Beágyazott és Ambiens Rendszerek

5. gyakorlat tematikája

Futási idő mérése, időmérés

A gyakorlat során a következő témakörökkel ismerkedünk meg:

- futási idő mérésének technikája,
- néhány tipikus utasítás erőforrásigénye,
- időmérés általában.

1. Futási idő mérése

Háttérismeretek

A következőkben megismerkedünk a futási idő mérésének módszerével és néhány tipikus utasítás futási idejét is megvizsgáljuk.

A futási idő mérésének több módszere ismert (lásd előadás), a gyakorlat során a mikrokontroller egy beépített speciális időzítőjét/számlálóját fogjuk használni. Ezt a számlálót a processzorban a Debug Interface-en belül a Data Watch point and Trace (DWT) nevű egység tartalmazza. Megoldható a feladat természetesen más számláló/időzítő egységgel is, de jelen esetben ezen számláló használata a egyszerűbb. A számláló 32 bites, tehát a 14 MHz-es default órajel esetén $T_{max} = \frac{1}{14\ \mathrm{MHz}} 2^{32} = 306\ \mathrm{sec} \approx 5\ \mathrm{min}$ hosszúságú futási időt is tudunk mérni. Itt jegyezzük meg, hogy a futási idő mérése igazából futási ciklusidő mérése, tehát azt fogjuk meghatározni, hogy hány órajelciklus alatt fut le egy adott kódrészlet, ami viszont az órajel-frekvencia ismeretében könnyen átváltható idővé is.

A futási idő mérésére felhasznált számláló neve: Cycle Count Register. Ez a regiszter a processzor indulásakor nulláról indul, és minden órajelütemben eggyel nő az értéke. A számláló regisztert a következő módon érhetjük el:

```
DWT->CYCCNT
```

Egy lehetséges megoldás a mérésre:

```
runTime = DWT->CYCCNT;
    Ide jön a programkód, aminek a futási idejét mérni szeretnénk.
runTime = DWT->CYCCNT - runTime - COMP_CONST;
```

A fenti kód lefutását követően a runTime változó fogja tartalmazni az órajelekben mért futási időt. A `COMP_CONST` helyére egy olyan konstanst írjunk, aminek a segítségével a fenti kifejezés nulla értéket ad, ha nincsen semmilyen programkód megadva. Ezzel korrigáljuk a számítással és a regiszterolvasással eltöltött időt, amit belemérünk a futási időbe, de egyébként nem része a programnak.

Feladatok

1. Hozzunk létre egy üres projektet, és a futási idő méréséhez megadott programkódban határozzuk meg a `COMP_CONST` konstans értékét.
2. Mérjük meg a következő három művelet futási idejét három féle adattípus esetén a következő táblázatnak megfelelően:

-O0	int32_t	float	double
összeadás			
szorzás			
osztás			

Mintakód:

```
int32_t a=300, b=20, c=0;
runTime = DWT->CYCCNT;
      c=a+b;
runTime = DWT->CYCCNT - runTime - COMP_CONST;
```

Honnan látható, hogy a processzor rendelkezik osztást támogató utasítással?

3. Kikapcsolt –O0 szintű optimalizáció mellett mérd meg, hogy a következő típuskonverziók hány órajel alatt futnak le.

Mintakód:

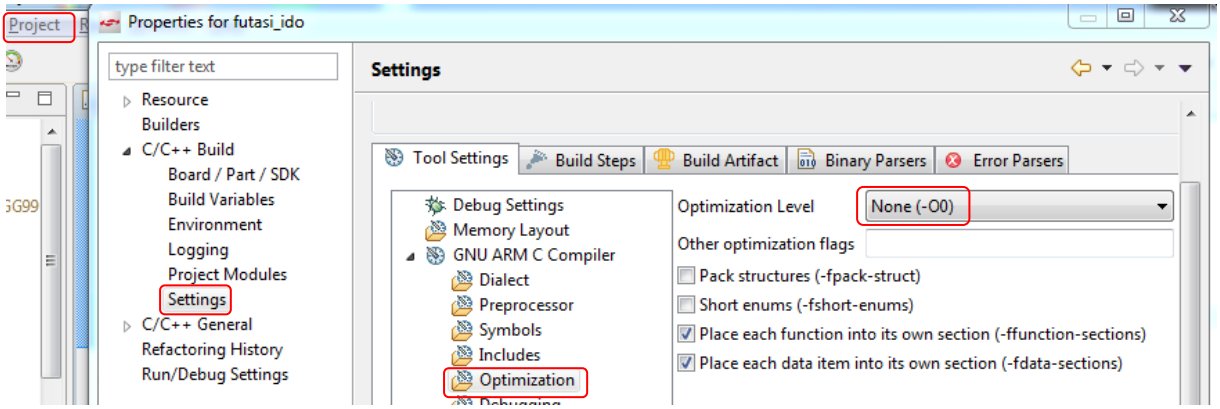
```
a_float = (float) a_int;
```

Megjegyezzük, hogy a típuskonverziót (kasztolást) akkor is elvégzi a fordító, ha azt explicite nem adjuk meg, tehát a következő sor is tartalmaz egy implicit típuskonverziót, ugyanaz, mintha kiírnánk a **`(float) a_int`** konverziót:

```
a_float = a_int;
```

forrás \ cél	int32_t	float	double
int32_t			
float			
double			

4. Mérjünk meg újra az egyik típuskonverzió futási idejét –O3 optimalizációs szint esetén is: Project → Properties → C/C++ Build →Settings → Optimization → Optimize most (-O3)



Figyeljük meg, hogy a projekt újrafordítását követően milyen eredményt kapunk a futási időre (pl. double osztás esetén).

Ebben az esetben furcsaságokat figyelhetünk meg, mert a fordító eltávolítja a nem használt változókat, így a mért programkód gyakorlatilag eltűnhet. Érdemes tehát a műveletek ki- és bemeneti változóit, illetve a futási idő mérésére szolgáló változót volatile változóként deklarálni, így a fordító nem optimalizálja ki azokat.

5. Bekapcsolt –O3 szintű optimalizáció mellett deklarálj két darab N méretű int32_t típusú tömböt (volatile legyen), és számítsd ki a szorzatok összegét: $s = \sum_{i=0}^{N-1} A[i] * B[i]$. Jegyezd fel a futási időket H=15…20 méretű tömbök esetére.

N	**15**	**16**	**17**	**18**	**19**	**20**
órajelciklus (–O3)						
órajelciklus (–O0)						

Vizsgáld meg, hogy mennyire egyenletes a változás a ciklus hosszának növelésével. Figyeld meg a fordító loop unroll[1] funkcióját a dissasemblált kódban.
N=15 esetére nézd meg, hogy kikapcsolt optimalizáció esetén mekkora a futási idő.

[1] loop unroll: a for ciklusok kifejtése úgy, hogy az N-szer végrehajtandó kódot N-szer megismétli a fordító. Növeli a kód méretét, de általában gyorsabb, mert nem kell a ciklusváltozót kezelni és ugró utasítást végrehajtani. Probléma lehet még, hogy az utasítás cache nem használható ki úgy, mint egy ciklus esetén.

Mintakód:

```
sum=0;
for (ii=0; ii<N; ii++) {

      sum += A[ii]*B[ii];

}
```

Kiegészítő feladatok

6. Bekapcsolt –O3 szintű optimalizáció mellett deklarálj két darab N=100 méretű int32_t típusú tömböt (volatile legyen), és számítsd ki a szorzatok összegét három módszerrel:

$s = \sum_{i=0}^{N-1} A[i] * B[i]$.

$s = \sum_{i=0}^{N/2-1} (A[2i] * B[2i] + A[2i+1] * B[2i+1])$.

$s = \sum_{i=0}^{N/4-1} (A[4i] * B[4i] + A[4i+1] * B[4i+1] + A[4i+2] * B[4i+2] + A[4i+3] * B[4i+3])$.

Ez a loop unroll egy speciális esete, amikor a programozó „kézzel” teríti ki a hurkot, de csak kisebb részletekben. Tehát például négyesével lépkedve a tömbben a négyes blokkokat egyszerre összeadjuk. Ebben az esetben a ciklusváltozó kezelése negyedannyi időt igényel.

A megvalósítás a következő lehet (pl. 2-es unroll esetén):

```
sum=0;
for (ii=0; ii<N; ii+=2) {
      sum += A[ii]*B[ii]+ A[ii+1]*B[ii+1];
}
```

A ciklusban tehát kettesével lépkedünk, és vesszük az i és (i+1)-edik elemet.

Végezd ez a mérést N=1000-re is.

fokszám \ unroll	1 (sum: i)	2 (sum: i, i+1)	4 (sum: i, i+1, i+2, i+3)
N=100			
N=1000			

7. Bekapcsolt –O3 szintű optimalizáció mellett deklarálj két darab N=100 méretű int32_t típusú tömböt (volatile legyen), és számítsd ki a szorzatok összegét: $s = \sum_{i=0}^{N-1} A[i] * B[i]$.
Hozzuk létre a következő függvényt:

```
int32_t mult(int32_t a, int32_t b){
      return a*b;
}
```

A szorzást végezd el háromféle módon:

- A*B
- A **mult** függvény használatával: mult(A,B).

- Úgy, hogy a **mult** függvény elé betesszük a következő függvényattribútumot: `__attribute__((noinline)) int32_t mult(int32_t a, int32_t b)`

Jegyezd fel a futási időket.

fokszám \ unroll	A*B	mult(A,B)	noinline mult
N=100			

Látható, hogy optimalizáció esetén a fordító automatikusan is felismerheti, hogy melyik függvényt érdemes inline függvényként használni.

2. Időmérés, időzítő konfigurálása

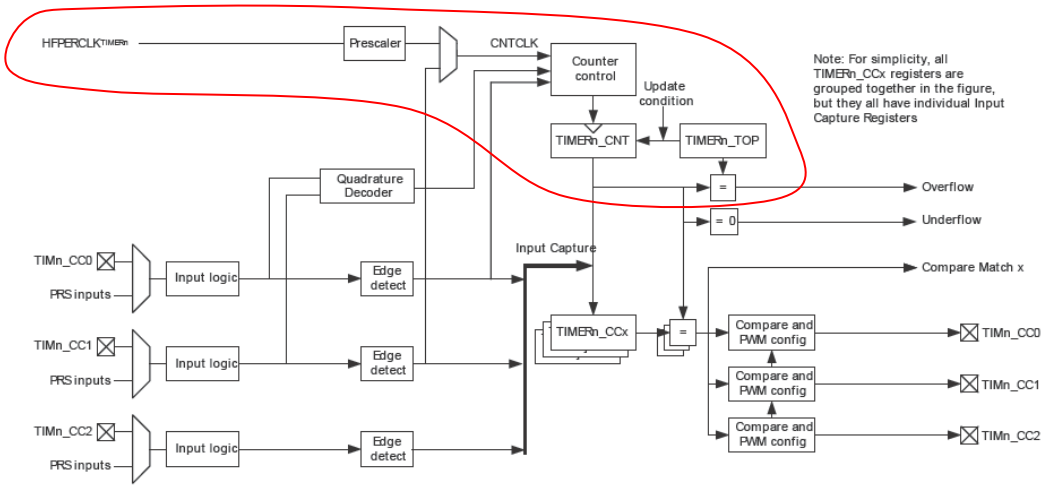
A következőkben az időmérés általános módszerével, az időzítők használatával ismerkedünk meg.

A mikrokontrollerek timer egységeinek általában két fő üzemmódja van:

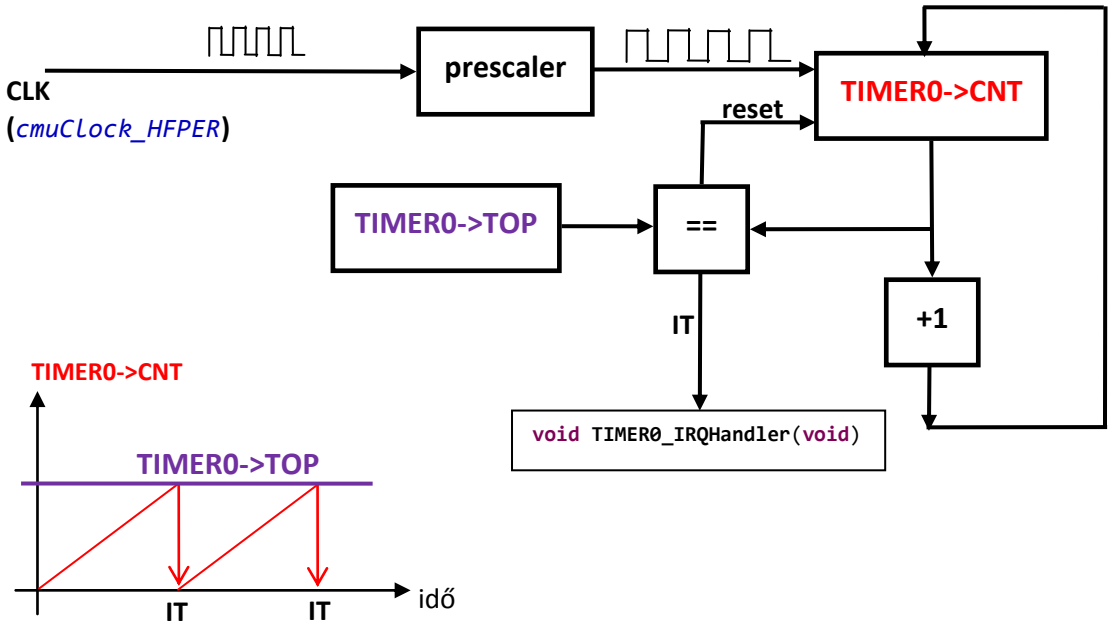
- Időzítő: valamilyen belső órajelforrás segítségével időt mérünk, megszakításokat generálunk.
- Számláló: általában valamilyen külső forrásból beérkező impulzusokat számláljuk.

Ezeken belül még több aleset is elképzelhető.

A gyakorlaton használt mikrokontroller timer egysége a következő módon épül föl (a Timer0 egységet fogjuk használni):

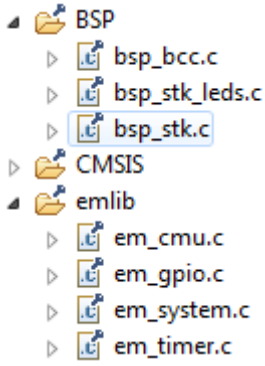


Mi az alábbi egyszerűsített ábrának megfelelő időzítő üzemmódban fogjuk használni (a fenti ábra pirossal bekeretezett részének magyarázata):



A Timer egyszerű időzítő üzemmódjában a bemenő órajel minden ütemére a CNT regiszter értéke eggyel nő mindaddig, amíg a TOP regiszterben tárolt értéket el nem éri. Ekkor a CNT számláló nullázódik, és a megfelelő megszakításvonalon megszakítást generál (`TIMER_IF_OF`).

Az időzítő kezeléséhez hozzá kell adni a projekthez az em_timer.c és em_cmu.c fájlokat[2], és include-olni kell a em_cmu.h és em_timer.h fájlokat. A LED-ek kezeléséhez a projekthez hozzá kell adni a bsp_bcc.c, bsp_stk_leds.c, bsp_stk.c és em_gpio.c fájlokat[3], és include-oljuk a bsp.h fájlt.




```
#include "em_device.h"
#include "em_chip.h"
#include "em_cmu.h"
#include "em_timer.h"
#include "bsp.h"
```

Az időzítőt könyvtári függvényei segítségével konfiguráljuk a következő módon:

- a periféria órajel osztójának beállítása
- időzítő órajelének engedélyezése
- létrehozzuk az inicializációhoz szükséges paraméterstruktúrát
    - a prescalert átállítjuk a megfelelő értékre
- reseteljük az időzítőt
- beállítjuk a TOP értéket
- töröljük az esetleges függő megszakítást
- Engedélyezzük a megszakítást
    - Engedélyezzük az időzítő perifériánál
    - Engedélyezzük a központi megszakítás-kezelőnél a Timer megszakítást (NVIC)

A fenti inicializációs folyamatot az alábbi programkód valósítja meg.

```
// a periféria órajel osztójának beállítása
CMU_ClockDivSet(cmuClock_HFPER, cmuClkDiv_1);

// ******************************
// *    TIMER inicializálása    *
// ******************************

// időzítő órajelének engedélyezése
CMU_ClockEnable(cmuClock_TIMER0, true);

// létrehozzuk az inicializációhoz szükséges paraméterstruktúrát
TIMER_Init_TypeDef TIMER0_init = TIMER_INIT_DEFAULT;
// a prescaler-t átállítjuk
```

[2] elérési útvonal: SimplicityStudio\developer\sdks\gecko_sdk_suite\v1.1\platform\emlib\src\
[3] elérési útvonal: SimplicityStudio\developer\sdks\gecko_sdk_suite\v1.1\hardware\kit\common\bsp\

```
  TIMER0_init.prescale = timerPrescale1; // timerPrescale1...timerPrescale1024
  // inicializálás a paraméterstruktúrával
  //void TIMER_Init(TIMER_TypeDef *timer, const TIMER_Init_TypeDef *init);
  TIMER_Init(TIMER0, &TIMER0_init);

  // reseteljük a számlálót
  TIMER_CounterSet(TIMER0, 0); //

  // beállítjuk a TOP értéket
  //__STATIC_INLINE void TIMER_TopSet(TIMER_TypeDef *timer, uint32_t val)
  TIMER_TopSet(TIMER0, IDE_KELL_ÍRNI_A_TOP_ÉRTÉKET); // 14MHz/presc/TOP

  // töröljük az esetleges függő megszakításokat
  //__STATIC_INLINE void TIMER_IntClear(TIMER_TypeDef *timer, uint32_t flags);
  TIMER_IntClear(TIMER0, TIMER_IF_OF);

  // Timer IT engedélyezése
  //TIMER_IntEnable(TIMER_TypeDef *timer, uint32_t flags);
  TIMER_IntEnable(TIMER0, TIMER_IF_OF);

  // Timer IT engedélyezése az NVIC-ben
  NVIC_EnableIRQ(TIMER0_IRQn);


  // ******************************
  // *     LED inicializálása     *
  // ******************************
  BSP_LedsInit();
```

A megszakítás lekezeléséhez az alábbi függvényt kell implementálni a programkódban:

```
void TIMER0_IRQHandler(void){
      BSP_LedToggle(0);
      TIMER_IntClear(TIMER0, TIMER_IF_OF); //TIMER flag törlése
}
```

A függvénynevek beszédesek, a fenti programkód értelmezését a hallgatókra bízzuk.

A timer inicializálásához használt default értékek.

```
#define TIMER_INIT_DEFAULT
{                                                                    \
    1,                 /* Enable timer when init complete. */        \
    0,                 /* Stop counter during debug halt. */         \
    timerPrescale1,        /* No prescaling. */                      \
    timerClkSelHFPerClk,  /* Select HFPER clock. */                  \
    0,                 /* Not 2x count mode. */                      \
    0,                 /* No ATI. */                                 \
    timerInputActionNone, /* No action on falling input edge. */     \
    timerInputActionNone, /* No action on rising input edge. */      \
    timerModeUp,           /* Up-counting. */                        \
    0,                 /* Do not clear DMA requests when DMA channel is active. */ \
    0,                 /* Select X2 quadrature decode mode (if used). */ \
    0,                 /* Disable one shot. */                       \
    0                  /* Not started/stopped/reloaded by other timers. */ \
  }
```

Feladatok

1. A fent leírtak alapján inicializáljuk a timer0 perifériát.
2. Állítsuk be a prescaler-t és az időzítő TOP értékét úgy, hogy 1 másodpercenként generáljunk megszakításokat (a megszakításrutinban váltogassuk az egyik LED állapotát).
    a. Valójában mekkora a beállított frekvencia névleges értéke?
3. Állítsuk be a prescaler-t 1-es értékűre, és az időzítő TOP értékét 14000-re. A megszakításrutinban szoftveresen oldjuk meg, hogy 1 másodpercenként váltson a LED állapotot.