Osnove mikroprocesorske elektronike

Vaja 9: Časovniki

V sklopu te vaje boste *na sistemskem nivoju* implementirali dva modula, ki nam *bosta pomagala izvajati programsko kodo ob pravih trenutkih v času*. Prvi modul bo namenjen *merjenju pretečenega časa* po principu ure štoparice. Drugi modul pa nam bo omogočal *periodično izvajanje časovno nekritičnih rutin* s točno določeno periodo. Oba modula pa boste seveda implementirali s pomočjo *časovnikov*.

Naloge vaje – merjenje časa

Vaša naloga je, da modul za merjenje časa `timing_utils` *dopolnite do konca* in tako implementirate funkcionalnost, ki je bila predstavljena v sklopu priprave v poglavjih v zvezi s postopkom za merjenje pretečenega časa.

To pomeni, da boste morali opraviti sledeče naloge:

1. ***v zglavni datoteki*** `timing_utils.h` ***dopolniti komentar s seznamom HAL funkcij***, ki jih modul uporablja.

2. ***V datoteki*** `timing_utils.h` ***dopolniti sledeče ključne funkcije modula:***

   a) `TIMUT_stopwatch_set_time_mark()` – funkcija, ki za uro štoparico *postavi časovni zaznamek* (angl. time mark),

   b) `TIMUT_stopwatch_update()` – funkcija, ki za uro štoparico *posodobi vrednost pretečenega časa* od trenutka, kjer smo postavili časovni zaznamek,

   c) `TIMUT_stopwatch_has_X_ms_passed()` – funkcija, ki za uro štoparico preveri, ali je od postavitve časovnega zaznamka že preteklo "x" milisekund, kjer pa je "x" vhodni argument funkcije.

   d) `TIMUT_stopwatch_has_another_X_ms_passed()` – funkcija, ki je namenjena periodičnemu preverjanju, ali je že preteklo "x" milisekund od zadnjega časovnega zaznamka, kjer pa je "x" vhodni argument funkcije.

   Pri implementaciji funkcij vas bodo vodili komentarji v datotekah modula.

3. ***Zgoraj implementirane funkcije boste stestirali s pomočjo demonstracijske funkcije*** `TIMUT_stopwatch_demo()`, kjer boste s pomočjo teh funkcij implementirali utripanje LEDic.

   Demonstracijsko funkcijo `TIMUT_stopwatch_demo()` seveda kličite iz glavne zanke `main.c` modula.

Naloge vaje – periodično izvajanje rutine

Modul za periodično izvajanje časovno nekritičnih rutin boste tokrat *v celoti implementirali sami*. Pri tem se *zgledujte* po modulih, ki ste jih implementirali tekom preteklih vaj. Držite se torej *metodologije*, ki smo jo predstavili v sklopu 6. vaje.

Spodaj povzemamo ključne korake, ki jih boste morali izvesti v sklopu implementacije tega modula:

4. ***V zglavni datoteki*** `periodic_services.h` ***bo potrebno:***
   a) v komentarju navesti, katere LL funkcije modul potrebuje za implementacijo (angl. dependencies),
   b) definirati "javne" funkcije modula s pomočjo prototipov.

5. ***V datoteki z izvorno kodo*** `periodic_services.c` ***bo potrebno:***
   a) ***vključiti potrebne knjižnice***
      i. lastne definicije modula `periodic_services` v zglavni datoteki,
      ii. LL knjižnico za podporo za delo s časovniki,
      iii. knjižnice ostalih *sistemskih* modulov, katerih rutine bomo izvajali periodično,

   b) ***definirati podatkovni tip za "handle" strukturo*** `periodic_services_handle_t`, ki bo *hranila vse, kar je potrebno, da upravljamo periodično izvajanje rutin* (namig: potrebno bo hraniti le en parameter),

   c) ***na podlagi tega tipa definirati "privatno" globalno strukturno spremenljivko*** `periodic_services`,

   d) ***implementirati sledeče "javne" funkcije s sledečo vsebino:***
      i. `void PSERV_init(void)`:
         1) inicializacija "handle" strukture `periodic_services`, kjer se specificira, kateri časovnik bo uporabljen pri implementaciji,
         2) zagon tega časovnika (tj. da časovnik prične s štetjem),
      ii. `void PSERV_enable(void)`: kjer *omogočite periodično izvajanje rutin* tako, da *omogočite prekinitve časovnika* ob dogodku *preliva* (pri STM časovnikih: "update event"). Periodični dogodek preliva boste uporabili za "avtomatsko" periodično izvajanje rutin.
      iii. `void PSERV_disable(void)`: kjer *onemogočite periodično izvajanje rutin* tako, da *onemogočite prekinitve časovnika* ob dogodku *preliva*,

iv. `void PSERV_run_services_Callback(void)`: "callback" funkcija, ki se bo izvajala periodično s pomočjo prekinitev časovnika in bo poskrbela za periodično izvajanje rutin. Znotraj te funkcije boste torej poklicali tiste časovno nekritične rutine, ki jih želite izvajati periodično.

Zaenkrat ***poskrbite, da se periodično izvaja sledeča funkcionalnost***:

1) izvaja naj *branje stanja tipkovnice* ("tj. skeniranje" tipkovnice),
2) po branju stanja tipkovnice naj se izvede še *obdelava informacije o pritisnjenih tipkah*, ki je implementirana v pomožni demonstracijski funkciji `KBD_demo_toggle_LEDs_if_buttons_pressed()`, in jo je potrebno še dopolniti do polne funkcionalnosti,

6. ***V projektni datoteki*** `stm32g4xx_it.c` ***bo potrebno dopolniti prekinitveno rutino*** `TIM6_DAC_IRQHandler()`.

Zgledujte se po implementaciji, ki smo jo uporabili pri prekinitvah v zvezi z USART prekinitvami pri SCI serijskem vmesniku. Metodologija je popolnoma enaka:

a) preverite, če so ustrezne prekinitve omogočene,

b) preverite, če je postavljena ustrezna zastavica,

c) pokličite ustrezno "callback" rutino.

Tokrat je *potreben še dodaten korak*. Namig najdete na spodnjem izseku iz dokumentacije:

31.4.4 TIMx status register (TIMx_SR)(x = 6 to 7)

Address offset: 0x10

Reset value: 0x0000

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	Res.	UIF
															rc_w0

Bits 15:1 Reserved, must be kept at reset value.

Bit 0 **UIF**: Update interrupt flag
This bit is set by hardware on an update event. It is cleared by software.
0: No update occurred.
1: Update interrupt pending. This bit is set by hardware when the registers are updated:
– On counter overflow if UDIS = 0 in the TIMx_CR1 register.
– When CNT is reinitialized by software using the UG bit in the TIMx_EGR register, if URS = 0 and UDIS = 0 in the TIMx_CR1 register.

7. ***Stestirajte in demonstrirajte delovanje "avtomatskega" periodičnega izvajanja rutin.***

Najbolj ilustrativno to storite tako, da znotraj `main.c` poskrbite, da je neskončna `while(1)` zanka prazna!

Dodatno – demonstracija uporabe stanja z nizko porabo

Ko boste uspešno implementirali zgornja modula, pa lahko pri tej vaji *preučite še možnost uporabe stanja mikrokrmilnika z nizko porabo*, natančneje *stanje spanja* (angl. sleep mode). V spodnjih izsekih si lahko pogledate, kako bi razmišljali, ko bi pripravljali programsko kodo za demonstracijo "sleep mode" delovanja. Kočni rezultat programske kode najdete v mapi "predloge".

Izhajamo iz demonstracije "avtomatskega" periodičnega izvajanja rutin:

```
// Glavna neskončna zanka v "main.c".
while (1)
{

    // Najbolj ilustrativno demonstriramo "avtomatsko" periodično izvajanje rutin tako,
    // da glavno zanko v main.c pustimo prazno. Torej v glavnem programu se ne izvaja nič!

}
```

Demonstracijo dopolnimo tako, da dodamo ukaz, ki mikrokrmilnik "porine" v stanje spanja.

```
// Glavna neskončna zanka v "main.c".
while (1)
{

    // Najbolj ilustrativno demonstriramo "avtomatsko" periodično izvajanje rutin tako,
    // da glavno zanko v main.c pustimo prazno. Torej v glavnem programu se ne izvaja nič!

    // Če pa mikrokrmilnik nima posebnega dela v glavni zanki, je pa smiselno mikrokrmilnik
    // tu postaviti v stanje z nizko porabo energije!

    // S pomočjo HAL knjižnice mikrokrmilnik postavimo v osnovno stanje nizke porabe - "sleep mode".
    // Parametra funkcije povesta, naj glavni napetostni regulator ostane vklopljen ter naj se
    // mikrokrmilnik zbudi iz spanja ob prekinitvah.
    HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFI);

}
```

Da pa bi lažje ugotovili, kdaj se mikrokrmilnik zbudi iz spanja, bi bilo smiselno dodati preprosto LED indikacijo:

```
// Glavna neskončna zanka v "main.c".
while (1)
{

    // Najbolj ilustrativno demonstriramo "avtomatsko" periodično izvajanje rutin tako,
    // da glavno zanko v main.c pustimo prazno. Torej v glavnem programu se ne izvaja nič!

    // Če pa mikrokrmilnik nima posebnega dela v glavni zanki, je pa smiselno mikrokrmilnik
    // tu postaviti v stanje z nizko porabo energije!

    // S pomočjo HAL knjižnice mikrokrmilnik postavimo v osnovno stanje nizke porabe - "sleep mode".
    // Parametra funkcije povesta, naj glavni napetostni regulator ostane vklopljen ter naj se
    // mikrokrmilnik zbudi iz spanja ob prekinitvah.
    HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFI);


    // Uporabimo LEDico LED7 za indikacijo, da se je procesor zbudil iz stanja spanja.
    LED_toggle(LED7);

}
```

Z dodano LED indikacijo bi ugotovili, da kljub temu, da vaš modul `periodic_services` zbuja mikrokrmilnik vsakih 50 milisekund, se mikrokrmilnik dejansko zbuja mnogo pogosteje! Razlog je v tem, da ga zbuja `SysTick` prekinitev, ki se proži vsako milisekundo! Zato dodamo še kodo, ki onemogoči `SysTick` prekinitev.

```
// Če želimo resnično zagotoviti "nemoteno spanje mikrokrmilnika" in če ne potrebujemo
// števca milisekund SysTick, potem lahko onemogočimo prekinitve ob "tiktakanju"
// SysTick števca s klicem funkcije HAL_SuspendTick().
// Poskusite spodnjo vrsto zakomentirati oz. odkomentirati.
HAL_SuspendTick();


// Glavna neskončna zanka v "main.c".
while (1)
{

    // Najbolj ilustrativno demonstriramo "avtomatsko" periodično izvajanje rutin tako,
    // da glavno zanko v main.c pustimo prazno. Torej, v glavnem programu se ne izvaja nič!

    // Če pa mikrokrmilnik nima posebnega dela v glavni zanki, je pa smiselno mikrokrmilnik
    // tu postaviti v stanje z nizko porabo energije!

    // S pomočjo HAL knjižnice mikrokrmilnik postavimo v osnovno stanje nizke porabe - "sleep mode".
    // Parametra funkcije povesta, naj glavni napetostni regulator ostane vklopljen ter naj se
    // mikrokrmilnik zbudi iz spanja ob prekinitvah.
    HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFI);


    // Uporabimo LEDico LED7 za indikacijo, da se je procesor zbudil iz stanja spanja.
    LED_toggle(LED7);

}
```

V zadnjem koraku pa se odločimo, da bi bilo vseeno zanimivo pogledati, kako bi se stvar obnašala, če bi v glavni zanki izvajali še demonstracijo `timing_utils` modula. In jo zato dodamo.

```
// Če želimo resnično zagotoviti "nemoteno spanje mikrokrmilnika" in če ne potrebujemo
// števca milisekund SysTick, potem lahko onemogočimo prekinitve ob "tiktakanju"
// SysTick števca s klicem funkcije HAL_SuspendTick().
// Poskusite spodnjo vrsto zakomentirati oz. odkomentirati.
HAL_SuspendTick();


// Glavna neskončna zanka v "main.c".
while (1)
{

    // Demonstracija uporabe modula za merjenje pretečenega časa.
    TIMUT_stopwatch_demo();

    // Če pa mikrokrmilnik nima posebnega dela v glavni zanki, je pa smiselno mikrokrmilnik
    // tu postaviti v stanje z nizko porabo energije!

    // S pomočjo HAL knjižnice mikrokrmilnik postavimo v osnovno stanje nizke porabe - "sleep mode".
    // Parametra funkcije povesta, naj glavni napetostni regulator ostane vklopljen ter naj se
    // mikrokrmilnik zbudi iz spanja ob prekinitvah.
    HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON, PWR_SLEEPENTRY_WFI);


    // Uporabimo LEDico LED7 za indikacijo, da se je procesor zbudil iz stanja spanja.
    LED_toggle(LED7);

}
```

Tako ste prišli do kode, ki vam omogoča, da raziskujete obnašanje mikrokrmilnika v stanju nizke porabe. ***Kodo najdete v mapi "predloge"***. Poskušajte zakomentirati klic funkcije `HAL_SuspendTick()` in opazujte, kaj se bo zgodilo.