

BEÁGYAZOTT RENDSZEREK (C)

4. forduló



BOSCH

A kategória támogatója: Robert Bosch Kft.

Ismertető a feladathoz

A 4. forduló után elérhetőek lesznek a helyezések %-os formában: azaz kiderül, hogy a kategóriában a versenyzők TOP 20% - 40% -60% -ához tartozol-e!

Szeretnénk rá felhívni figyelmedet, hogy a játék nem Forma-1-es verseny! Ha a gyorsaságod miatt kilököd a rendesen haladó versenyzőket, kizárást vonhat maga után!

4.forduló

Gipsz Szabolcs már látja, hogy az STM32 vezérlőnek rengeteg dolga lesz: kezelnie kell a szenzorokból jövő adatokat, működtetni kell az aktuátorokat, szabályozási feladatokat kell ellátnia miközben a fő számítógéppel is együtt kell működnie. Szabolcs szeretné az egyes feladatokat megvalósító kódrészleteket egymástól jól elszeparáltan implementálni úgy, hogy a nem kívánatos egymásra hatásokat minimalizálja a stabil működés elérése érdekében.

A megoldásban a következő adatlapok lesznek a segítségedre:

STM32F103RB adatlap: <https://www.st.com/resource/en/datasheet/stm32f103rb.pdf>

NUCLEO-F103RB adatlap: https://www.st.com/resource/en/data_brief/nucleo-f103rb.pdf

HAL API dokumentáció: https://www.st.com/resource/en/user_manual/dm00154093-description-of-stm32f1-hal-and-lowlayer-drivers-stmicroelectronics.pdf

FreeRTOS referencia: https://www.freertos.org/fr-content-src/uploads/2018/07/FreeRTOS_Reference_Manual_V10.0.0.pdf

Felhasznált idő: 00:00/20:00

Elért pontszám: 0/34

1. feladat 0/4 pont

Melyik állítás igaz az ütemezőre?

Válaszok

- ☐ Akkor fut le, amikor egy taszk befejezi a futását és a vezérlést visszaadja az ütemezőnek
- ☐ Folyamatosan fut a háttérben
- ☒ A taszkok állapotát, esetleges prioritását figyelembe véve választja ki a futtatandó taszkot
Ez a válasz helyes, de nem jelölted meg.
- ☒ Módosítja a stack pointert
Ez a válasz helyes, de nem jelölted meg.

- ✓ Egy időzítő hardvermodul által generált megszakításhoz tartozó megszakításkezelő rutinban fut le
Ez a válasz helyes, de nem jelölted meg.
- ✓ Többféle algoritmus elterjedt a következő taszk kiválasztására, de lényeges szempont, hogy az ütemező a lehető legrövidebb idő alatt lefusson
Ez a válasz helyes, de nem jelölted meg.

Magyarázat

Az "Akkor fut le, amikor egy taszk befejezi a futását és a vezérlést visszaadja az ütemezőnek" és a "Folyamatosan fut a háttérben" válaszlehetőségek hamisak.

Az ütemező algoritmus egy időzítő által generált megszakítás hatására, egy megszakításkezelő rutinban fut le periodikusan. Tehát nincs szüksége arra, hogy az aktuálisan futó taszk explicit módon befejezze/felfüggeszse a működését. Az algoritmus kellően egyszerű és gyors ahhoz, hogy a megszakításkezelő rutinban lefusson, majd átadja a vezérlést a soron következő taszknak, tehát nem fut folyamatosan a háttérben.

2. feladat 0/10 pont

Szabolcs egy egyszerű LED-villogtatós példával szeretné kipróbálni, hogyan lehet FreeRTOS-ben taszkot létrehozni, de véletlenül benne hagyott egy hibát a kódban, ami miatt az – bár lefordul, de – nem működik.

Segíts neki megtalálni! Hányadik sorban van a hiba? (számmal add meg)

```
1 #include "main.h"
2 #include "cmsis_os.h"
3
4 #define STACK_SIZE 128
5
6 void SystemClock_Config(void);
7 static void MX_GPIO_Init(void);
8 static void MX_USART2_UART_Init(void);
9
10 UART_HandleTypeDef huart2;
11 TaskHandle_t myLEDTaskHandle = NULL;
12 StaticTask_t myLEDTaskControlBlock;
13
14 void myLEDTaskMain(void *argument) {
15     for(;;)
16     {
17         HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
18         osDelay(500);
19     }
20 }
21
22 void createLEDTask() {
23
24     StackType_t myLEDTaskStack[128];
25     myLEDTaskHandle = xTaskCreateStatic(
26         myLEDTaskMain,
27         "myLEDTask",
28         STACK_SIZE,
29         ( void * ) 1,
30         tskIDLE_PRIORITY,
31         myLEDTaskStack,
32         &myLEDTaskControlBlock);
33 }
34
35 int main(void) {
36     HAL_Init();
37     SystemClock_Config();
38     MX_GPIO_Init();
39     MX_USART2_UART_Init();
40     osKernelInitialize();
41
42     createLEDTask();
43
44     osKernelStart();
45     while (1)
46     {
47     }
48 }
```

Válasz

A helyes válasz:

24

Magyarázat

A helyes válasz: a 24. sorban van a hiba.

A taszkt az `xTaskCreateStatic()` függvényhívással hozzuk létre, így explicit módon megadhatjuk, hogy hol helyezkedjen el a taszk TCB-je és stackje a memóriában. A taszk stackjét a `myLEDTaskStack` tömbben tároljuk, viszont ezt a tömböt a `createLEDTask()` stackjén hoztuk létre, ami hiba, hiszen ez egy ideiglenes változó, a függvényből visszatérés után a helyét felszabadítjuk, így később az bármikor felülíródhat.

A legkézenfekvőbb helyes megoldás ezt a tömböt globális változóként létrehozni.

3. feladat 0/20 pont

Szabolcs több párhuzamosan futó taszkra bontja a szoftver által végzett feladatokat. Tudja, hogy bizonyos erőforrások nem szeretik, ha egyszerre több taszkból próbálják elérni őket, ezért mutex-szel védi a közös erőforrásokat. **Segíts neki kiválasztani az alábbi kódrészletek közül a helyesen működő változatot!**

Válaszok

☐

1.válasz:

```

/* Definitions for myLEDTask */
osThreadId_t myLEDTaskHandle;
const osThreadAttr_t myLEDTask_attributes = {
    .name = "myLEDTask",
    .stack_size = 128 * 4,
    .priority = (osPriority_t) osPriorityLow,
};
/* Definitions for myPIDTask */
osThreadId_t myPIDTaskHandle;
const osThreadAttr_t myPIDTask_attributes = {
    .name = "myPIDTask",
    .stack_size = 128 * 4,
    .priority = (osPriority_t) osPriorityHigh,
};
/* Definitions for mySonarTask */
osThreadId_t mySonarTaskHandle;
const osThreadAttr_t mySonarTask_attributes = {
    .name = "mySonarTask",
    .stack_size = 128 * 4,
    .priority = (osPriority_t) osPriorityNormal,
};
/* Definitions for myUSARTMutex */
osMutexId_t myUSARTMutexHandle;
const osMutexAttr_t myUSARTMutex_attributes = {
    .name = "myUSARTMutex"
};

int main(void) {
    HAL_Init();
    SystemClock_Config();
    MX_USART2_UART_Init();
    osKernelInitialize();

    myUSARTMutexHandle = osMutexNew(&myUSARTMutex_attributes);

    myLEDTaskHandle = osThreadNew(myLEDTaskMain, NULL, &myLEDTask_attributes);
    myPIDTaskHandle = osThreadNew(myPIDTaskMain, NULL, &myPIDTask_attributes);
    mySonarTaskHandle = osThreadNew(mySonarTaskMain, NULL, &mySonarTask_attributes);

    osKernelStart();
    while(1) {}
}

void myLEDTaskMain(void *argument) {
    for(;;) {
        char* msg = "LED control task is working OK.";
        controlLEDs();
        osMutexAcquire(myUSARTMutexHandle, 0);
        HAL_UART_Transmit(&huart2, msg, sizeof(msg), HAL_UART_TIMEOUT_VALUE);
        myLEDTaskPrintDebug();
        osMutexRelease(myUSARTMutexHandle);
        osDelay(100);
    }
}

void myPIDTaskMain(void *argument) {
    for(;;) {
        char* msg = "PID control task is working OK.";
        controlPIDs();
        osMutexAcquire(myUSARTMutexHandle, 0);
        HAL_UART_Transmit(&huart2, msg, sizeof(msg), HAL_UART_TIMEOUT_VALUE);
        myPIDTaskPrintDebug();
        osMutexRelease(myUSARTMutexHandle);
        osDelay(100);
    }
}

void mySonarTaskMain(void *argument) {
    for(;;) {
        char* msg = "HC-SR04 control task is working OK.";
        controlHCSR();
        osMutexAcquire(myUSARTMutexHandle, 0);
        HAL_UART_Transmit(&huart2, msg, sizeof(msg), HAL_UART_TIMEOUT_VALUE);
        mySonarTaskPrintDebug();
        osMutexRelease(myUSARTMutexHandle);
        osDelay(100);
    }
}

```

☐ 2.válasz:

```

/* Definitions for myLEDTask */
osThreadId_t myLEDTaskHandle;
const osThreadAttr_t myLEDTask_attributes = {
    .name = "myLEDTask",
    .stack_size = 128 * 4,
    .priority = (osPriority_t) osPriorityLow,
};
/* Definitions for myPIDTask */
osThreadId_t myPIDTaskHandle;
const osThreadAttr_t myPIDTask_attributes = {
    .name = "myPIDTask",
    .stack_size = 128 * 4,
    .priority = (osPriority_t) osPriorityHigh,
};
/* Definitions for mySonarTask */
osThreadId_t mySonarTaskHandle;
const osThreadAttr_t mySonarTask_attributes = {
    .name = "mySonarTask",
    .stack_size = 128 * 4,
    .priority = (osPriority_t) osPriorityNormal,
};
/* Definitions for myUSARTMutex */
osMutexId_t myUSARTMutexHandle;
const osMutexAttr_t myUSARTMutex_attributes = {
    .name = "myUSARTMutex"
};

int main(void) {
    HAL_Init();
    SystemClock_Config();
    MX_USART2_UART_Init();
    osKernelInitialize();

    myUSARTMutexHandle = osMutexNew(&myUSARTMutex_attributes);

    myLEDTaskHandle = osThreadNew(myLEDTaskMain, NULL, &myLEDTask_attributes);
    myPIDTaskHandle = osThreadNew(myPIDTaskMain, NULL, &myPIDTask_attributes);
    mySonarTaskHandle = osThreadNew(mySonarTaskMain, NULL, &mySonarTask_attributes);

    osKernelStart();
    while(1) {}
}

void myLEDTaskMain(void *argument) {
    for(;;) {
        char* msg = "LED control task is working OK.";
        controlLEDs();
        osMutexAcquire(myUSARTMutexHandle, 0);
        osThreadSetPriority(myLEDTaskHandle, osPriorityHigh);
        HAL_UART_Transmit(&huart2, msg, sizeof(msg), HAL_UART_TIMEOUT_VALUE);
        myLEDTaskPrintDebug();
        osMutexRelease(myUSARTMutexHandle);
        osThreadSetPriority(myLEDTaskHandle, osPriorityLow);
        osDelay(100);
    }
}

void myPIDTaskMain(void *argument) {
    for(;;) {
        char* msg = "PID control task is working OK.";
        controlPIDs();
        osMutexAcquire(myUSARTMutexHandle, 0);
        osThreadSetPriority(myPIDTaskHandle, osPriorityHigh);
        HAL_UART_Transmit(&huart2, msg, sizeof(msg), HAL_UART_TIMEOUT_VALUE);
        myPIDTaskPrintDebug();
        osMutexRelease(myUSARTMutexHandle);
        osThreadSetPriority(myPIDTaskHandle, osPriorityHigh);
        osDelay(100);
    }
}

void mySonarTaskMain(void *argument) {
    for(;;) {
        char* msg = "HC-SR04 control task is working OK.";
        controlHCSR();
        osMutexAcquire(myUSARTMutexHandle, 0);
        osThreadSetPriority(mySonarTaskHandle, osPriorityHigh);
        HAL_UART_Transmit(&huart2, msg, sizeof(msg), HAL_UART_TIMEOUT_VALUE);
        mySonarTaskPrintDebug();
        osMutexRelease(myUSARTMutexHandle);
    }
}

```

```
osThreadSetPriority(mySonarTaskHandle, osPriorityNormal);  
osDelay(100);  
}  
}
```



3.válasz:


```

/* Definitions for myLEDTask */
osThreadId_t myLEDTaskHandle;
const osThreadAttr_t myLEDTask_attributes = {
    .name = "myLEDTask",
    .stack_size = 128 * 4,
    .priority = (osPriority_t) osPriorityLow,
};
/* Definitions for myPIDTask */
osThreadId_t myPIDTaskHandle;
const osThreadAttr_t myPIDTask_attributes = {
    .name = "myPIDTask",
    .stack_size = 128 * 4,
    .priority = (osPriority_t) osPriorityHigh,
};
/* Definitions for mySonarTask */
osThreadId_t mySonarTaskHandle;
const osThreadAttr_t mySonarTask_attributes = {
    .name = "mySonarTask",
    .stack_size = 128 * 4,
    .priority = (osPriority_t) osPriorityNormal,
};
/* Definitions for myUSARTMutex */
osMutexId_t myUSARTMutexHandle;
const osMutexAttr_t myUSARTMutex_attributes = {
    .name = "myUSARTMutex"
};

int main(void) {
    HAL_Init();
    SystemClock_Config();
    MX_USART2_UART_Init();
    osKernelInitialize();

    myUSARTMutexHandle = osMutexNew(&myUSARTMutex_attributes);

    myLEDTaskHandle = osThreadNew(myLEDTaskMain, NULL, &myLEDTask_attributes);
    myPIDTaskHandle = osThreadNew(myPIDTaskMain, NULL, &myPIDTask_attributes);
    mySonarTaskHandle = osThreadNew(mySonarTaskMain, NULL, &mySonarTask_attributes);

    osKernelStart();
    while(1) {}
}

void myLEDTaskMain(void *argument) {
    for(;;) {
        char* msg = "LED control task is working OK.";
        controlLEDs();
        osThreadSetPriority(myLEDTaskHandle, osPriorityHigh);
        osMutexAcquire(myUSARTMutexHandle, 0);
        HAL_UART_Transmit(&huart2, msg, sizeof(msg), HAL_UART_TIMEOUT_VALUE);
        myLEDTaskPrintDebug();
        osMutexRelease(myUSARTMutexHandle);
        osThreadSetPriority(myLEDTaskHandle, osPriorityLow);
        osDelay(100);
    }
}

void myPIDTaskMain(void *argument) {
    for(;;) {
        char* msg = "PID control task is working OK.";
        controlPIDs();
        osMutexAcquire(myUSARTMutexHandle, 0);
        HAL_UART_Transmit(&huart2, msg, sizeof(msg), HAL_UART_TIMEOUT_VALUE);
        myPIDTaskPrintDebug();
        osMutexRelease(myUSARTMutexHandle);
        osDelay(100);
    }
}

void mySonarTaskMain(void *argument) {
    for(;;) {
        char* msg = "HC-SR04 control task is working OK.";
        controlHCSR();
        osThreadSetPriority(mySonarTaskHandle, osPriorityHigh);
        osMutexAcquire(myUSARTMutexHandle, 0);
        HAL_UART_Transmit(&huart2, msg, sizeof(msg), HAL_UART_TIMEOUT_VALUE);
        mySonarTaskPrintDebug();
        osMutexRelease(myUSARTMutexHandle);
        osThreadSetPriority(mySonarTaskHandle, osPriorityNormal);
        osDelay(100);
    }
}

```

```
}  
}  
}
```



4.válasz:

```

/* Definitions for myLEDTask */
osThreadId_t myLEDTaskHandle;
const osThreadAttr_t myLEDTask_attributes = {
    .name = "myLEDTask",
    .stack_size = 128 * 4,
    .priority = (osPriority_t) osPriorityLow,
};

/* Definitions for myPIDTask */
osThreadId_t myPIDTaskHandle;
const osThreadAttr_t myPIDTask_attributes = {
    .name = "myPIDTask",
    .stack_size = 128 * 4,
    .priority = (osPriority_t) osPriorityHigh,
};

/* Definitions for mySonarTask */
osThreadId_t mySonarTaskHandle;
const osThreadAttr_t mySonarTask_attributes = {
    .name = "mySonarTask",
    .stack_size = 128 * 4,
    .priority = (osPriority_t) osPriorityNormal,
};

/* Definitions for myUsartTask */
osThreadId_t myUsartTaskHandle;
const osThreadAttr_t myUsartTask_attributes = {
    .name = "myUsartTask",
    .stack_size = 128 * 4,
    .priority = (osPriority_t) osPriorityNormal,
};

/* Definitions for myUSARTQueue */
#define MSGQUEUE_OBJECTS 16
osMessageQueueId_t myUSARTQueueHandle;
const osMessageQueueAttr_t myUSARTQueue_attributes = {
    .name = "myUSARTQueue"
};

int main(void) {
    HAL_Init();
    SystemClock_Config();
    MX_USART2_UART_Init();
    osKernelInitialize();

    myUSARTQueueHandle = osMessageQueueNew (MSGQUEUE_OBJECTS, 64, &myUSARTQueue_attributes);

    myLEDTaskHandle = osThreadNew(myLEDTaskMain, NULL, &myLEDTask_attributes);
    myPIDTaskHandle = osThreadNew(myPIDTaskMain, NULL, &myPIDTask_attributes);
    mySonarTaskHandle = osThreadNew(mySonarTaskMain, NULL, &mySonarTask_attributes);
    myUsartTaskHandle = osThreadNew(myUsartTaskMain, NULL, &myUsartTask_attributes);

    osKernelStart();
    while(1) {}
}

void myLEDTaskMain(void *argument) {
    for(;;) {
        char* msg = "LED control task is working OK.";
        controlLEDs();
        osMessageQueuePut(myUSARTQueueHandle, msg, 0, 0);
        myLEDTaskSendDebug();
        osDelay(100);
    }
}

void myPIDTaskMain(void *argument) {
    for(;;) {
        char* msg = "PID control task is working OK.";
        controlPIDs();
        osMessageQueuePut(myUSARTQueueHandle, msg, 0, 0);
        myPIDTaskSendDebug();
        osDelay(100);
    }
}

void mySonarTaskMain(void *argument) {
    for(;;) {
        char* msg = "HC-SR04 control task is working OK.";
        controlHCSR();
        osMessageQueuePut(myUSARTQueueHandle, msg, 0, 0);
        mySonarTaskSendDebug();
        osDelay(100);
    }
}

void myUsartTaskMain(void *argument) {
    for(;;) {
        char msg[64];
        status = osMessageQueueGet(myUSARTQueueHandle, &msg, 0, 0);
        if(status == osOK)

```

```
    HAL_UART_Transmit(&huart2, msg, sizeof(msg), HAL_UART_TIMEOUT_VALUE);  
  }  
}
```

Ez a válasz helyes, de nem jelölted meg.

Magyarázat

Az első és a második megoldás esetén is előfordulhat prioritás inverzió, valamint abból eredő deadlock.

Előfordulhat, hogy az alacsony prioritású taszk megszerzi a mutexet, majd a magas prioritású taszk is megpróbálja, de nem sikerül neki. Ekkor az ütemező a közepes prioritású taszknak adja a vezérlést, ami korlátlan ideig futhat (hiszen a magasabb prioritású taszk vár a mutexre, az alacsonyabb prioritású taszk pedig... alacsonyabb prioritású).

A második megoldás esetén Szabolcs megpróbálta a prioritásplafon protokollal orvosolni a problémát, de a taszk prioritását csak a mutex megszerzése után emelné meg. Szerencsétlen esetben erre azonban már egyáltalán nem biztos, hogy van lehetősége.

A prioritásplafon protokollt a harmadik megoldásban implementálja helyesen.

A negyedik megoldásban egy másik stratégiához folyamodott: egy önálló taszkot hozott létre a szűk erőforrás kezelésére, ami egy üzenetsoron keresztül várja a feldolgozandó adatokat. A többi taszk ebbe az üzenetsorba küldi be a kiíratandó üzeneteit, így elkerülhető a fentebb vázolt szerencsétlen helyzet.



[Legfontosabb tudnivalók](#)

[Kapcsolat](#)

[Versenyszabályzat](#)

[Adatvédelem](#)

© 2023 Human Priority Kft.

KÉSZÍTETTE 

Megjelenés

 Világos 