Real-time operációs rendszerek RTOS

Dr. Schuster György

2015. október 8.

FreeRTOS

A FreeRTOS egy real-time operációs rendszer beágyazott rendszerek számára. Számos processzorra portolták. A FreeRTOS terjesztése GPL alatt történik egy apró kivétellel, nevezetesen: az a kód, amelyet a felhasználó saját maga ír, az zárt kód marad, míg a kernel marad szabad.

A FreeRTOS -t kicsinek és egyszerűnek tervezték. A kód letölthető a

freertos.org

-ról.
A forrás olvasható.
A kernel 3-4 C fájlból áll összesen.

Kicsi, mert még AVR8 architektúrán is alkalmazható.

Támogatott architektúrák

ARM	ARM7, ARM9, **ARM Cortex-M3**
Atmel	AVR8, AVR32
Cortus	APS3
Fujitsu	MB91460 sorozat, MB96340
Freescale	Coldfire V1, Coldfire V2, HCS12
Intel	x86, 8052
PIC	PIC18, PIC24, dsPIC, PIC32
Renesas	78K0R, H8/S, RX600, SuperH, V850
Texas	MSP430
Xilinx	
MicroBlaze	

FreeRTOS tulajdonságai

- pre-emptive, vagy kooperatív működés
- flexibilis taszk prioritás állítás
- sorok
- bináris szemaforok
- számláló szemaforok
- rekurzív szemaforok
- mutexek
- tick hook menedzsment
- idle hook menedzsment
- nyomkövetési szolgáltatás
- stack overflow ellenőrzés

Taszk kezelés

Megjegyzés: A FreeRTOS esetén a szálakat (thread) taszknak hívjuk.

A taszkokat mint C függvényeket implementáljuk.

- típusuk **`void`**
- az átadott paraméter **`void *`**
- a taszk függvény végtelen ciklusban fut és soha ki nem lép
- ha mégis kilépne, akkor a taszkot még azelőtt törölni kell mielőtt a program kilép a függvényből

Példa

Deklaráció:

```
void ATaskFunction(void *pvParameter);
```

Definíció:

```
 void ATaskFunction(void *pvParameter)
  {
  int i=0;
  while(1)
     {
        ⋮
  a taszk törzse
        ⋮
     }
  vTaskDelete(NULL);
  }
```

Taszk állapotok

Egyprocesszoros gépen egyidőben csak egy taszk futhat.
A nem futó taszkok adatai el vannak mentve.
A taszk amikor futó állapotba kerül pontosan ott folytatja a program végrehajtását, ahol előzőleg az fel lett függesztve.
Taszk futó állapotba helyezése: switched in
Taszk futásának felfüggesztése: swapped out

A taszk futó, nem futó allapot változását csak és kizárólag az ütemező végezheti.

Taszk előállítása

```
portBASE_TYPE xTaskCreate(pdTASK_CODE pvTaskCode,
                          const signed portCHAR * const pcName,
                          unsigned portSHORT usStackDepth,
                          void *pvParameters,
                          unsigned portBASE_TYPE uxPriority,
                          xTaskHandle *pxCreateTask);
```

pvTaskCode a taszk függvény címe

pcName a taszk leíró neve, ez csak a debughoz kell

usStacDepth a taszkhoz rendelt stack méret

pvParameters a taszknak átadott paraméter

uxPriority a taszk prioritását adja meg, a legalacsonyabb a 0, a legmagasabb **configMAX_PRIORITY-1**

pxCreatedTask ez a taszk handlere

Példa

```
 void vTask1(void *pvParameters)
     {
     const char pcTaskName[]="T 1";
     while(1)
         {
        .
        .
        .
         }
     }
void vTask2(void *pvParameters)
     {
     const char pcTaskName[]="T 2";
     while(1)
         {
        .
        .
        .
         }
     }
int main(void)
     {
     xTaskCreate(vTask1,"Task 1",512,NULL,1,NULL);
     xTaskCreate(vTask2,"Task 2",512,NULL,1,NULL);
     vTaskStartScheduler();
     while(1);
     return 0;
     }
```

A taszk paraméter használata

Egyetlen taszk függvénnyel két taszkot hozunk létre.

```
 void vTask(void *pvParameters)
     {
     const char *pcTaskName;
     pcTaskName=(char *)pvParameters;
     while(1)
         {
        .
        .
        .
         }
     }
int main(void)
     {
     xTaskCreate(vTask,"Task 1",512,NULL,1,NULL);
     xTaskCreate(vTask,"Task 2",512,NULL,1,NULL);
     vTaskStartScheduler();
     while(1);
     return 0;
     }
```

Taszkok prioritása

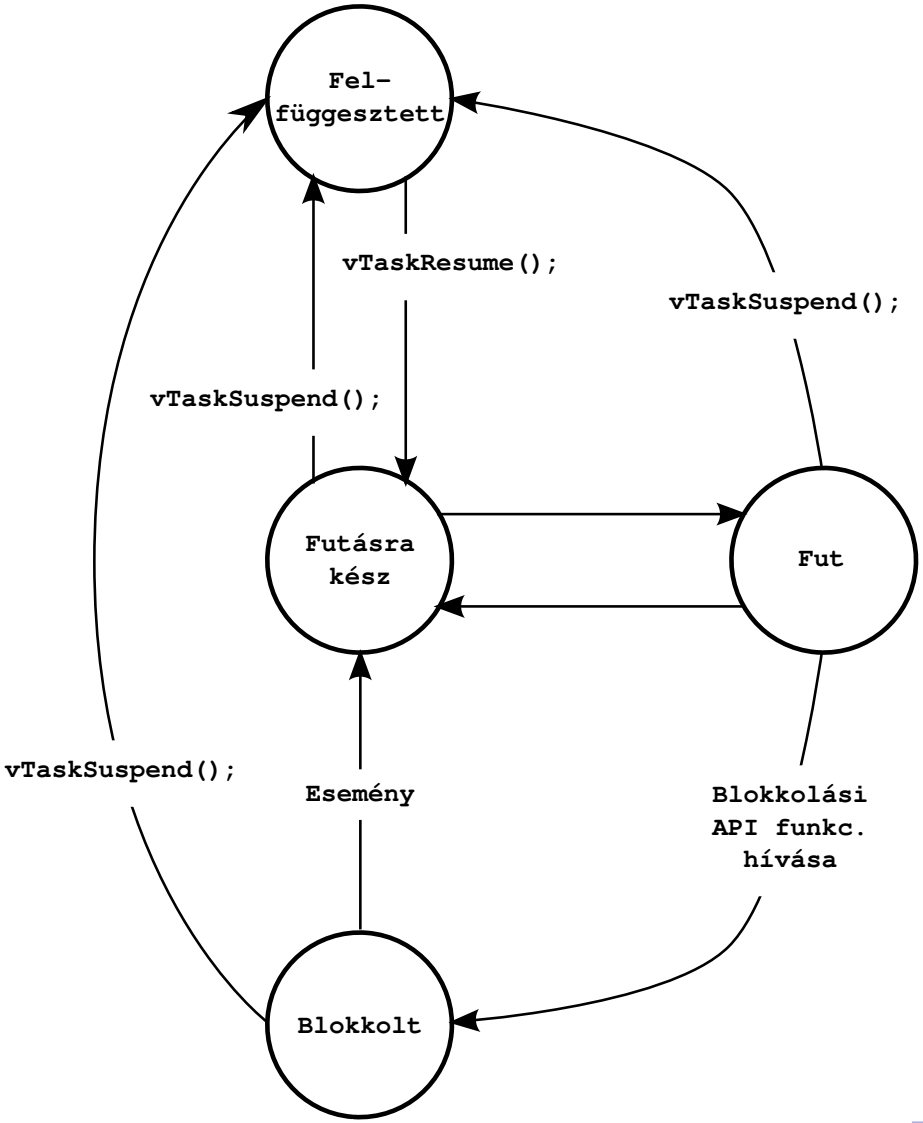
A **`uxPriority`** paraméter beállítja a taszk kezdeti prioritását.
Ez a paraméter a továbbiakban módosítható a **`vTaskPrioritySet()`** függvénnyel.
Minél alacsonyabb a prioritás érték, annál alacsonyabb a taszk prioritása.

Az ütemező folyamatosan figyeli, hogy a legmagasabb prioritású futásra képes taszk legyen kiválasztva futásra.
Ha több azonos prioritású taszk van kiválasztva, akkor ezek között egyenletesen elosztja az erőforrásokat.
Az alacsonyabb prioritású taszkok akkor kerülnek futó állapotba, ha a magasabb prioritásúak valamilyen okból nem futhatnak.

Tehát:

```
 void vTask(void *pvParameters)
     {
     const char *pcTaskName;
     pcTaskName=(char *)pvParameters;      while(1)
         {
        .
        .
        .
         }
     }
int main(void)
     {
     xTaskCreate(vTask,"Task 1",512,NULL,2,NULL); // Ez fog futni
     xTaskCreate(vTask,"Task 2",512,NULL,1,NULL); // Ez nem fog futni
     vTaskStartScheduler();
     while(1);
     return 0;
     }
```

A taszk állapotai



Blokkolt állapot

A taszk blokkolt állapotba két esemény miatt kerülhet:

1. valamilyen idővel összefüggő esemény miatt, mert egy időzítési periódus fut, vagy egy abszolut időpontra vár a rendszer. Például a taszk 10 másodpercet vár.
2. a taszk valamilyen szinkronizációs eseményre vár valamely másik taszktól, vagy egy megszakítási rutintól.

Felfüggesztett állapot

Az a taszk, amely felfüggesztett állapotban van az ütemező számára nem elérhető.

Az egyetlen mód, hogy a taszk felfüggesztett legyen a **`vTaskSuspend`** függvény hívása.

Visszakerülés ebből az állapotból a **`vTaskResume()`**, vagy a **`vTaskResumeFromISR()`** függvényekkel lehetséges.

Megjegyzés: a legtöbb alkalmazás nem használja a felfüggesztett állapotot.

Futó és futásra kész állapot

A futó állapotban a taszk[1] használja a processzort.

A futásra kész állapotban lévő taszk se nem blokkolt, se nem felfüggésztett, viszont nem ”rendelkezik fizikai processzorral”.

[1] Némelyik szakíró a taszkot virtuális processornak hívja, a futó állapot ekkor azt jelenti, hogy a virtuális processzor fizikai processzoron fut.

TCB (Task Control Block)

A taszk kezeléséhez az információkat a FreeRTOS a TCB-ben tárolja. Szerkezete:

Top of stack	A stack aktuális állapota
Task state	A taszk állapotára mutató lista elem blokkolt, vagy futásra kész
Event list	Lista elem a TCB-t az esemény listába teszi
Priority	A taszk prioritása
Stack start	Pointer a stack kezdetére
TCB number	Debug és trace mező
Task name	A taszk neve
Stack Depth	A taszk maximális mérete

Tegyük fel, hogy az előző példában szereplő példa taszk függvénye egy végtelen ciklussal időzíti a működését.

```
void vTask(void *pvParameters)
    {
    const char *pcTaskName;
    volatile unsigned long i;
    pcTaskName=(char *)pvParameters;
    while(1)
        {
                :
                :
        for(i=0;i<100000;i++);
        }
    }
```

A prioritás miatt a **`Task 1`** fog futni, mert az ő prioritása magasabb.

```
            :
            :
 xTaskCreate(vTask,"Task 1",512,NULL,2,NULL); // Ez fog futni
 xTaskCreate(vTask,"Task 2",512,NULL,1,NULL); // Ez nem fog futni
            :
            :
```

Holott lehet, hogy csak várakozik. Van-e megoldás?

Persze, hogy van.

Időzítő és várakozó függvények

Adott ideig tartó időzítés:

`vTaskDelay(portTickType);`

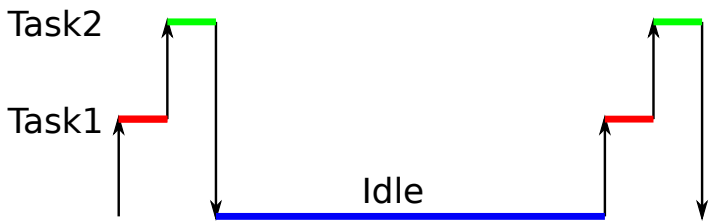
Célszerű használat: pl.: 100ms időzítés:

`vTaskDelay(100/portTick_RATE_MS);`

Cseréljük ki a taszk ciklusának a végét:

```
void vTask(void *pvParameters)
    {
    const char *pcTaskName;
    volatile unsigned long i;
    pcTaskName=(char *)pvParameters;
    while(1)
        {
                :
                :
        vTaskDelay(100/portTick_RATE_MS);
        }
    }
```

A két taszk prioritása eltérő lehet.
Ekkor a magasabb prioritású nem ”tolja ki” az alacsonyabbat, mert blokkolt állapotba kerül időnként mindkettő.



Időzítő és várakozó függvények

Magyarázat: minden egyes un. tick interrupt növel egy változót. Ez az un. **`TickCounter`**. Ennek a segítségével időzít a FreeRTOS .

```
void vTaskDelayUntil(portTickType * pxPreviousWakeTime, portTickType xTimeIncrement);
```

`pxPreviousWakeTime` megadja a számláló azon értékét, ahol a taszk felébredt

`xTimeIncrement` az előző értékhez viszonyítva megadja a késleltetés mértékét

Mielőtt először használnánk ezt a függvényt a **`TickCounter`** értékét le kell olvasnunk. Ez a

```
portTickType xTaskGetTickCount(void);
```

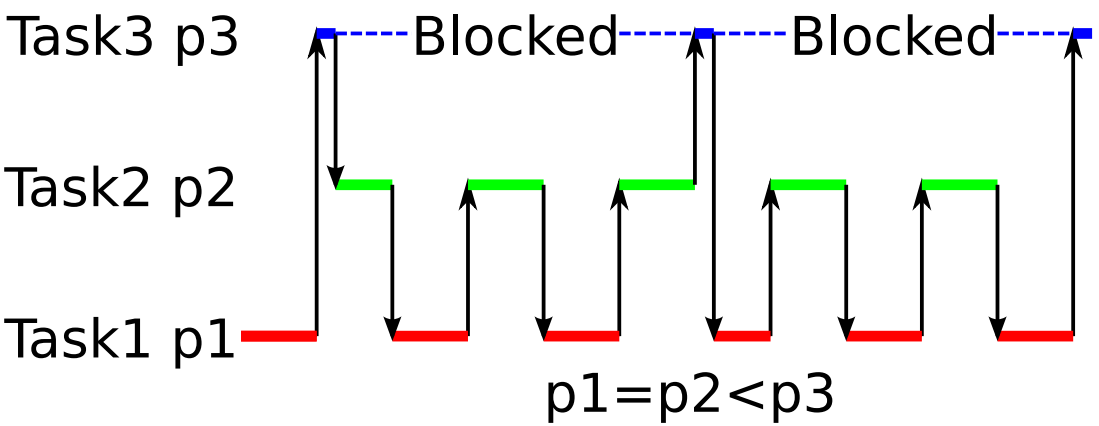
függvénnyel tehetjük meg.

Mi a különbség?

Az, hogy a **`vTaskDelay`** a meghívás pillanatától számítva időzít. Kb +1 tick-nyi bizonytalansággal. A **`vTaskDelayUntil`** mindig a belépési időponttól számítva időzít pontosan.

Esettanulmány: blokkolt és nem blokkolt taszkok együttes használata

Legyen három taszkunk.
Ebből kettő fusson azonos proritással folyamatosan.
A harmadik magasabb prioritással blokkolt időzítéssel.
Hogyan néz ki a futásuk?



p1, p2, p3 a taszkok prioritásai.

Látható, hogy a magasabb prioritású taszk engedi futni a folyamatosan futó taszkokat, amíg ő blokkolt állapotban van.

Az üres (idle) taszk

Ha futás közben az összes taszk blokkolt állapotban van a processzornak akkor is kell valamit csinálnia.

Ekkor az un. üres (idle) taszkot futtatja. Ezt a taszkot az ütemező a **`vTaskStartScheduler()`** függvény hívásakor automatikusan előállítja.

Az üres taszknak van a legalacsonyabb prioritása. Ezzel biztosítja azt, ha egy magasabb prioritású taszk belép ez azonnal ki lesz söpörve a fizikai processzorról (swapped out).

Létezik az a lehetőség, hogy az üres taszkot ”magunkévá tegük”.

Az üres taszk átdefinálása

Az üres taszk átdefiniálásához a írnunk kell egy függvény. A deklarációja:

```
void vApplicationIdleHook(void);
```

Ezután egyszerűen megírjuk.
Vegyük figyelembe:

- a prioritása a legalacsonyabb
- csak akkor fut, ha más nem
- egyszerű alkalmazások futtathatók csak (pl. a kapcsold a procit low power módba)

A következő szabályokat kell betartani:

- a taszk nem mehet se blokkolt, se felfüggesztett módba
- ha egy taszk kilép az üres taszknak kellene takarítani, ha ezt mi csináljuk valószínűleg ez nem történik meg

A **`FreeRTOSConfig.h`**-ban a **`configUSE_IDLE_HOOK`** **`#define`**-t egyre kell állítani.

Taszk prioritásának megváltoztatása

Van lehetőség az ütemező indítása után a taszk prioritásának megváltoztatására.

```
void vTaskPrioritySet(xTaskHandle pxTask,unsigned portBASE_TYPE uxNewPr);
```

pxTask a taszk leíró az **xTaskCreate** függvény utolsó paramétere

uxNewPr a kívánt prioritás

Egy taszk prioritási szintje a

```
unsigned portBASE_TYPE uxTaskPriorityGet( xTaskHandle pxTask );
```

függvénnyel lekérdezhető.

Egy taszk törlése

Egy taszk törölheti saját magát és más taszkokat a következő függvénnyel:

```
void vTaskDelete( xTaskHandle pxTaskToDelete );
```

Ha a handler helyett egy NULL pointert adunk meg, akkor a taszk saját magát törli.

Példa

```
int main(void)
  {
  xTaskCreate(vTask1,"Task 1",500,NULL,1,NULL);
  vTaskStartScheduler();
  while(1);
  }

void vTask1(void *pvParameters)
  {
  const portTickType xD100ms=10/portTICK_RATE_MS;
  while(1)
    {
    xTaskCreate(vTask2,
      "Task 2",500,NULL,2,&xT2Hnd);
    vTaskDelay(xD100ms);
    }
  }

void vTask2(void *pvParameters)
  {
  vTaskDelete(xT2Hnd);
  }
```