Osnove mikroprocesorske elektronike

Priprava 9: Časovniki

V sklopu te vaje bomo 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 bomo seveda implementirali s pomočjo časovnikov.

Priprava novega projekta

1. ***S pomočjo orodja*** `STM Cube Project Copy` ***ustvarite nov projekt z imenom***

   `VAJA_09-timers`

   na podlagi projekta, kjer ste rešili prejšnjo vajo. Projekt nato uvozite v "STM Cube IDE".

Priprava na vajo – merjenje časa

2. ***Novemu projektu dodajte modula*** `timing_utils.c` ***in*** `timing_utils.h`, znotraj katerega bomo v sklopu vaje implementirali funkcionalnost za merjenje časa po principu ure štoparice. Modula najdete v mapi "predloge". Dodajte ju v projektno mapo znotraj podmape "`System`" na ustrezno mesto.

3. ***S pomočjo HAL dokumentacije ugotovite***, katero HAL funkcijo (*in ne LL funkcijo*) je potrebno uporabiti, če želimo dobiti trenutno vrednost `SysTick` števca milisekund.

4. ***Preučite dva glavna postopka za merjenje pretečenega časa***, na podlagi katerih bomo implementirali merjenje časa s pomočjo ure štoparice.

   Opis postopkov najdete v poglavju spodaj.

Priprava na vajo – periodično izvajanje rutin

Priprava v CubeMX

5. ***V orodju CubeMX preverite in zagotovite, da je vaša sistemska ura nastavljena pravilno*** (`SYSCLK` = 144 MHz).

6. ***Premislite sledečo odločitev:*** za implementacijo periodičnega izvajanja rutin bomo uporabili *osnovni* časovnik `TIM6`. Poglejte tabelo spodaj.

Table 7. Timer feature comparison (continued)

Timer type	Timer	Counter resolution	Counter type	Prescaler factor	DMA request generation	Capture/ compare channels	Complementary outputs
General-purpose	TIM3, TIM4	16-bit	Up, down, Up/down	Any integer between 1 and 65536	Yes	4	No
General-purpose	TIM15	16-bit	Up	Any integer between 1 and 65536	Yes	2	1
General-purpose	TIM16, TIM17	16-bit	Up	Any integer between 1 and 65536	Yes	1	1
Basic	TIM6, TIM7	16-bit	Up	Any integer between 1 and 65536	Yes	0	No

7. **V orodju CubeMX omogočite časovnik `TIM6` in poskrbite za sledeče nastavitve:**

   a) ustrezno nastavite *delilnik frekvence* (angl. prescaler) tako, da bo *osnovna časovna enota* časovnika 1 mikrosekunda,

   b) *modul* časovnika naj bo 50 milisekund (angl. counter period),

   c) časovniku *omogočite globalne prekinitve* (rubrika "NVIC settings"),

   d) za delo s časovniki naj se uporablja *nizko-nivojske LL knjižnice* in ne HAL knjižnica (rubrika "Project Manager → Advanced Settings → Driver Selector").

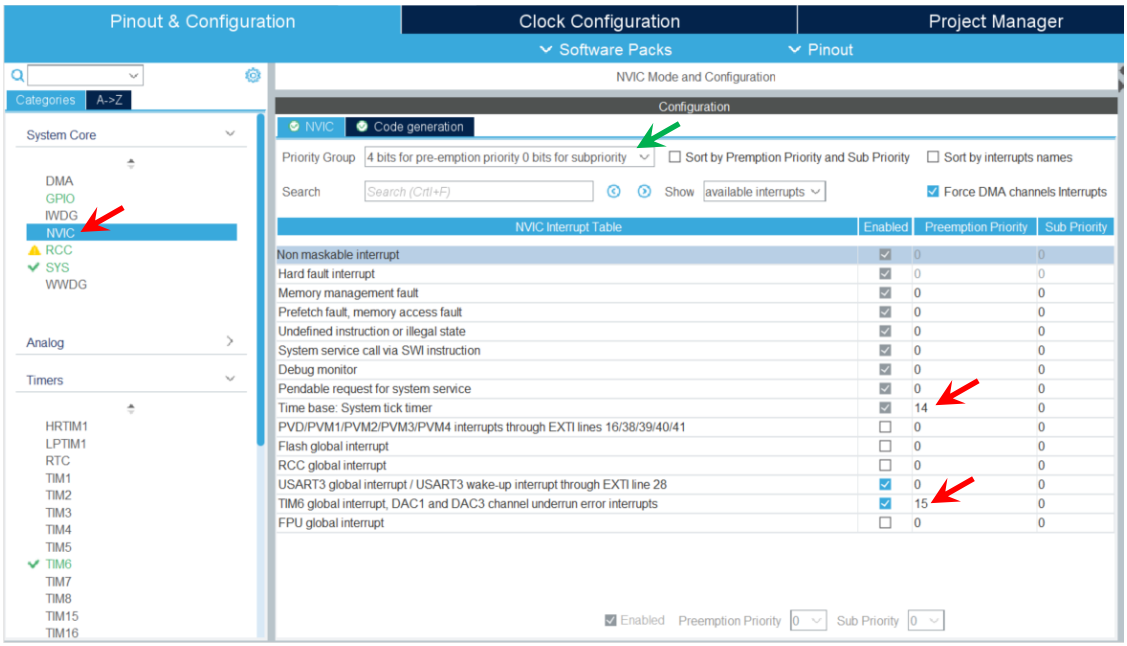
   Pri tem si pomagajte z video lekcijo "Timers and Timer Interrupts".

   OPOZORILO: ne uporabljajte nasveta z video lekcije, da vrednost delilnika frekvence in modula določate s pomočjo odštevanja z "−1". Ta pristop ne deluje dobro in lahko boste imeli težave. *V polje torej vpišite končni rezultat in ne matematični izraz.*

8. ***Poskrbite za smiselno nastavitev prioritet prekinitev.***

Razmislek: s pomočjo časovnika želimo implementirati periodično izvajanje *preprostih rutin z nizko prioriteto, ki niso časovno občutljive* (angl. time in-sensitive functions). Taka rutina je v našem primeru branje trenutnega stanja tipkovnice. Zato je smiselno, da ima prekinitev časovnika `TIM6`, ki bo poskrbela za periodično zaganjanje rutin, *najnižjo prekinitveno prioriteto*. Vse ostale prekinitve naj imajo višjo prioriteto. Na primer, za prekinitve USART vmesnika, s katerim smo implementirali SCI modul, želimo zelo visoko prioriteto, saj želimo preprečiti izgube podatkov pri sprejemu zaradi slabe odzivnosti sistema.

Prioritete prekinitev nastavljate znotraj orodja CubeMX v rubriki "System Core → NVIC". Poglejte spodaj. Poskrbite, da izklopite možnost "pod-prioritet" (angl. subpriority, zelena puščica).



9. ***Nastavitve v orodju CubeMX shranite in preverite, kje se je sedaj dodala avtomatsko generirana koda.***

Orodje CubeMX bo poskrbelo za *nizko-nivojsko inicializacijo strojne opreme*, torej za inicializacijo časovnika `TIM6`.

```
/* Initialize all configured peripherals */
MX_GPIO_Init();
MX_USART3_UART_Init();
MX_TIM6_Init();
/* USER CODE BEGIN 2 */
```

In ker ste časovniku TIM6 omogočili globalne prekinitve, se bo zgenerirala tudi prazna funkcija splošne prekinitvene rutine.

```
/**
  * @brief This function handles TIM6 global interrupt, DAC1 and DAC3 channel underrun error interrupts.
  */
void TIM6_DAC_IRQHandler(void)
{
  /* USER CODE BEGIN TIM6_DAC_IRQn 0 */

  /* USER CODE END TIM6_DAC_IRQn 0 */

  /* USER CODE BEGIN TIM6_DAC_IRQn 1 */

  /* USER CODE END TIM6_DAC_IRQn 1 */
}
```

Mimogrede, opazite lahko, da to prekinitveno rutino lahko prožijo še druge periferne enote (digitalno-analogna pretvornika `DAC1` in `DAC3`).

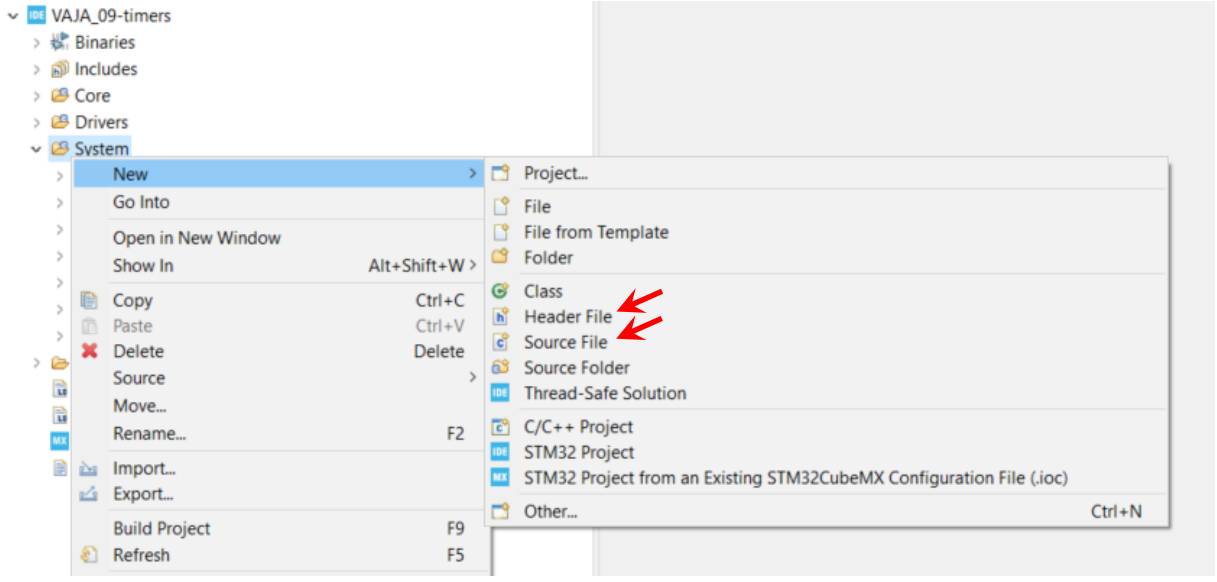
Preostali del priprave

10. ***V projektu <u>sami ustvarite datoteki modula</u>*** `periodic_services.c` ***in*** `periodic_services.h`.

    Tokrat boste modul v celoti implementirali sami. Pri tem *se boste zgledovali po modulih, ki smo jih implementirali tekom preteklih vaj.*

    Datoteki dodajte v projektno mapo znotraj podmape "`System`" na ustrezno mesto.

    Nove datoteke lahko v projektu ustvarite tako, da z desnim miškinim gumbom kliknete na mapo, kjer želite dodati datoteko, nato pa izberete vrsto datoteke (torej .c oziroma .h). Poglejte spodaj.



Opozorilo: imenom datotek morate nujno dodati še končnico (torej.c oziroma .h).

11. ***Preučite, katero LL knjižnico .h in katere nizko-nivojske funkcije bomo potrebovali za delo s časovnikom.***

    Potrebovali bomo sledečo nizko-nivojsko funkcionalnost LL knjižnice:

    a) *omogočiti števec*, da prične s štetjem (angl. enable counter),

    b) *omogočiti prekinitev* števca ob dogodku *preliva* (angl. update event interrupt),

    c) ugotoviti, *ali je omogočena prekinitev* časovnika ob dogodku preliva,

    d) preveriti, *ali je postavljena zastavica* za dogodek preliva časovnika,

    e) *pobrisati zastavico*, ki nakazuje dogodek preliva časovnika.

    Seznam LL funkcij, ki jih boste potrebovali, si dodajte v komentar v zglavni .h datoteki po vzoru prejšnjih vaj.

    Iz tega seznama nato ugotovite, katere parametre bo potrebno hraniti v *"handle" strukturi za upravljanje periodičnega izvajanja rutin*, če bomo želeli uporabljati te LL funkcije (bodite pozorni tudi na tip teh parametrov).

12. ***Modulu tipkovnice*** `kbd.c` ***na smiselno mesto dodajte funkcijo***

    ```
    KBD_demo_toggle_LEDs_if_buttons_pressed(),
    ```

    ki jo najdete v mapi "predloge".

    V zglavno datoteko modula `kbd.h` dodajte tudi prototip te funkcije, da jo bodo lahko uporabljati tudi preostali moduli. To funkcijo bomo namreč uporabili pri testiranju "avtomatskega" branja stanja tipkovnice.

13. ***Preučite postopek, na podlagi katerega bomo implementirali periodično izvajanje rutin s pomočjo prekinitev časovnika.***

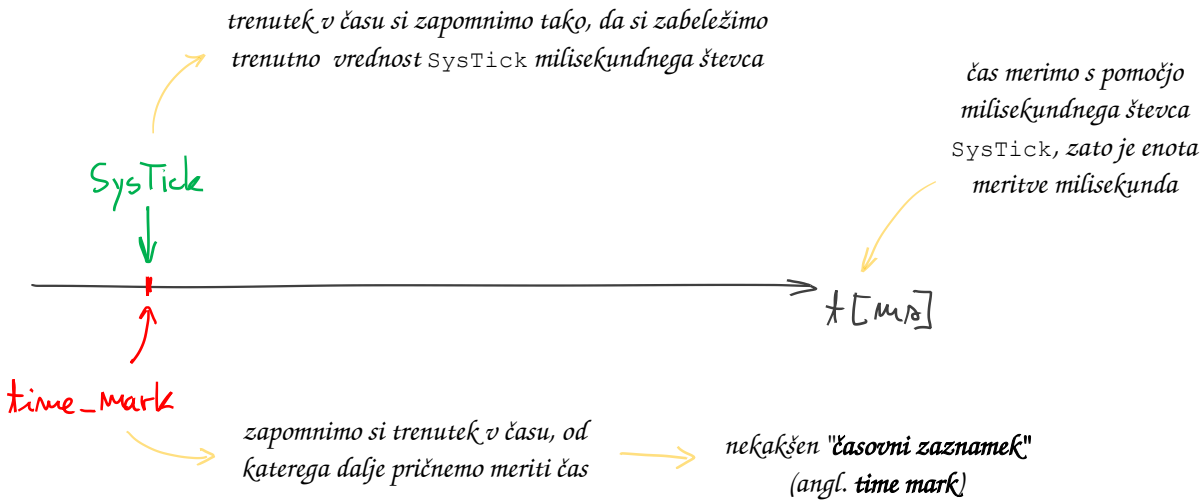
    Opis postopka najdete v poglavju spodaj.

Postopek za merjenje pretečenega časa

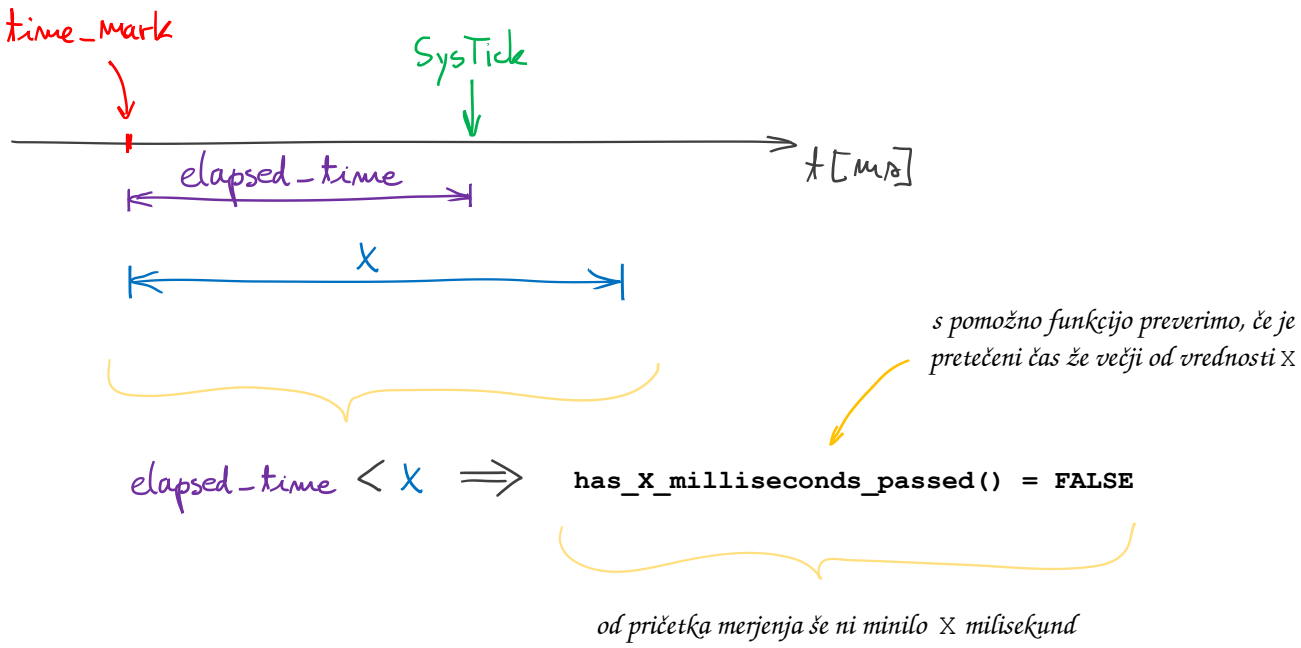
Pri programiranju vgrajenih sistemov je ključno, **da *znamo meriti pretečeni čas med dogodki* ter se tako *odzivati na dogodke v točno določenem časovnem zaporedju*.**

Pri merjenju časa si seveda pomagamo s *časovnikom ter pomožnimi funkcijami*, ki lahko implementirajo različne načine merjenje časa. V našem primeru bomo uporabili števec milisekund `SysTick` ter si pogledali implementacijo funkcionalnosti, ki spominja na *merjenje čas po principu ure štoparice*: s pritiskom na gumb sprožimo štoparico ter nato spremljamo, koliko časa je že menilo od tistega trenutka, ko smo pritisnili na gumb. Poglejte spodaj zaporedje skic, ki vam bodo pomagale razumeti, ključne korake pri implementaciji ure štoparice.

Prvi korak pri merjenju časa je ta, da se odločimo, od katerega trenutka dalje bomo merili pretečeni čas (z analogijo: kdaj bomo na štoparici pritisnili gumb). Poglejte idejo za implementacijo spodaj.



Ko pa imamo enkrat zabeležen časovni zaznamek `time_mark`, pa lahko preprosto *z odštevanjem določimo pretečeni čas od časovnega zaznamka* `elapsed_time`. Tako zelo enostavno dobimo funkcionalnost ure štoparice. Vendar pa je za programiranje vgrajenih sistemov smiselno to funkcionalnost še malenkost nadgraditi.

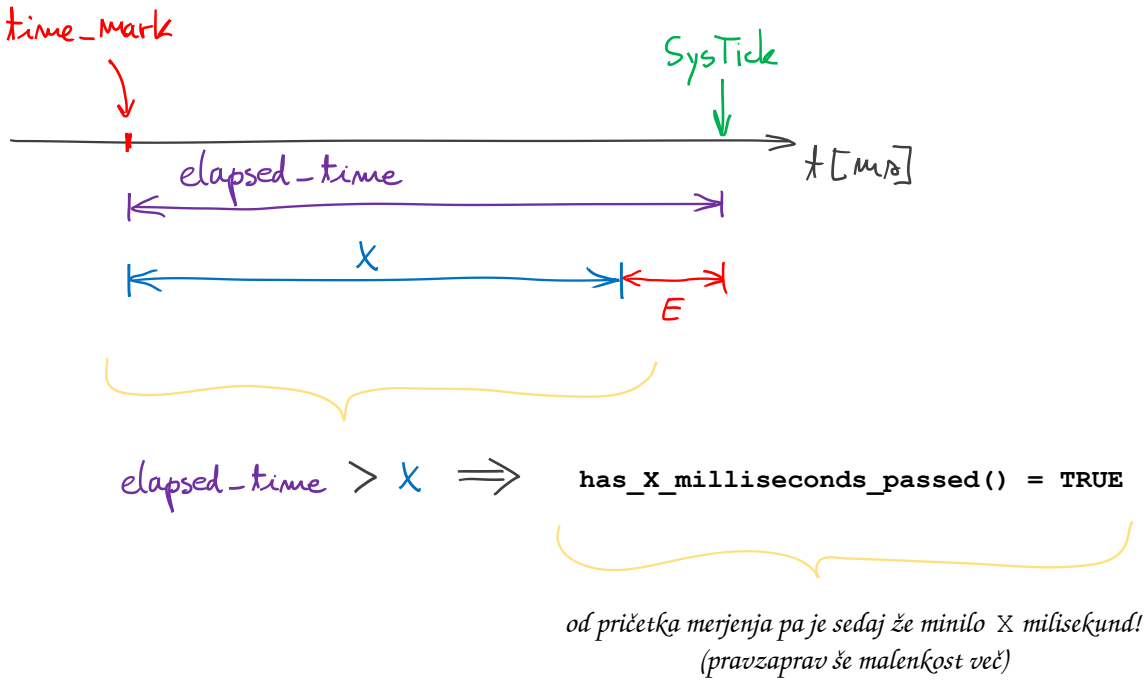


Običajno se želimo pri programiranju vgrajenih sistemov *odzivati na dogodke v točno določenem časovnem zaporedju* (npr. sistem vklopi alarm pet sekund po pritisku stikala). V tej luči je smiselno funkcionalnost štoparice nadgraditi tako, da ji ***dodamo funkcijo za preverjanje, če je od časovnega zaznamka že pretekel točno določen interval časa*** (v zgornjem primeru bi nas zanimalo, če je že preteklo pet sekund). V našem primeru bomo to funkcijo poimenovali

`has_X_milliseconds_passed()`.

kjer tisti "`X`" v imenu predstavlja dolžino časovnega intervala, ki jo podamo kot vhodni argument funkcije. Če sedaj še enkrat pogledate skico situacije zgoraj, lahko vidite, da bi z v zgornjem primeru ta funkcija vrnila logično vrednost `FALSE`, saj je pretečeni čas `elapsed_time` *manjši* od dolžine intervala "x".

Ko pa čas teče dalje, vrednost števca milisekund `SysTick` narašča. In ko bi naslednjič s pomočjo funkcije `has_X_milliseconds_passed()` preverili, ali je od začetka merjenja že pretekel interval dolžine "x", bi pa lahko naleteli na sledečo situacijo spodaj. V tem primeru pa je pretečeni čas `elapsed_time` *daljši* od dolžine intervala "x" in funkcija bi `has_X_milliseconds_passed()` vrnila logično vrednost `TRUE`.



Če dobro pogledate, opazite, da je v tej situaciji od začetka merjenja časa preteklo še celo nekoliko več časa kot le interval dolžine "x". To pa pomeni, da smo pri preverjanju pretečenega časa s funkcijo `has_X_milliseconds_passed()` pravzaprav pridelali napako (rdeči interval "E"). Izkaže se, da če funkcijo `has_X_milliseconds_passed()` uporabljamo dovolj pogosto, je ta napaka zanemarljiva in ne vpliva bistveno na obnašanje vgrajenega sistema (če nadaljujemo primer: če pri vklopu alarma zamudimo 10 milisekund, to ne predstavlja problema).

Vidite, kako lahko ***funkcijo*** `has_X_milliseconds_passed()` ***uporabite, da dva dogodka izvedete v točno določenem časovnem razmiku***.

Postopek za periodično merjenje pretečenega časa

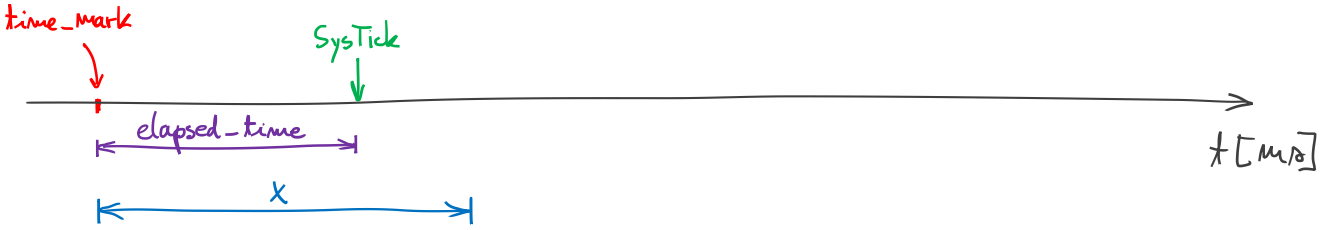
Vprašanje: kaj pa če bi želeli ***dogodke izvajati periodično po preteku točno določenega intervala časa***? (Za primer: prižiganje in ugašanje LEDice z intervalom ene sekunde.)

Ideja: še vedno bi lahko uporabili idejo zgornje funkcije `has_X_milliseconds_passed()`, le da bi bilo sedaj potrebno vsakič, ko bi pretekel interval dolžine "x" *smiselno ponastaviti časovni zaznamek* `time_mark`. To idejo bomo mi implementirali v pomožni funkciji, ki jo bomo poimenovali

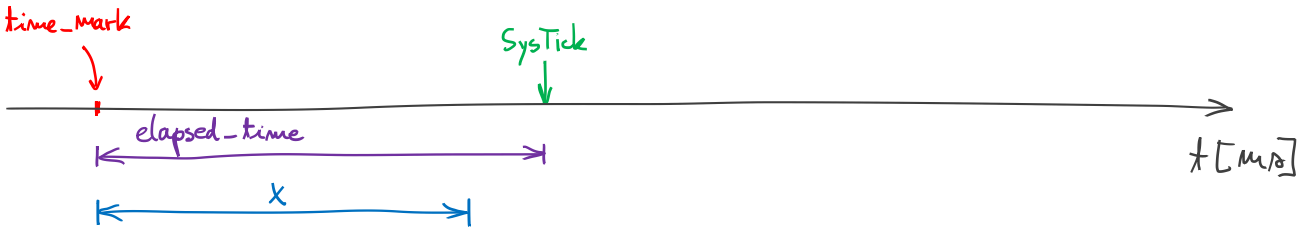
```
has_another_X_milliseconds_passed().
```

Pokažimo idejo te funkcije s pomočjo skic spodaj.

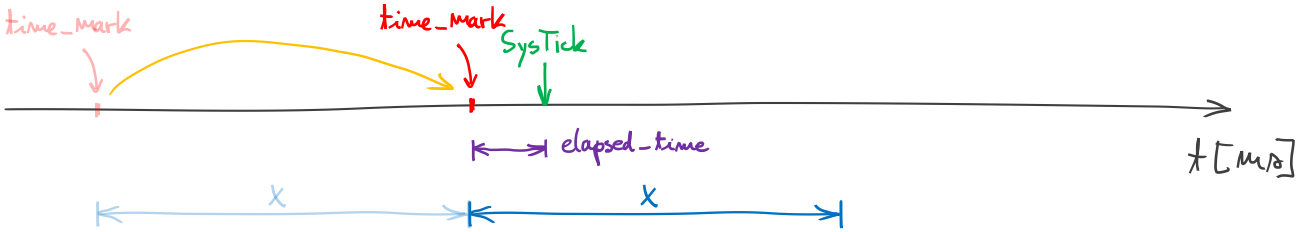
V prvem koraku postavimo časovni zaznamek `time_mark` in pričnemo z merjenjem pretečenega časa `elapsed_time`. Z dovolj pogostim preverjanjem ugotavljamo, ali je že pretekel interval dolžine "x". V spodnji situaciji še ni. V takem primeru bi funkcija `has_another_X_milliseconds_passed()` vrnila logično vrednost `FALSE`.



Pri preverjanju pretečenega časa bi lahko kasneje naleteli na spodnjo situacijo: pretečeni čas bi bil večji od dolžine intervala "x". Takrat bi funkcija `has_another_X_milliseconds_passed()` vrnila logično vrednost `TRUE`.



Ker pa želimo implementirati *periodično merjenje pretečenega časa* med dogodki, pa mora sedaj funkcija `has_another_X_milliseconds_passed()` ponastaviti časovni zaznamek `time_mark`! Postaviti ga mora za dolžino intervala "x" naprej. Postaviti ga mora na konec *prejšnjega* intervala dolžine "x" (prosojna modra barva). Poglejte situacijo spodaj. Na ta način sedaj že merimo pretečeni čas v *naslednjem* intervalu dolžine "x" (polno modra barva).

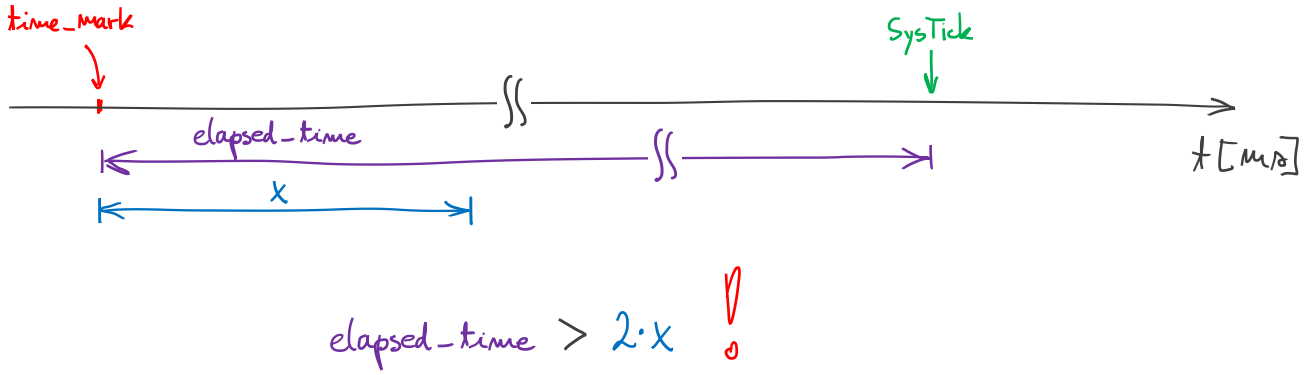


V tem se funkcija `has_another_X_milliseconds_passed()` bistveno razlikuje od funkcije `has_X_milliseconds_passed()`:

> da poskrbi za ustrezen premik časovnega zaznamka naprej in tako omogoči, da po preteku intervala dolžine "x" pričnemo s ***<u>ponovnim</u>*** merjenjem pretečenega časa in preverjanjem, ali je ***<u>spet</u>*** minil interval dolžine "x".

Poseben primer premika časovnega zaznamka

Pri premikanju časovnega zaznamka `time_mark` naprej pa je potrebno upoštevati izjemno situacijo, ki se lahko pripeti, če s funkcijo `has_another_X_milliseconds_passed()` ne uspemo preverjati pretečenega časa dovolj pogosto. Zgodi se namreč lahko, da pretečeni čas `elapsed_time` postane tako velik, da ***ni minil le en sam interval dolžine "x" temveč kar <u>dva ali več</u> intervalov dolžine "x"***! Poglejte situacijo spodaj.



V takem primeru pa je smiselno, da časovni zaznamek ne premikamo naprej za dolžino intervala "x", temveč ga *kar ponovno nastavimo na trenutno vrednost časa*. Poglejte idejo spodaj.

Tako smo pravzaprav pričeli s ponovnim merjenjem pretečenega časa `elapsed_time` od začetka. Za dogodke, ki smo jih zamudili, ne moremo storiti nič. Lahko pa pričnemo s takojšnjim ponovnim merjenjem časa. Taka implementacija očitno ne zagotavlja precizne periode med periodičnimi dogodki, a za veliko *nekritičnih rutin*, ki se morajo izvajati z določeno periodo, je taka implementacija več kot primerna.

Ideja periodičnega izvajanja rutin s pomočjo prekinitev časovnika

Idejo za implementacijo periodičnega izvajanja rutin s pomočjo časovnika si lahko ogledate na spodnjem diagramu.