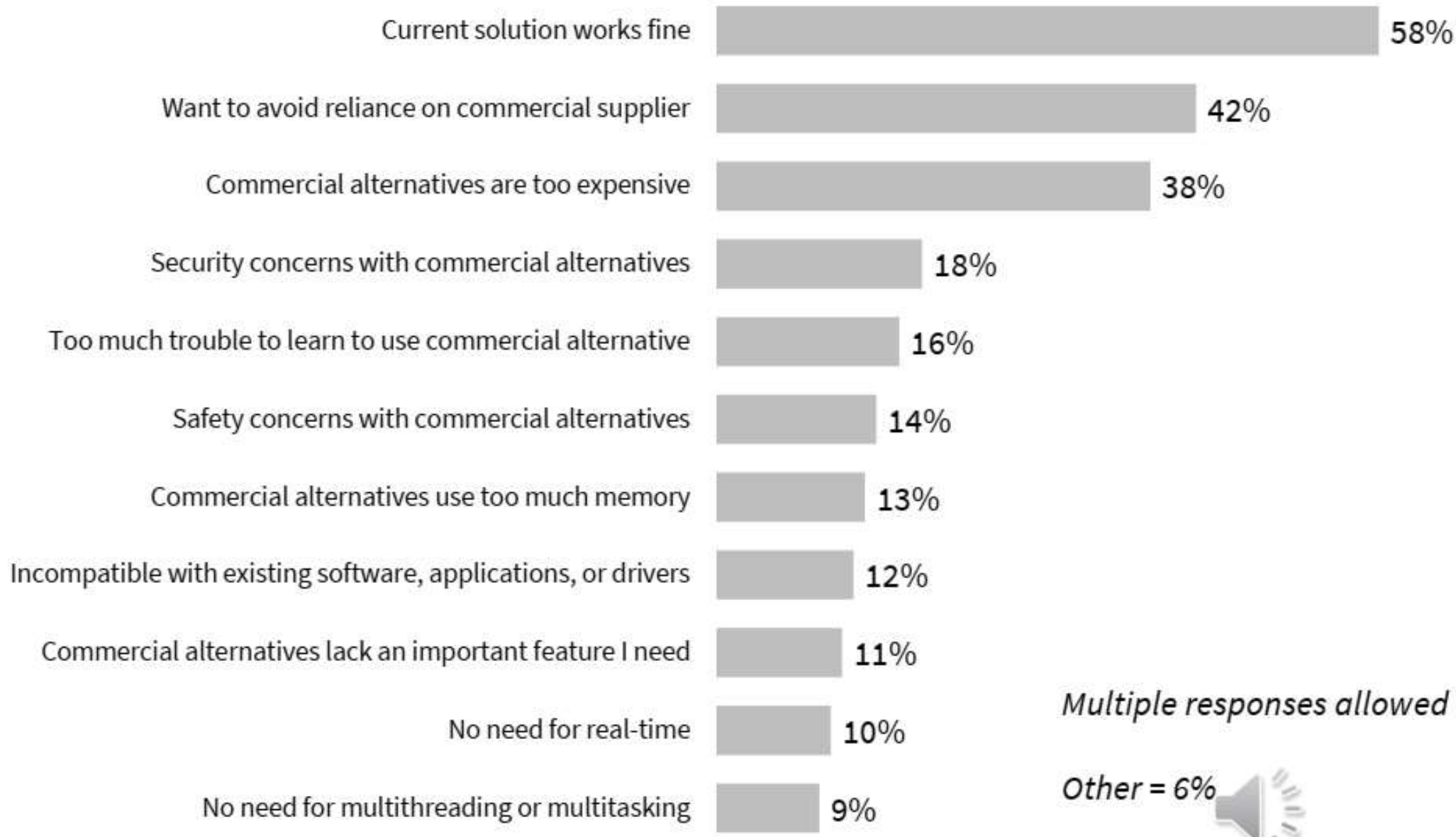


A beágyazott OS-ek elterjedtsége

- Jelen (2023-as) arány 74% használ OS-t



Miért nem használnak RTOS-t

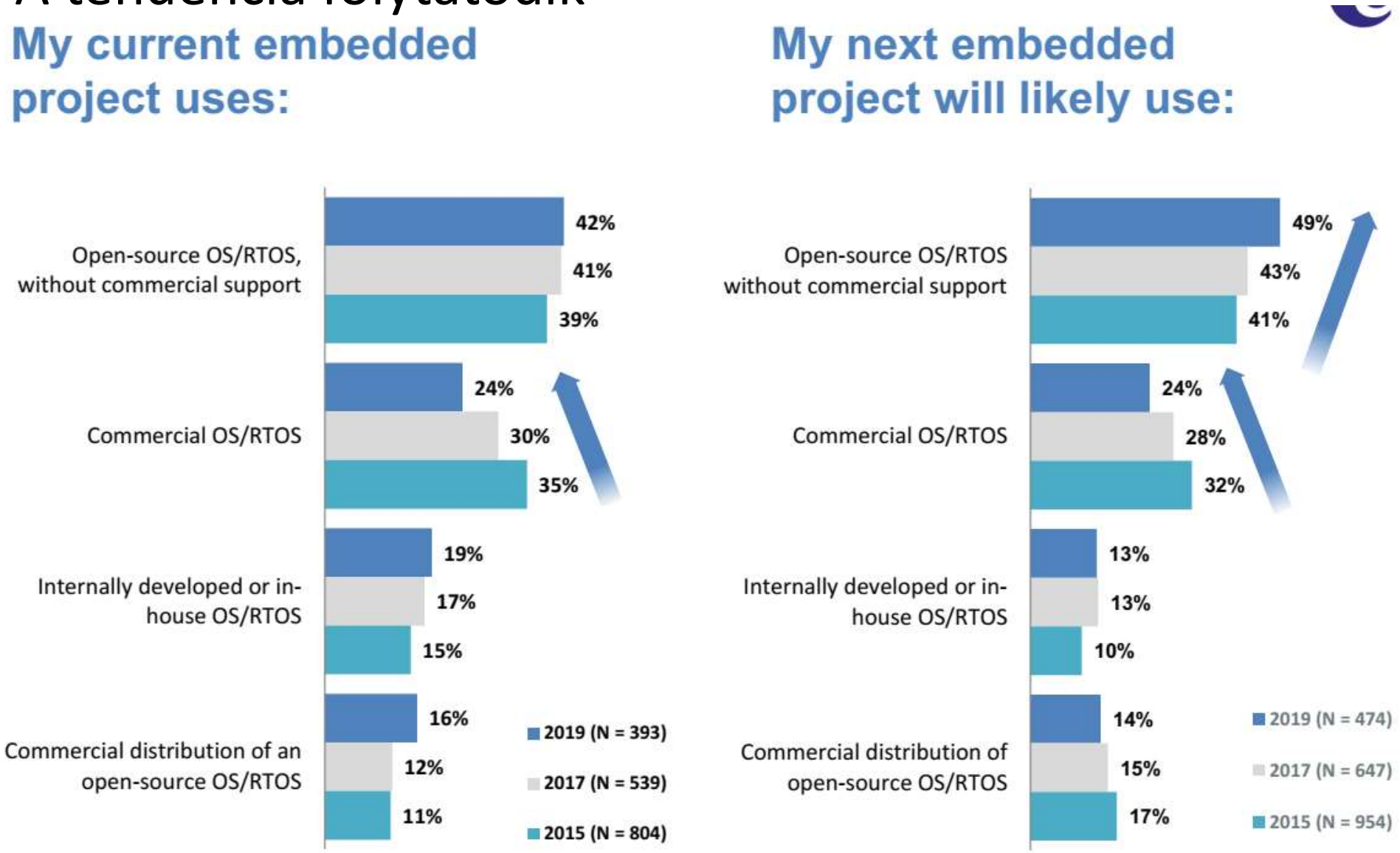


Multiple responses allowed

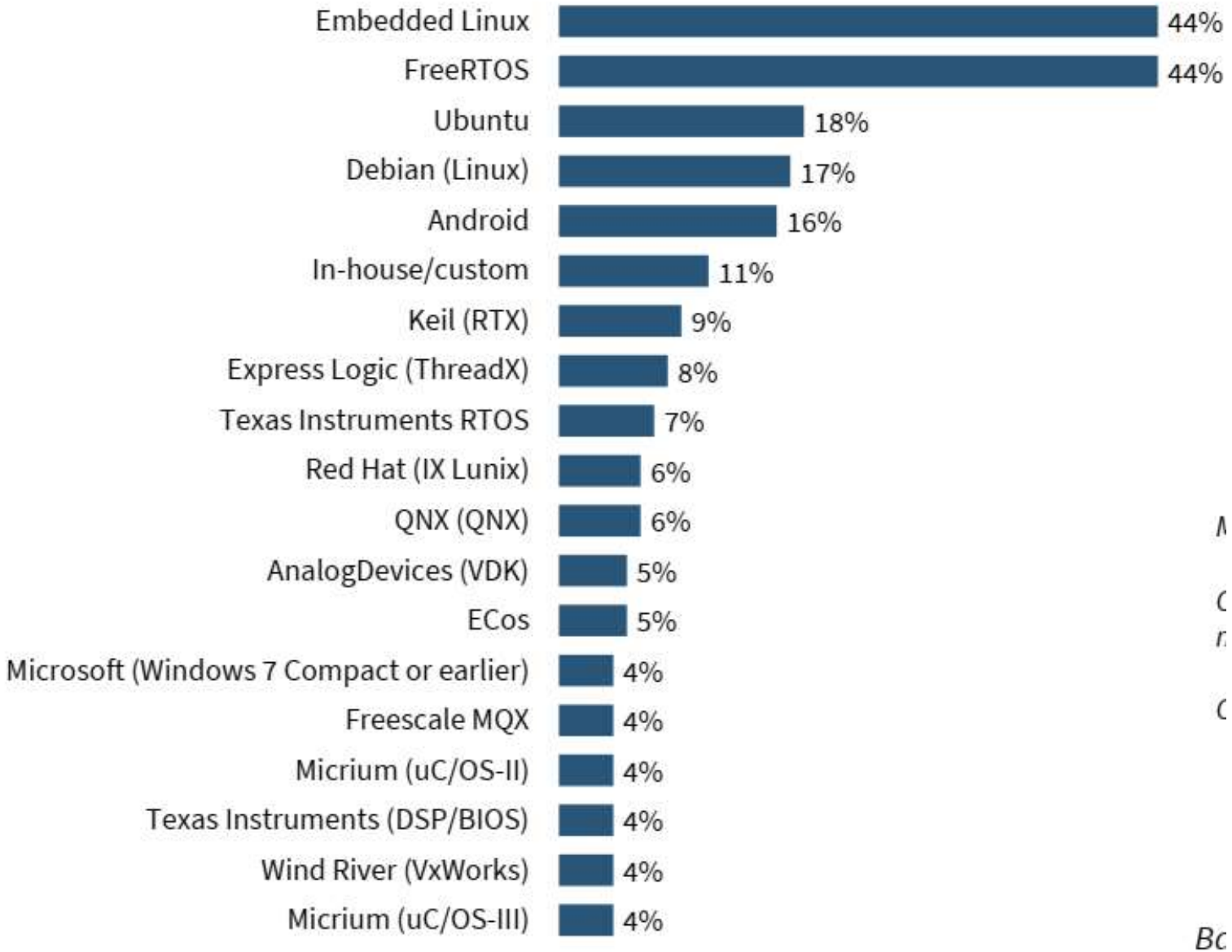
Other = 6%

Milyen OS-t használtak az elmúlt években?

- A tendencia folytatódik



Milyen OS-t használtak az elmúlt években?



µC-OS

Micrium

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

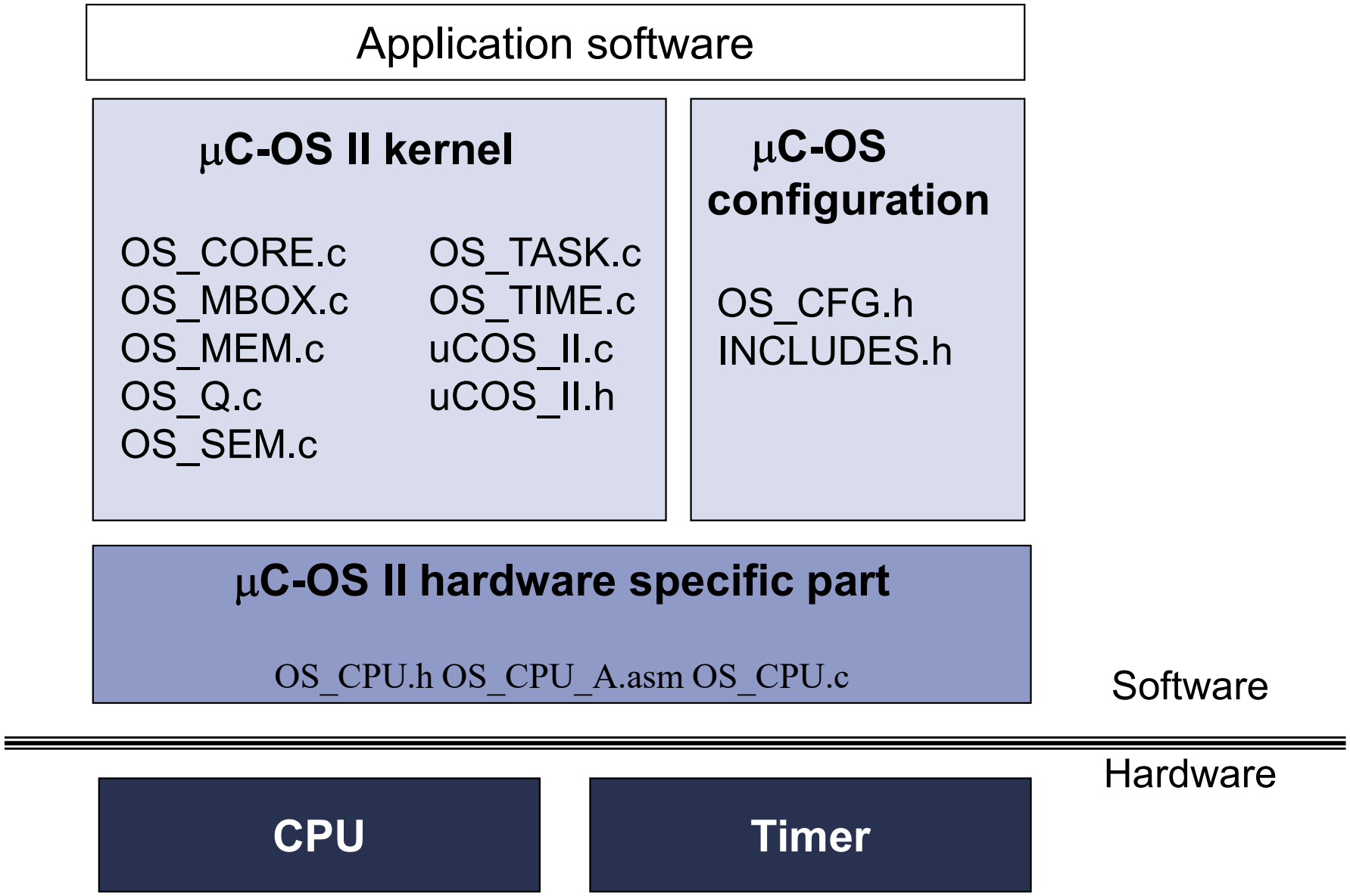
A µC/OS tulajdonságai

- 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 II architektúrája



A µC/OS konfigurálása

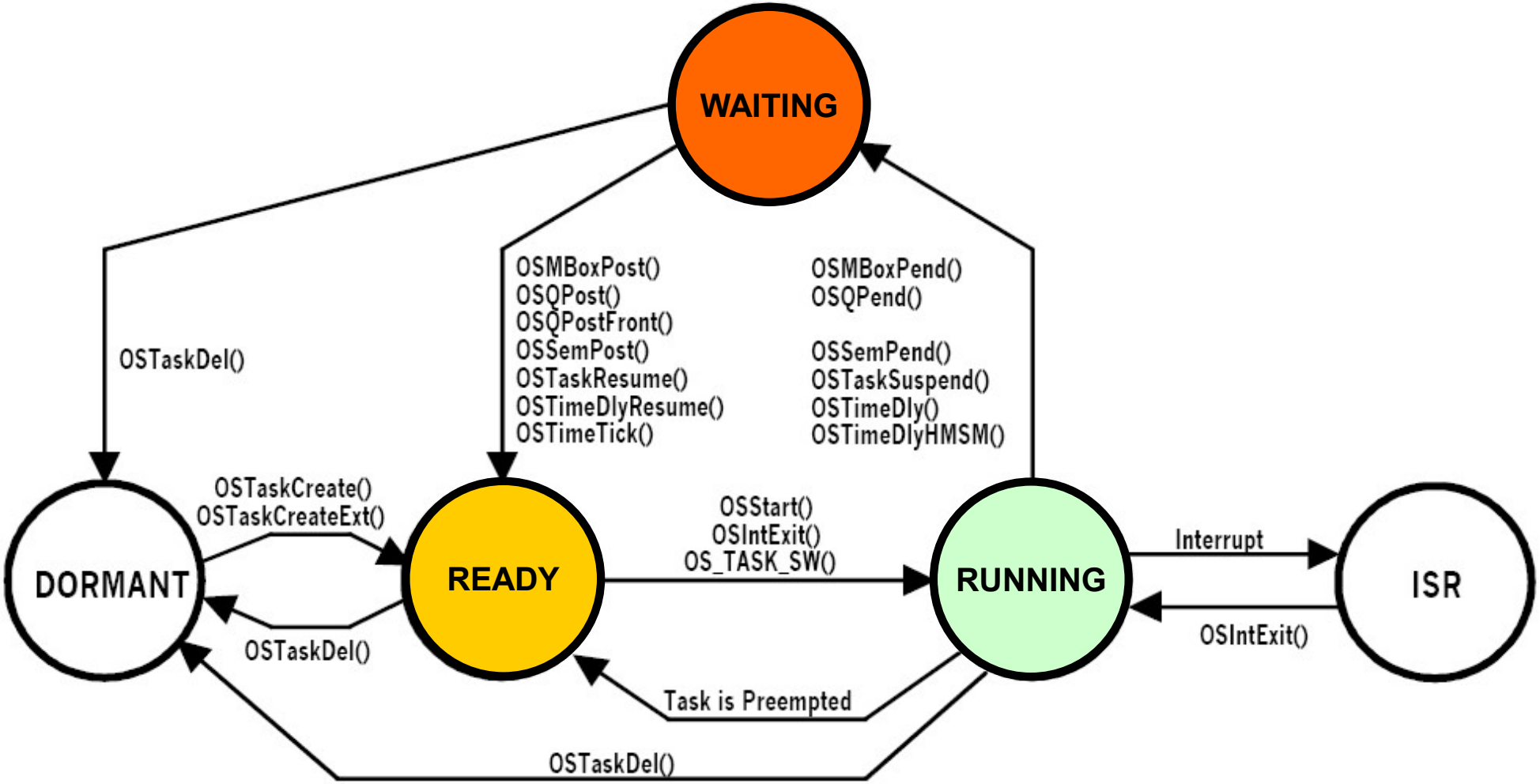
OS_CFG.h

```
/* ---------------------- 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()                         */
```

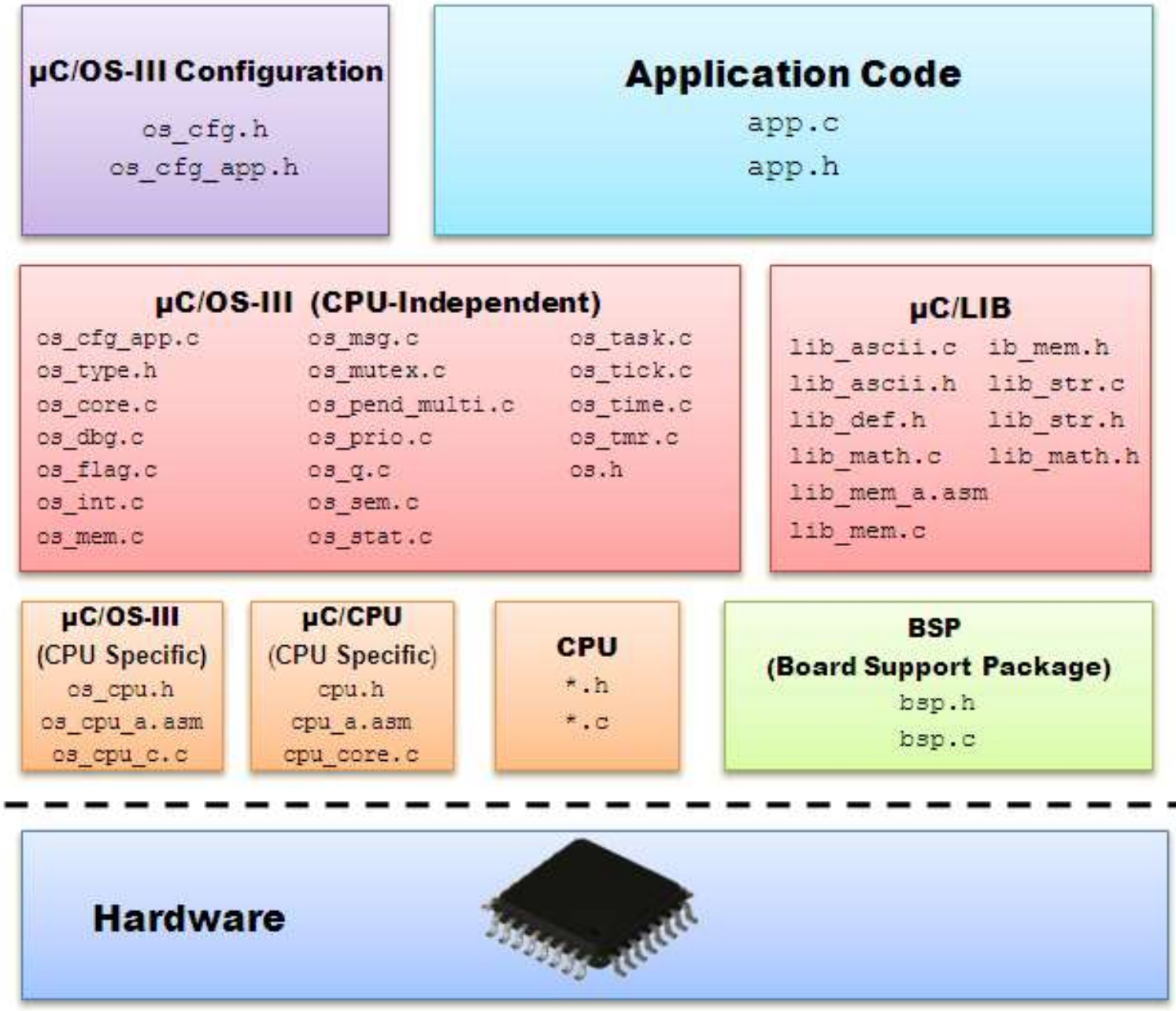
OS_MBOX.c

```
#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



A µC/OS III felépítése



FreeRTOS

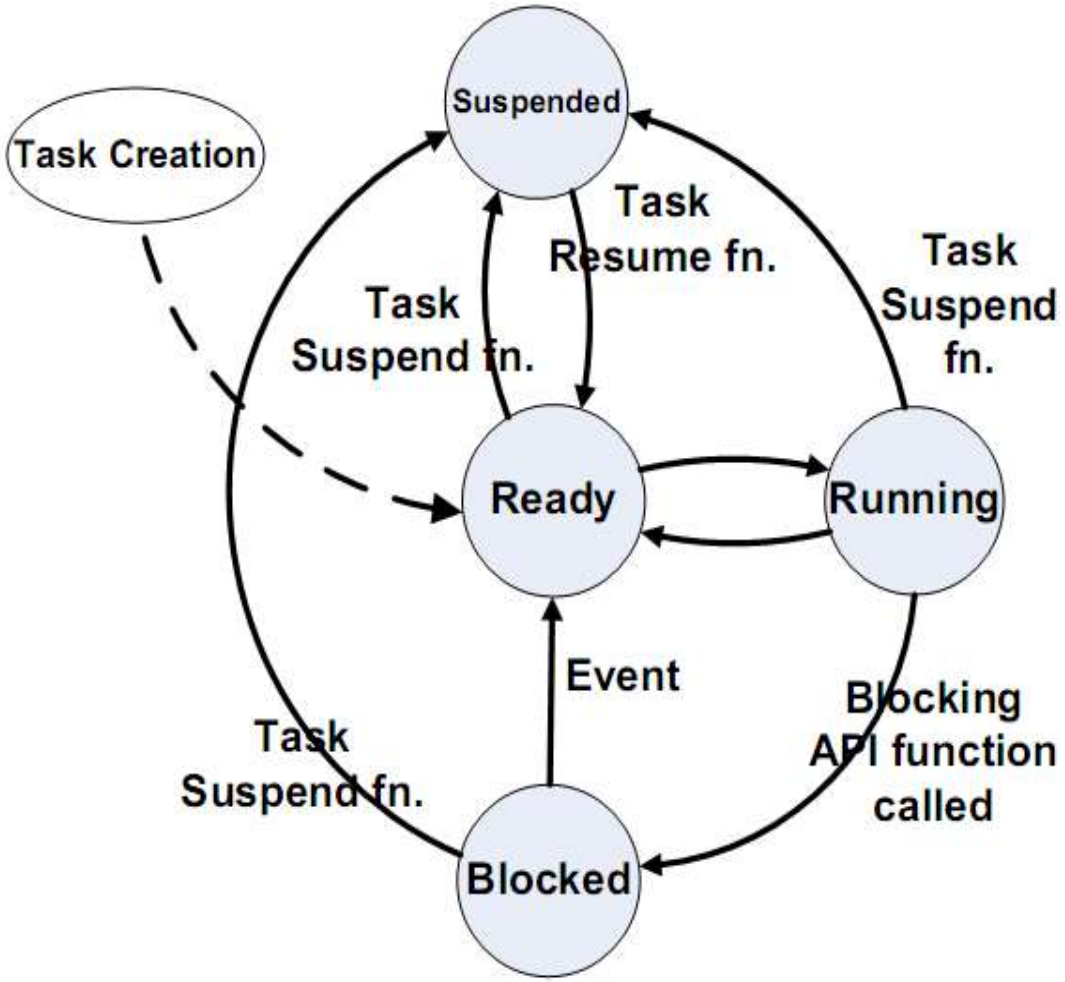
FreeRTOS

- Nyílt forráskódú egyszerű kernel
  - www.freertos.org

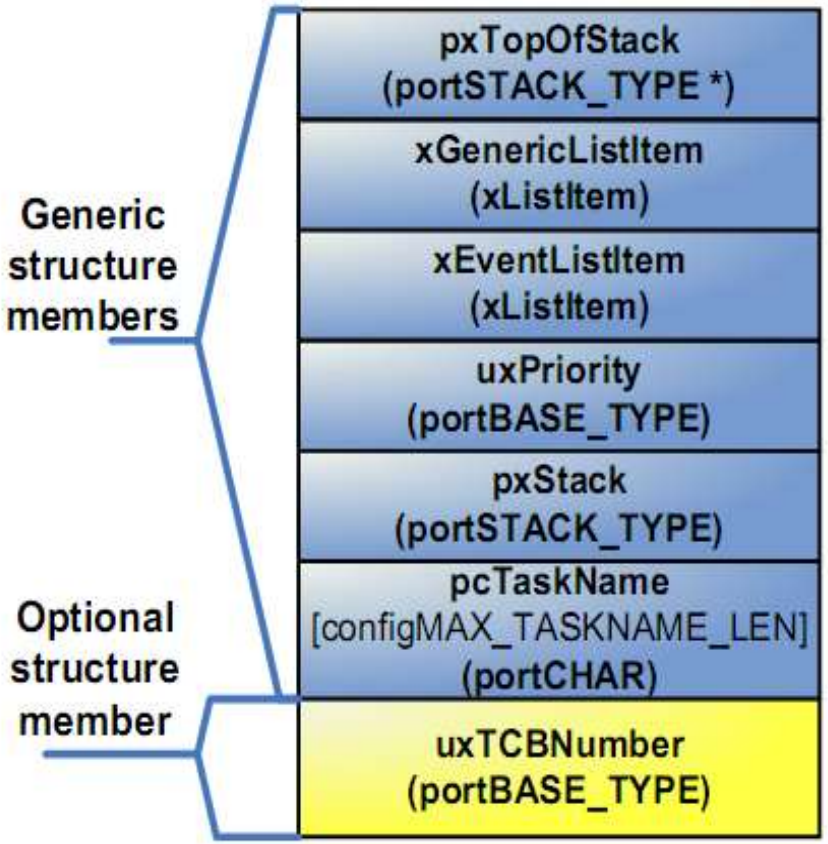
- Az elmúlt időszak legdinamikusabban fejlődő könnyű kategóriájú kernelje
- 2010 óta a legnépszerűbb RTOS
- Portok:
  - ARM7, ARM9, CortexM
  - Atmel AVR, AVR32
  - PIC18, PIC24, dsPIC, PIC32
  - Microblase…

FreeRTOS taszkok

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



FreeRTOS taszk control block

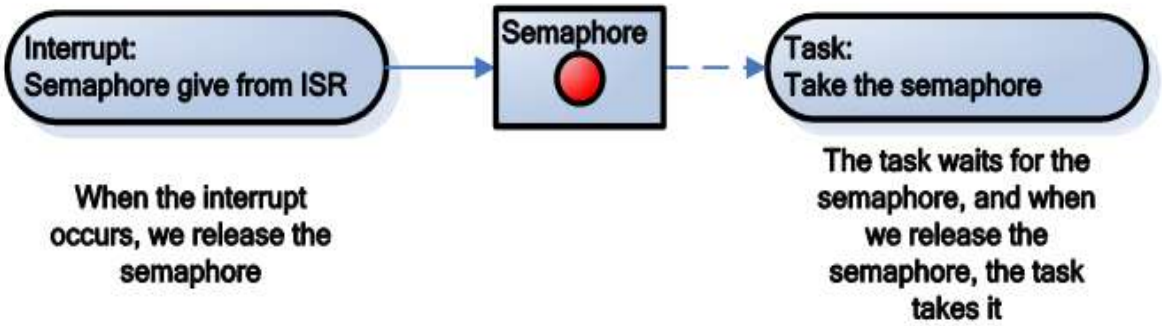


FreeRTOS taszkok kezelése

```
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 );
}
```

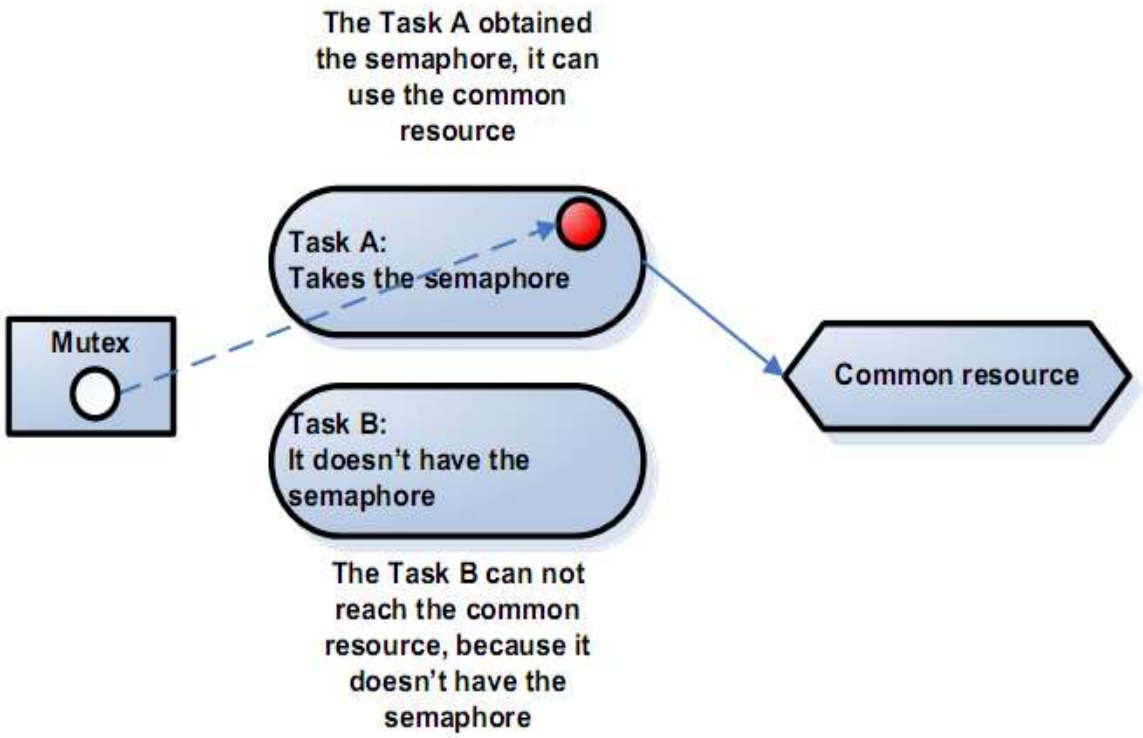
FreeRTOS taszk szinkronizáció

- 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.



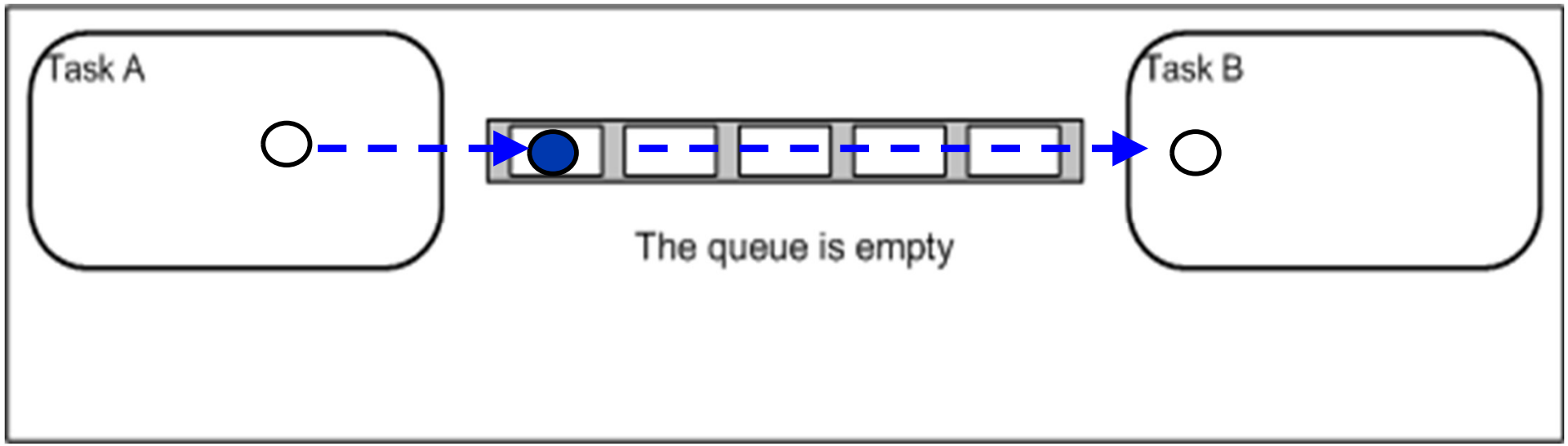
FreeRTOS taszk szinkronizáció mutex

- Mutexek
  - Prioritás inverzió ellen védettek
  - Ne használjuk megszakításból



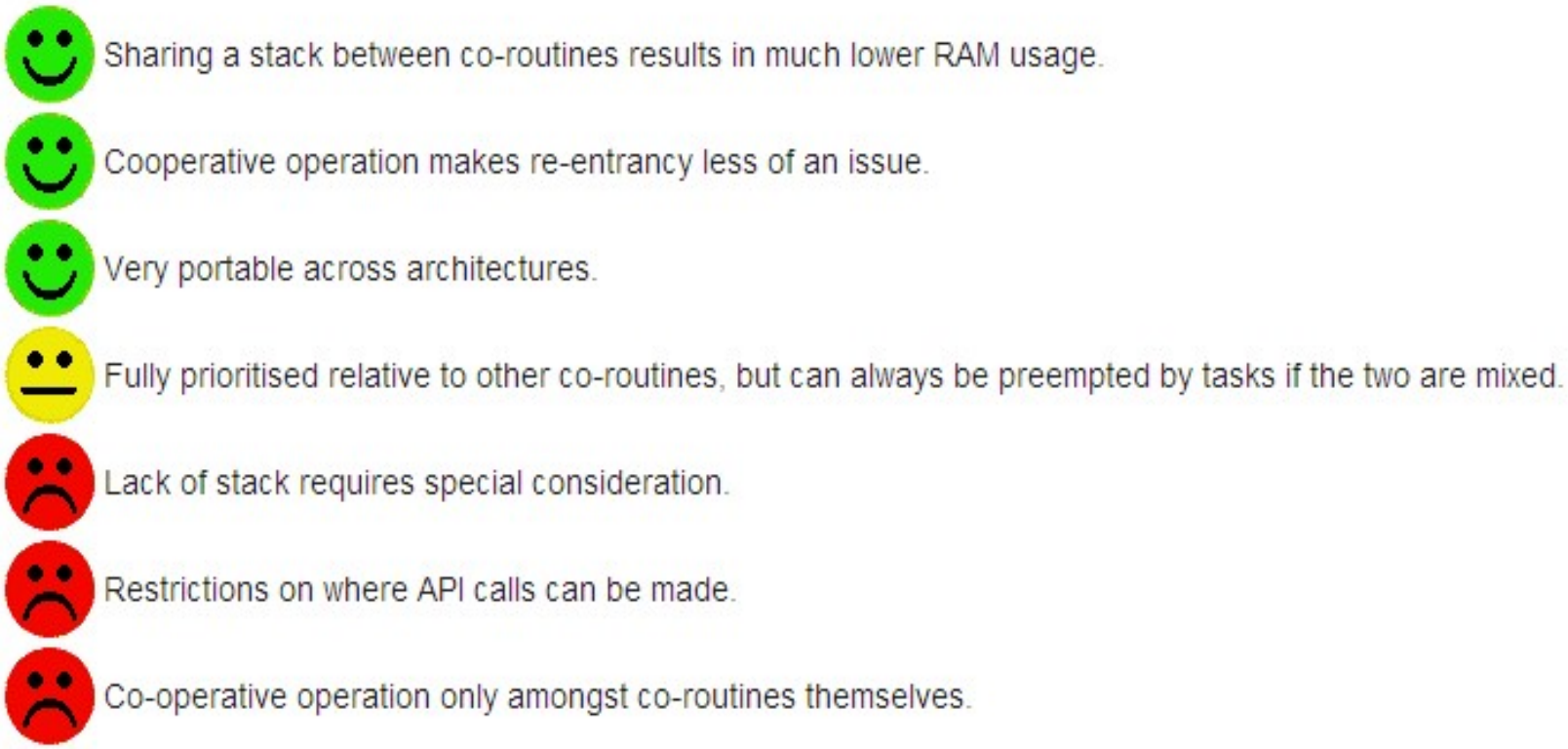
FreeRTOS Queue

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



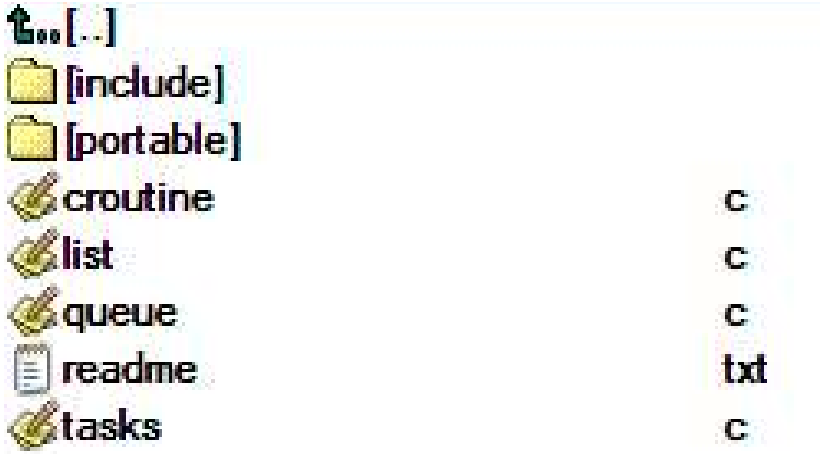
FreeRTOS CoRutin

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



FreeRTOS forráskód elrendezés

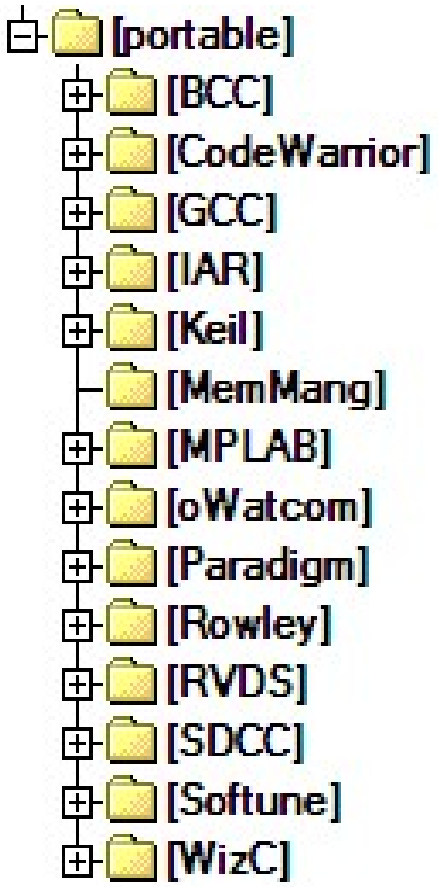
- Nagyon egyszerű alap kernel
  - tasks.c, queue.c, list.c. File-ok az Source könyvtárban



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

FreeRTOS portolás

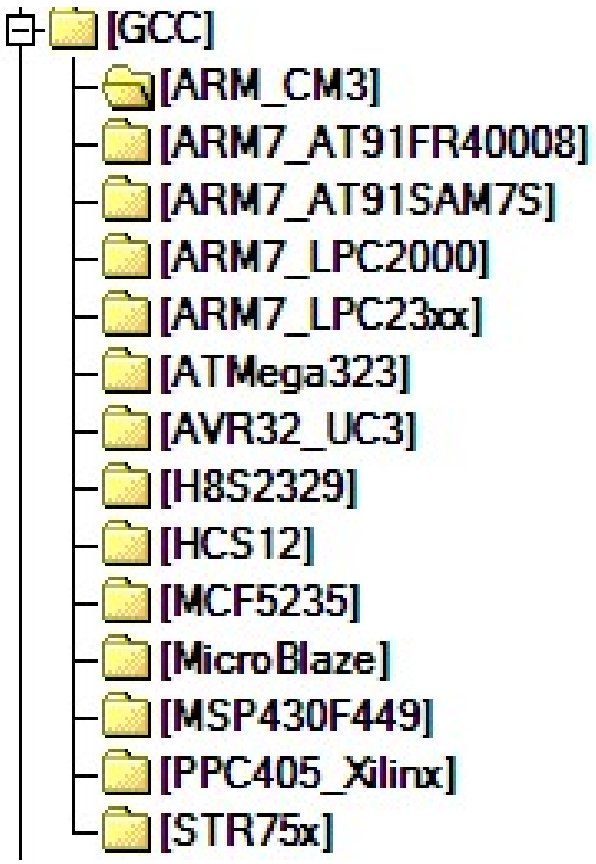
- A port specifikus kód részek a portable directoryban



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

GCC specifikus részek

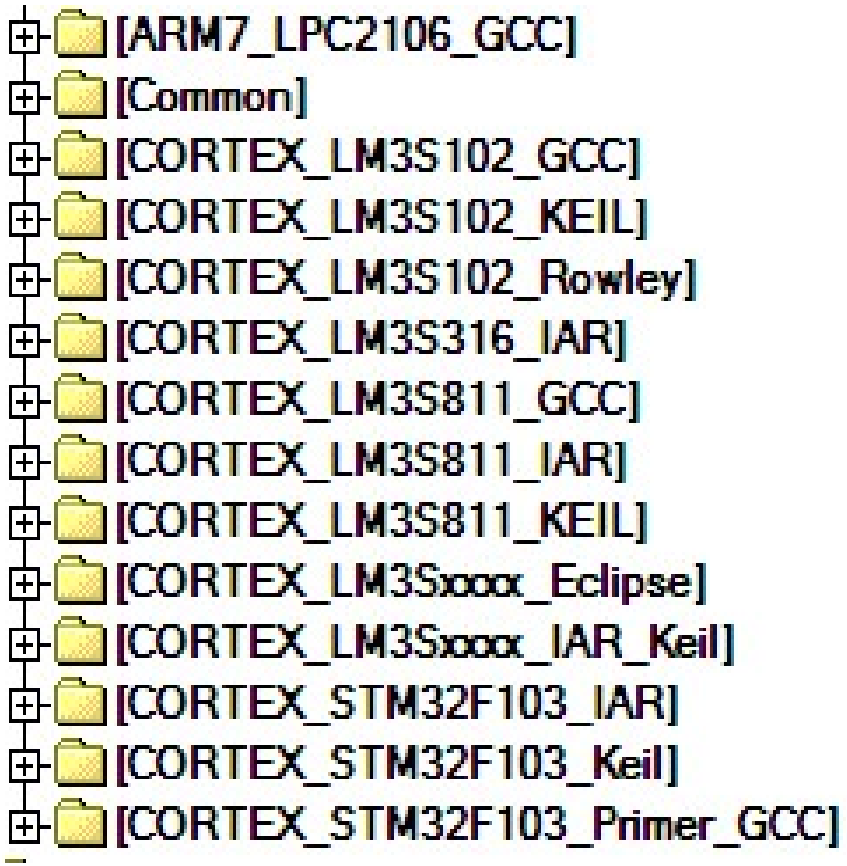
- GCC specifikus részek



- Egy port file

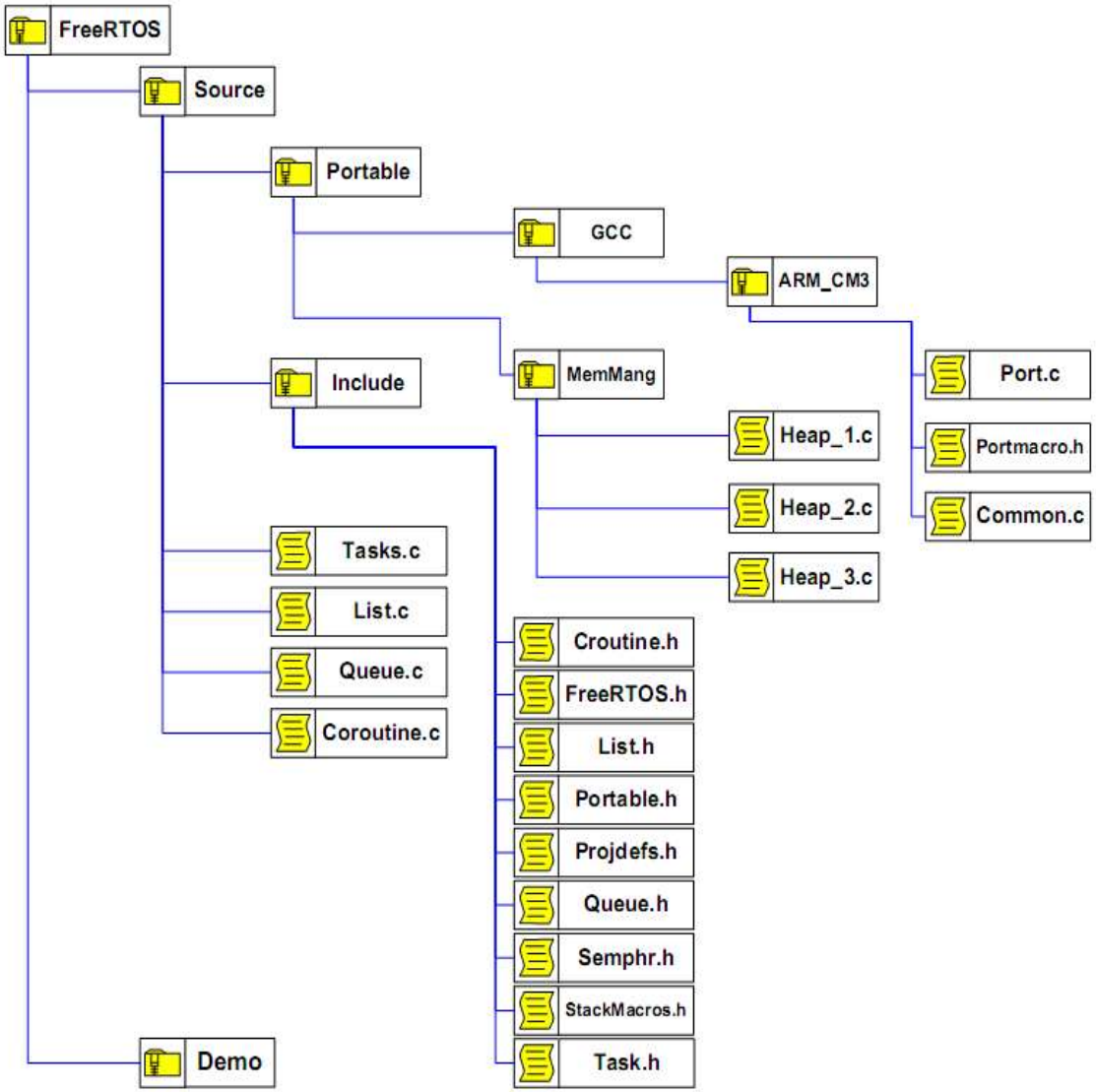
GCC demó projectek

- Kártya és fordító specifikus részek



- Startup kód
- Kártya specifikus kód

A FreeRTOS könyvtárszerkezete *áttekintés*



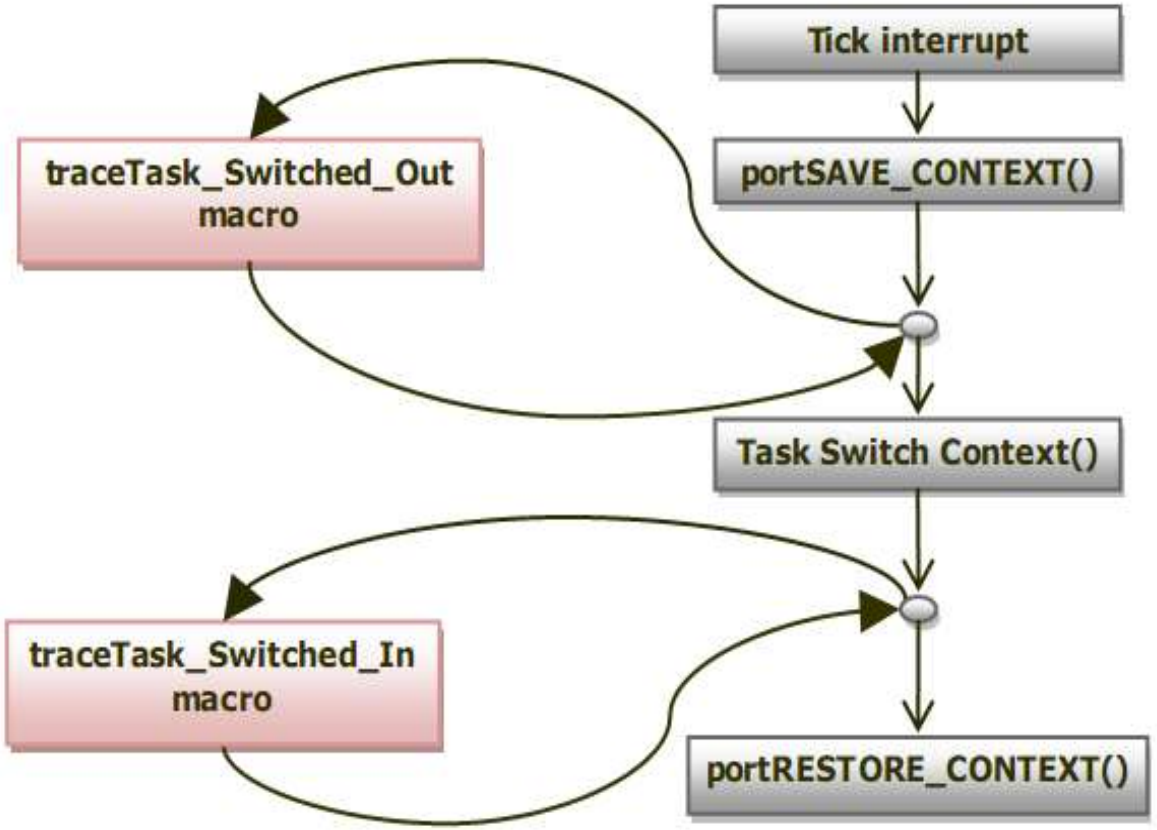
FreeRTOS konfiguráció

- 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 )
```

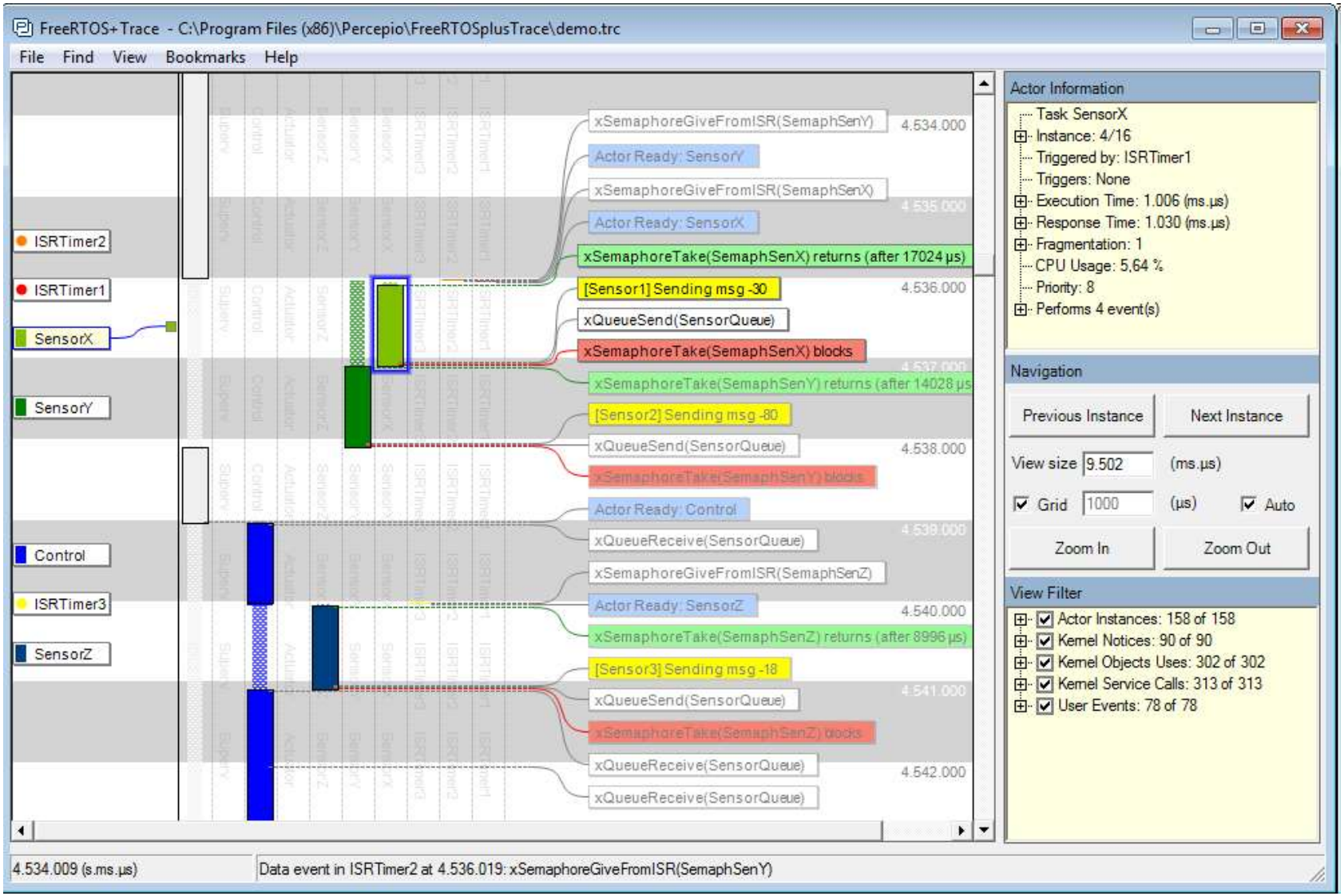
Trace hookok

- Gyakorlatilag minden fontosabb belső lépéshez tartozik



Trace alkalmazás

- Teljes szinkronizációs történet megjelenés



FreeRTOS kereskedelmi változatok

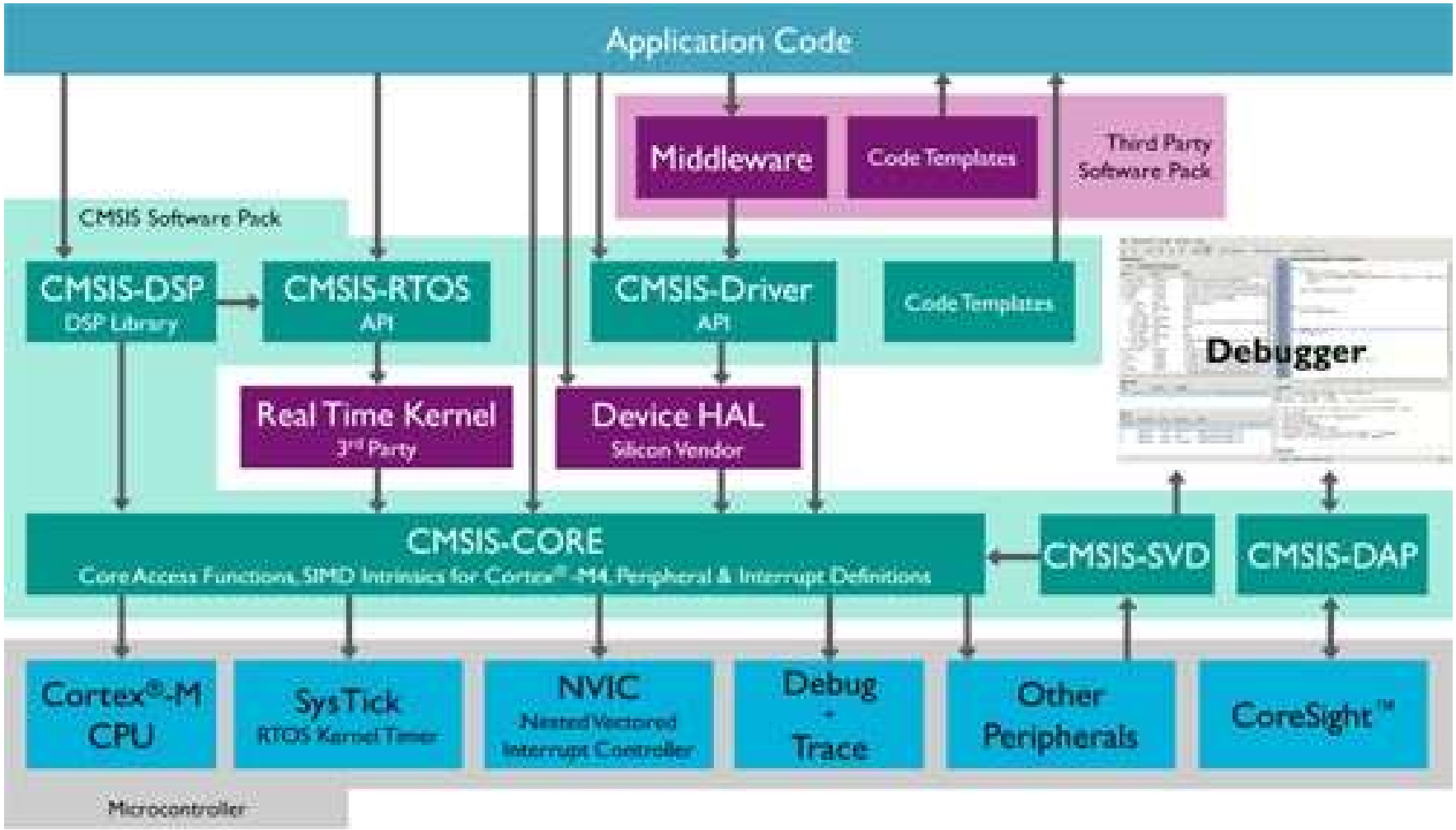
- OpenRTOS
  - Kereskedelmi szinten támogatott verzió
  - USB, File rendszer, TCP-IP támogatás
- SafeRTOS
  - SIL3 szintű tanúsítvány
  - Stellaris LM3S9B96 Microcontrollerbe ROM szinten beépítve

Mit hoz magával egy RTOS?

- Régebben általában célszerű volt 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
- Most már az RTOS inkább része a gyártók által összeállított firmware platformoknak

Újdonságok CMSIS RTOS?

- Általános felület RTOS absztrakcióra



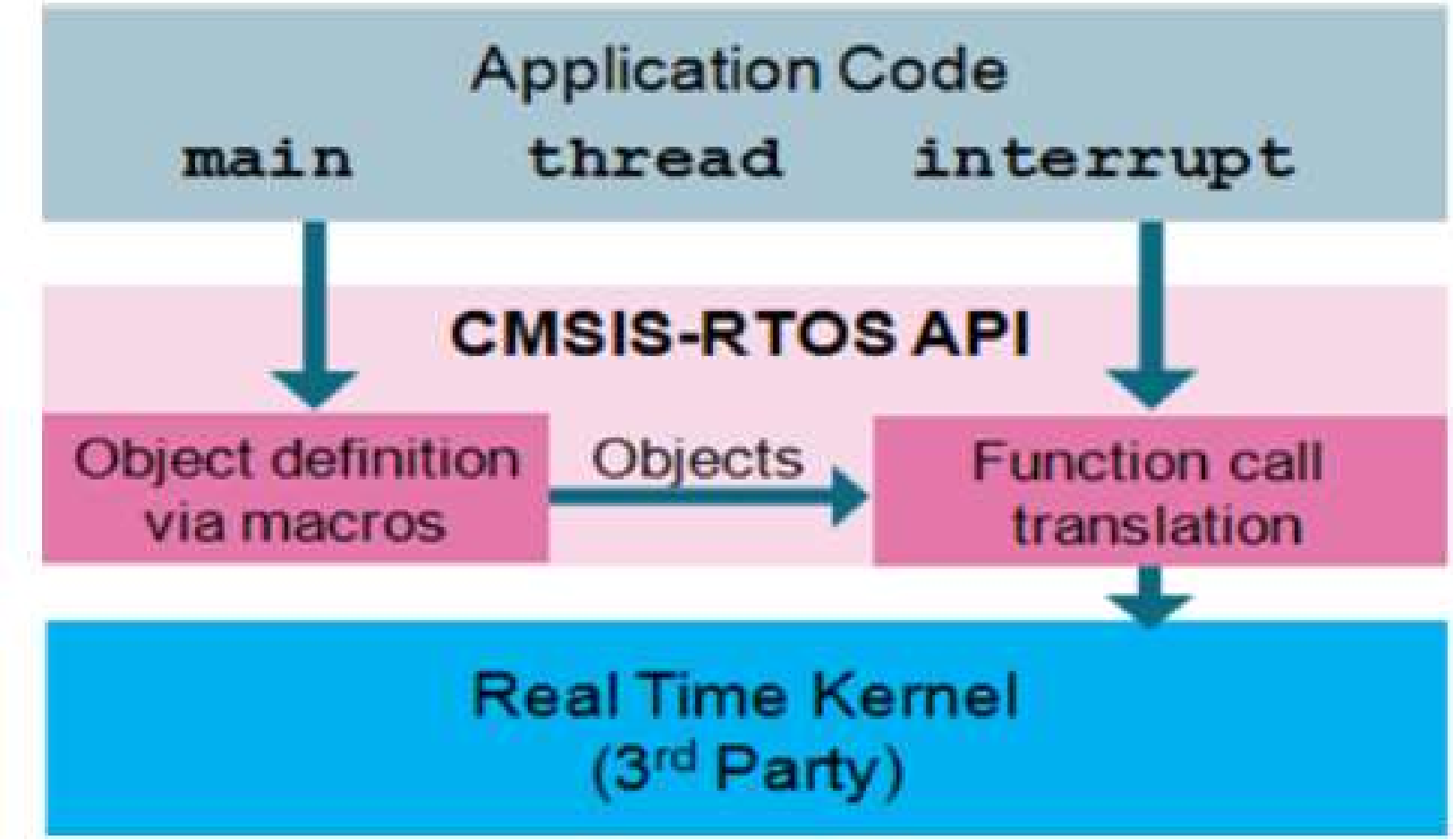
CMSIS RTOS

- Egy konform API, ami RTOS független megvalósítást ad
  - Thread kezelő függvények
  - Szinkronizációs és időzítési függvények

- Egyre több környezet támogatja
  - Keil MDK
  - STM32 Cube
  - Mbed

CMSIS RTOS

- Felépítése



CMSIS RTOS

- Kernel kezelő függvények

osStatus	**osKernelInitialize** (void) Initialize the RTOS Kernel for creating objects.
osStatus	**osKernelStart** (void) Start the RTOS Kernel.
int32_t	**osKernelRunning** (void) Check if the RTOS kernel is already started.
uint32_t	**osKernelSysTick** (void) Get the RTOS kernel system timer counter.

CMSIS RTOS

- Thread menedzsment függvények

osThreadId	**osThreadCreate** (const **osThreadDef_t** *thread_def, void *argument) Create a thread and add it to Active Threads and set it to state READY.
osThreadId	**osThreadGetId** (void) Return the thread ID of the current running thread.
osStatus	**osThreadTerminate** (**osThreadId** thread_id) Terminate execution of a thread and remove it from Active Threads.
osStatus	**osThreadSetPriority** (**osThreadId** thread_id, **osPriority** priority) Change priority of an active thread.
osPriority	**osThreadGetPriority** (**osThreadId** thread_id) Get current priority of an active thread.
osStatus	**osThreadYield** (void) Pass control to next thread that is in state **READY**.

CMSIS RTOS

- Általános időzítő függvények

osStatus **osDelay** (uint32_t millisec)
Wait for Timeout (Time Delay).

osEvent **osWait** (uint32_t millisec)
Wait for Signal, Message, Mail, or Timeout.

- Általános timer függvények

osTimerId **osTimerCreate** (const **osTimerDef_t** *timer_def, **os_timer_type** type, void *argument)
Create a timer.

osStatus **osTimerStart** (**osTimerId** timer_id, uint32_t millisec)
Start or restart a timer.

osStatus **osTimerStop** (**osTimerId** timer_id)
Stop the timer.

osStatus **osTimerDelete** (**osTimerId** timer_id)
Delete a timer that was created by **osTimerCreate**.

CMSIS RTOS

- Szálkommunikációs függvények
  - Signal events
  - Semaphores
  - Mutex
  - Message queue
  - Mail queue
  - Memory Pool