



System FreeRTOS a obsluha akcelerometru

1 Zadání

- Vytvořte projekt s aktivovanou podporou systému FreeRTOS. V základním systému poběží úloha vizualizace VisualTask, která čte frontu xVisualQueue a dle hodnot aktivuje dvojici LED diod LED1-LED2, a demo úloha AcceleroTask, která do fronty periodicky posílá testovací data tak, aby LED střídavě s prodlevou blikaly.
- Doplňte do projektu driver akcelerometru LIS2DW12. Údaje z akcelerometru vypisujte v rámci úlohy AcceleroTask na sériový port. Upravte úlohu AcceleroTask tak, aby místo testovacích dat posílala do fronty údaje z osy X akcelerometru – LED diody tedy budou svítit podle náklonu desky.

2 Návod

2.1 Základní seznámení

- Vytvořte si pracovní kopii svého repozitáře z Githubu (Git Clone), příp. aktualizujte repozitář ze serveru (Git Pull).
- Založte nový projekt přes File / New / STM32 Project / Board Selector / NUCLEO-F030R8. Budeme využívat HAL knihovny, proto ponechte Targeted Project Type na STM32Cube. Potvrďte inicializaci všech periférií do výchozího nastavení.
- Budeme využívat LED diody (LED1 @ PA4, LED2 @ PB0, push-pull výstup).
- FreeRTOS je preemptivní operační systém (plánovač), který umožňuje paralelní běh více úloh. Každá úloha se píše do samostatné funkce, která může obsahovat inicializaci a následuje vždy nekonečná smyčka s kódem úlohy. Úlohy mezi sebou komunikují prostřednictvím front, zpráv, semaforů, mutexů atd.

2.2 Konfigurace FreeRTOS

- V konfiguraci aktivujte middleware FreeRTOS (Interface CMSIS_V1). Vytvořte obě úlohy a frontu pro typ int16_t:

Tasks and Queues	Timers and Semaphores	Mutexes	FreeRTOS Heap Usage					
Config parameters	Include parameters	Advanced settings	User Constants					
Tasks								
Task Name	Priority	St...	Entry Function	Code ...	Para...	Allocation	Buffer...	Contr...
defaultTask	osPriorityNormal	128	StartDefaultTask	Default	NULL	Dynamic	NULL	NULL
VisualTask	osPriorityNormal	128	StartVisualTask	Default	NULL	Dynamic	NULL	NULL
AcceleroTask	osPriorityNormal	128	StartAcceleroT...	Default	NULL	Dynamic	NULL	NULL
<div>Add Delete</div>								
Queues								
Queue Name	Queue Size	Item Size	Allocation	Buffer Name	Control Block ...			
xVisualQueue	16	int16_t	Dynamic	NULL	NULL			
<div>Add Delete</div>								

- Dle doporučení změňte zdroj časové základny (SYS – Timebase Source) ze SysTick na např. TIM14.



Mikrokontroléry a embedded systémy – cvičení

- Úloha vizualizace bude v nekonečné smyčce přijímat data z fronty xVisualQueue a na základě hodnoty (msg) rozhodovat o rozsvícení LED1 a LED2. LED1 bude rozsvícena při hodnotě pod -1000, LED2 při hodnotě nad +1000:

```
int16_t msg;

if (xQueueReceive(xVisualQueueHandle, &msg, portMAX_DELAY)) {
    ...
    ...
}
```

- Demo úloha bude v nekonečné smyčce plnit frontu, tedy odesílat zprávy např. -5000, 0, 5000, 0, atd. Za odeslání do fronty se přidá krátké čekání:

```
xQueueSend(xVisualQueueHandle, &msg, 0);
osDelay(300);
```

2.3 Podpora akcelerometru LIS2DW12

- Pro akcelerometry STM výrobce poskytuje standardní platformě nezávislou knihovnu:
https://github.com/STMicroelectronics/STMems_Standard_C_drivers
- Do projektu přidejte soubory lis2dw12_reg.c a lis2dw12_reg.h ze složky /lis2dw12_STdC/driver/. Příklady použití lze nalézt v adresáři example.
- Akcelerometr je připojený přes I2C rozhraní, povolte tedy na periférii I2C1 režim I2C a přemapujte SDA a SCL na odpovídající piny (PB8, PB9 – postup viz cvičení *UART komunikace s DMA, EEPROM na I2C*).
- V rámci portování na platformu je potřeba definovat kontext zařízení:

```
static int32_t platform_write(void *handle, uint8_t reg, uint8_t *bufp, uint16_t len);
static int32_t platform_read(void *handle, uint8_t reg, uint8_t *bufp, uint16_t len);
static stmdev_ctx_t lis2dw12 = {
    .write_reg = platform_write,
    .read_reg = platform_read,
    .handle = &hi2c1,
};
```

- A dále funkce pro čtení a zápis:

```
/*
 * Replace the functions "platform_write" and "platform_read" with your
 * platform specific read and write function.
 */
static int32_t platform_write(void *handle, uint8_t reg, uint8_t *bufp, uint16_t len)
{
    HAL_I2C_Mem_Write(handle, LIS2DW12_I2C_ADD_H, reg, I2C_MEMADD_SIZE_8BIT, bufp, len, 1000);
    return 0;
}

static int32_t platform_read(void *handle, uint8_t reg, uint8_t *bufp, uint16_t len)
{
    HAL_I2C_Mem_Read(handle, LIS2DW12_I2C_ADD_H, reg, I2C_MEMADD_SIZE_8BIT, bufp, len, 1000);
    return 0;
}
```



Mikrokontroléry a embedded systémy – cvičení

- Nejdříve je vhodné v rámci AcceleroTasku otestovat komunikaci s akcelerometrem a její výsledek vypsát na sériový výstup:

```
// Check device ID
uint8_t whoamI = 0;
lis2dw12_device_id_get(&lis2dw12, &whoamI);
printf("LIS2DW12_ID %s\n", (whoamI == LIS2DW12_ID) ? "OK" : "FAIL");
```

- Aby fungoval výstup na UART, je třeba předefinovat funkci `_write()`, postup viz cvičení *UART komunikace s DMA, EEPROM na I2C*.

- Použití funkce `printf()` je pod RTOS problematické, protože dynamicky alokuje paměť. Vysvětlení (pro zájemce):

- Knihovna `newlib` udržuje řadu statických proměnných ve struktuře `struct _reent` na adrese `_impure_ptr`. Ty jsou inicializovány v rámci inicializačního kódu kopírováním segmentu `.data`. Součástí jsou však i ukazatele na standardní deskriptory `stdin`, `stdout` a `stderr`, které jsou alokovány dynamicky z haldy při prvním použití některé z funkcí standardního vstupu nebo výstupu.
- Dynamická alokace provedená pomocí `malloc()` spoléhá na systémovou funkci `_sbrk()`, která je implementovaná v `system.c` od STM. Funkce postupně alokuje z haldy, při každé alokaci ověřuje, zda halda nezasáhla do zásobníku. V případě použití libovolného RTOS má ale každá úloha vlastní zásobník. Alokaci z haldy by proto měl řešit RTOS.
- Kromě problémů s alokací existuje řada problémů s reentrancí. Většina standardních funkcí nesmí být zároveň použita v různých vláknech, pokud nejsou zabezpečeny pomocí mutexů nebo prostřednictvím `configUSE_NEWLIB_REENTRANT`.
- Pro většinu RTOS aplikací je obvykle snazší **použít jinou implementaci** `printf()` než standardní `newlib`.
- Detailní rozbor problému: <http://www.nadler.com/embedded/newlibAndFreeRTOS.html>.

- Inicializujeme funkci akcelerometru (volně dle `lis2dw12_read_data_single.c`):

```
lis2dw12_full_scale_set(&lis2dw12, LIS2DW12_2g);
lis2dw12_power_mode_set(&lis2dw12, LIS2DW12_CONT_LOW_PWR_LOW_NOISE_2);
lis2dw12_block_data_update_set(&lis2dw12, PROPERTY_ENABLE);
lis2dw12_fifo_mode_set(&lis2dw12, LIS2DW12_STREAM_MODE); // enable continuous FIFO
lis2dw12_data_rate_set(&lis2dw12, LIS2DW12_XL_ODR_25Hz); // enable part from power-down
```

- A následně čteme FIFO paměť akcelerometru, je třeba vždy vyčíst celou FIFO:

```
uint8_t samples;
int16_t raw_acceleration[3];

lis2dw12_fifo_data_level_get(&lis2dw12, &samples);
for (uint8_t i = 0; i < samples; i++) {
    // Read acceleration data
    lis2dw12_acceleration_raw_get(&lis2dw12, raw_acceleration);
    printf("X=%d Y=%d Z=%d\n", raw_acceleration[0], raw_acceleration[1], raw_acceleration[2]);
}
```

- Vzorek z akcelerometru stačí pro výstup do terminálu aktualizovat po 1 sekundě, pro odeslání do fronty `xVisualQueue` po cca 50ms. Vzhledem k preemptivnímu charakteru plánovače RTOS je využívá blokující čekání realizované funkcí `osDelay()`. Nejjednodušším řešením je číst vždy všechny vzorky (celou FIFO) s požadovanou periodou aktualizace, ale použít jen poslední z nich.

- Předávání dat do fronty `xVisualQueue` je prosté, posílá se vzorek `raw_acceleration[0]`.

- Na závěr provedte commit pracovní kopie do Gitu, uložte repozitář pomocí `Git Push`.

- Z využití paměti reportovaného při kompilaci je zřejmé, že použitý mikrokontrolér je na hraně svých možností. Systémů jako `FreeRTOS` se zpravidla používají na vyšších MCU (STM řady F1, F4 apod.), MCU s jádrem `Cortex-M0` si obvykle lépe vystačí s kooperativním řešením pomocí `supersmyčky`.