Beágyazott és Ambiens Rendszerek

7. gyakorlat tematikája

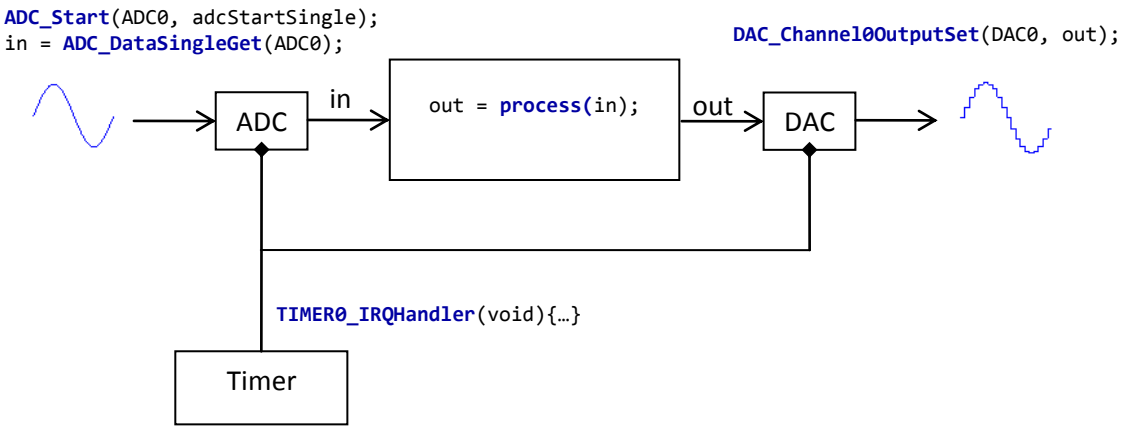
Adatfeldolgozási minták a gyakorlatban

A gyakorlat során a következő témakörökkel ismerkedünk meg:

- CMSIS adatfeldolgozást segítő moduljainak használata
- Szinuszos jel generálása
- Zaj generálása
- Szűrés műveletek implementációja, szűrés hatásának vizsgálata
- Fix pontos tört számábrázolás használata

1. Rendszer felépítése

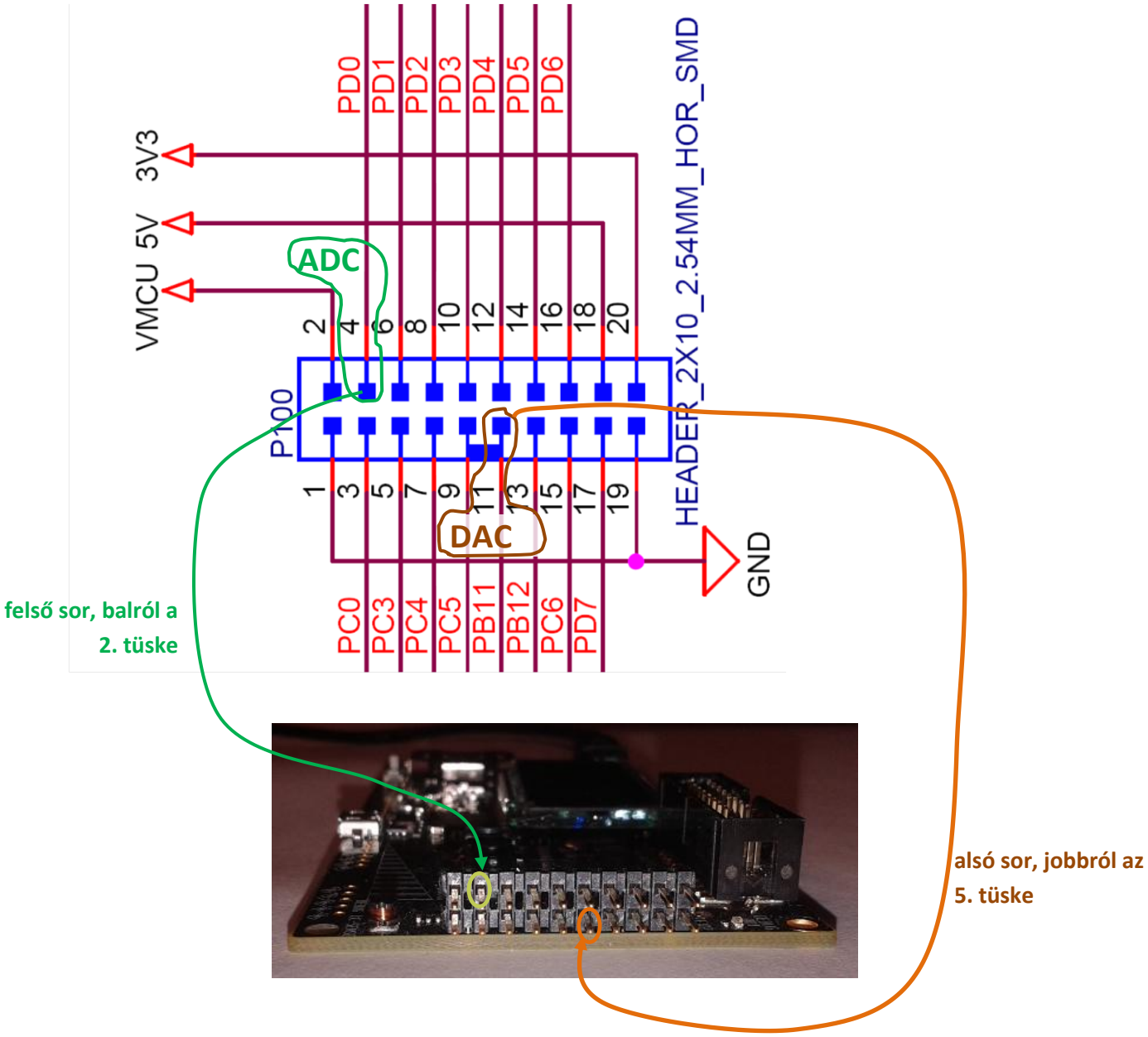
Munkánk során az alábbi ábrán látható, az előző gyakorlaton létrehozott keretrendszert fogjuk használni. A mintafájl rendelkezésre áll a honlapon, illetve mellékletként az útmutató végén.



A mikrokontroller által generált kimeneteket visszavezetjük a bemenetre, így végig valódi analóg jelekkel tudunk dolgozni. A ki- és bemeneti lábak az alábbi dokumentáció szerint a PB11 (DAC0 kimenet) és a PD0 (ADC_CH0 bemenet) lábakon találhatók:

Alternate	LOCATION							
Functionality	0	1	2	3	4	5	6	Description
ADC0_CH0	PD0							Analog to digital converter ADC0, input channel number 0.
DAC0_OUT0 / OPAMP_OUT0	PB11							Digital to Analog Converter DAC0_OUT0 / OPAMP output channel number 0.

A mikrokontroller bővítő csatlakozóján az alábbiaknak megfelelően azonosíthatók a lábak:



A programunkat a `process` függvényben kell megvalósítani. Itt kapjuk meg bemeneti paraméterként az AD átalakítóból származó jeleket, és a visszatérési értékként átadott változó kerül ki a DA kimenetére.

```
// ****************************************
// * process data                         *
// ****************************************
uint32_t process(uint32_t in){
        uint32_t out;

        //------------------
        // save input data
        inputData_idx = (inputData_idx+1)&(N-1);
        inputData[inputData_idx] = in;
        inputData_filt[inputData_idx] = in;
        //------------------
```

```
        return out;
}
```

A process függvényben található két tömb, amelyekbe a bemeneti adatokat, illetve később a feldolgozott adatokat mentjük (inputData és inputData_filt nevű tömbök).

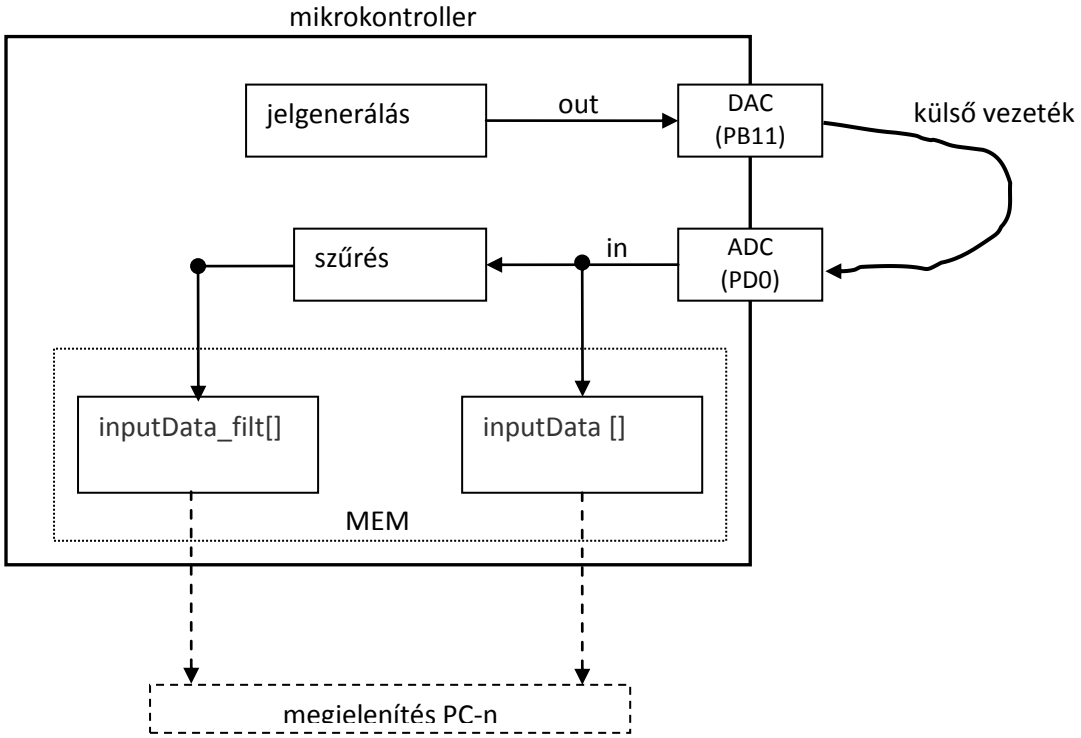
A main függvényben a következő végtelen ciklus található:

```
  /* Infinite loop */
  while (1) {

          if (((N-1)==inputData_idx)&&getBP0()){
                  BSP_LedSet(1); //set here breakpoint
          } else{
                  BSP_LedClear(1);
          }
  }
```

A kódrészlet lényege, hogy amikor az adatokat tároló tömb végére értünk, és gombnyomás van, akkor felvillan az 1-es számú LED. Amennyiben erre a sorra beállítunk egy breakpontot, akkor mindig olyan állapotban vizsgálhatjuk az adatbuffert, amikor teljesen megtelt, és nem látjuk, hogy az új adatok felülírják a régieket cirkuláris buffer jelleggel.

A teljes működési mód blokkvázlata:



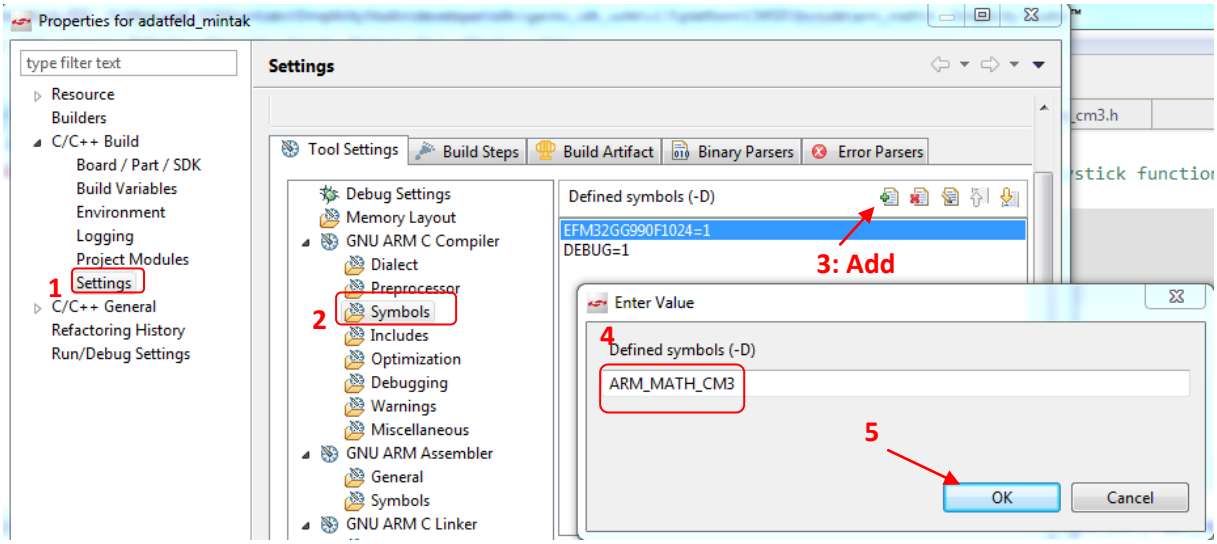
2. Alapprojekt létrehozása

A honlapon található process_base.c fájl tartalmazza a kiinduló kódot.

Hozzunk létre egy üres projektet, és a main.c fájlba másoljuk be a process_base.c fájl tartalmát.

A munkánk során az ARM CMSIS matematikai könyvtárát használjuk. Ennek jellegzetessége, hogy meg kell adni, hogy milyen típusú processzormag található a mikrokontrollerben. Bizonyos műveletek esetén ennek megfelelő optimalizációs módszereket használt a gyártó.

A processzor típusának megadása úgy történik, hogy a projektre a bal oldali „Project Explorer” ablakban jobb gombbal kell kattintani, ki kell választani a Properties menüpontot, majd az alábbi ábrán mutatott sorrendnek megfelelően be kell írni projektszinten defieniált változóként az „ARM_MATH_CM3” konstanst.



A projekt létrehozása során már a kezdetben érdemes a következő c fájlokat hozzáadni a projekthez, illetve header fájlokat include-olni: em_adc.c, em_cmu.c, em_core.c, em_dac.c em_gpio.c, em_system.c és em_timer.c fájlokat[1].A LED-ek kezeléséhez a projekthez hozzá kell adni a bsp_bcc.c, bsp_stk_leds.c, bsp_stk.c[2]. A header fájlokat már tartalmazza a program.

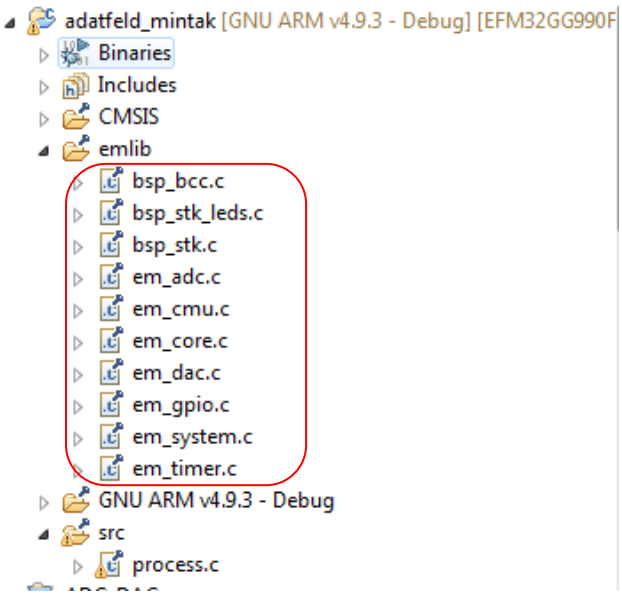
A munka gyorsítása érdekében a következő gyors linkre kattintva elérhetők a C fájlokat tartalmazó könyvtárak:

c:\SiliconLabs\SimplicityStudio\v4\developer\sdks\gecko_sdk_suite\v2.3\platform\emlib\src\

c:\SiliconLabs\SimplicityStudio\v4\developer\sdks\gecko_sdk_suite\v2.3\hardware\kit\common\bsp\

[1] elérési útvonal: SimplicityStudio\developer\sdks\gecko_sdk_suite\v1.1\platform\emlib\src\

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



3. Jelfeldolgozási műveleteket segítő függvények hozzáadása

Az általános matematikai függvényhívásokhoz szükségünk van az `arm_math.h` fájlra. A trigonometrikus függvények használatához szükséges az `arm_common_tables.h` fájl. A véletlenszám-generátorként használt rand() függvény eléréséhez szükségünk van az `stdlib.h` fájlra. A következő include-okat még adjuk hozzá a projekthez:

```
#include "arm_math.h"
#include "arm_common_tables.h"
#include <stdlib.h>
```

A szinusz függvény használatához a következő két fájlt kell behúzni a projekt alá tartozó CMSIS könyvtárba:

arm_sin_f32.c

arm_sin_q15.c

A fájlok az alábbi alkönyvtárban érhetők el (a telepítési könyvtárhoz viszonyítva verzió szám függvényében)

SimplicityStudio\developer\sdks\gecko_sdk_suite\v2.3\platform\CMSIS\DSP_Lib\Source\FastMathFunctions\

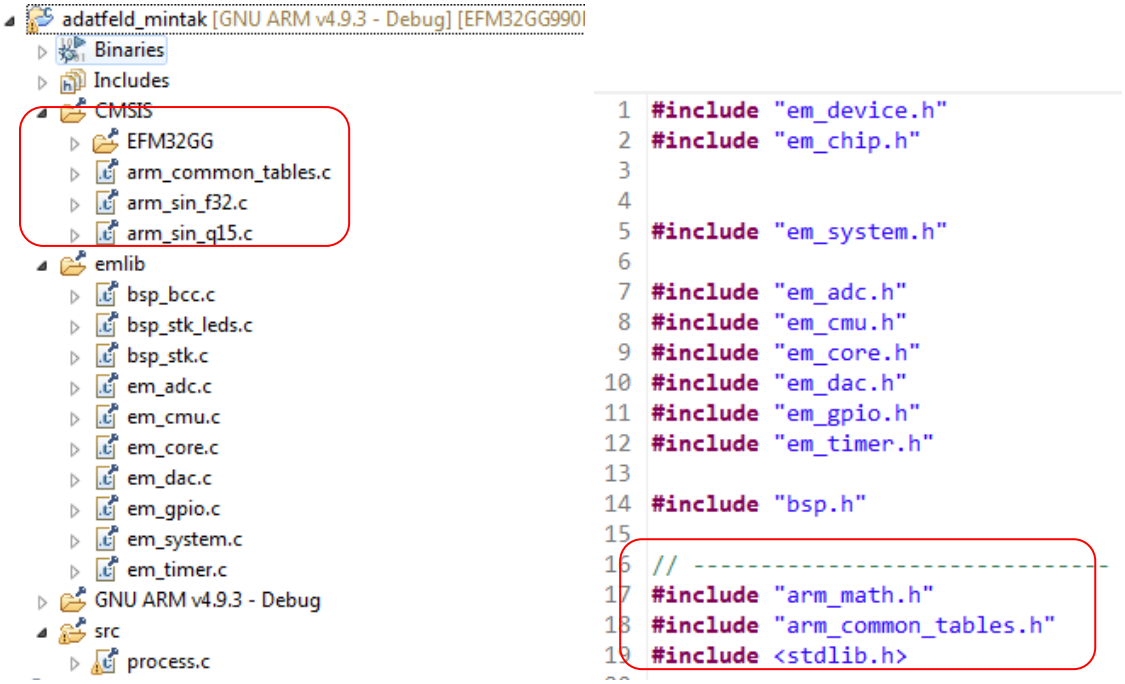
Ezen felül a trigonometrikus számításokat segítő táblázatok a következő fájlban találhatók, amelyet szintén be kell húzni a CMSIS alkönyvtárunkba:

arm_common_tables.c

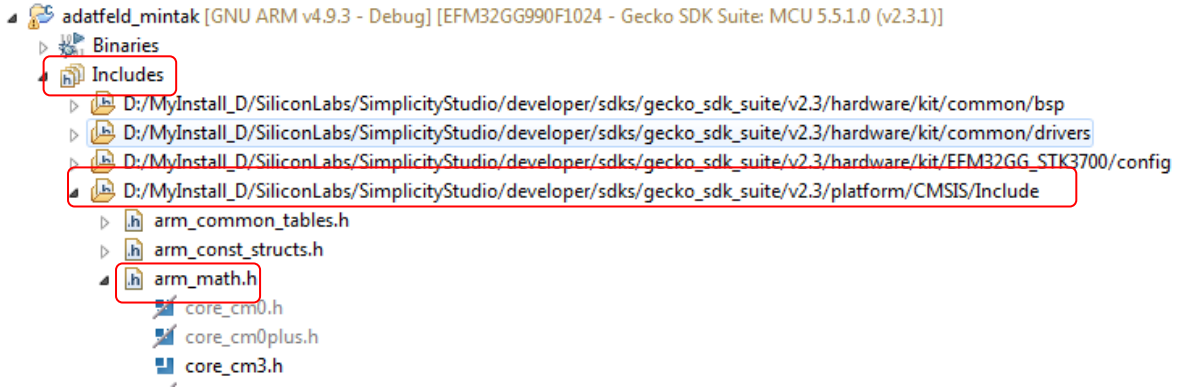
A fájl a következő helyen található

\SimplicityStudio\developer\sdks\gecko_sdk_suite\v2.3\platform\CMSIS\DSP_Lib\Source\CommonTables\

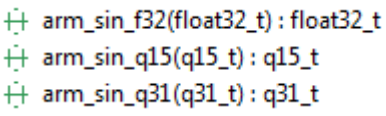
Az összes c és h fájl hozzáadását követően a projekt a következő módon néz ki:



A rendelkezésre álló matematikai függvényekről információt szerezhetünk, amennyiben az include-olt fájlok közül kikeressük az arm_math.h fájlt a következő helyen, és lenyitva a hozzá tartozó kis háromszöget láthatjuk a benne megadott függvények listáját:



Itt rengeteg belső változó és konstans között megtaláljuk például a számunkra szükséges trigonometrikus függvényeket:




```
/**
 * @brief  Fast approximation to the trigonometric sine function for
floating-point data.
 * @param[in] x input value in radians.
 * @return  sin(x).
 */

float32_t arm_sin_f32(
  float32_t x)
{
```

```
/**
 * @brief Fast approximation to the trigonometric sine function for Q15
data.
 * @param[in] x Scaled input value in radians.
 * @return  sin(x).
 *
 * The Q15 input value is in the range [0 +0.9999] and is mapped to a
radian value in the range [0 2*pi).
 */

q15_t arm_sin_q15(
  q15_t x)
```

A fenti módszeren kívül a könyvtárrendszer dokumentációja megtalálható az alábbi elérési útvonalon:
\SiliconLabs\SimplicityStudio\developer\sdks\gecko_sdk_suite\v2.3\platform\CMSIS\Documentation\

4. Szinusz jel generálása lebegő pontos számokkal

Egy szinuszos jel időfüggvénye:

$$x = A \cdot \sin(2\pi f t) + C = A \cdot \sin(\varphi) + C$$

ahol a fázis:

$$\varphi = 2\pi f t$$

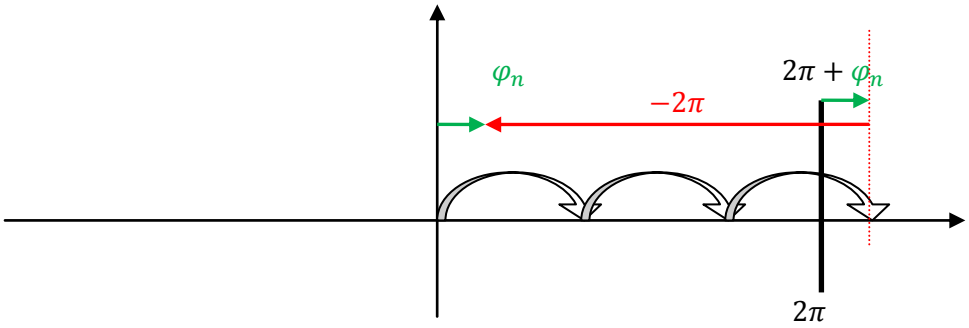
A mi esetünkben ezt a fázist iteratívan állítjuk elő minden egyes mintavételi időpontban:

$$\varphi_{n+1} = \varphi_n + 2\pi f T_s$$

A függvény radiánban várja a bemeneti értéket. Célszerű $[0 \dots 2\pi]$ intervallumban tartani az eredményt, így minden egyes ütemben elvégezzük az alábbi korrekciót:

$$ha\ (\varphi_{n+1} > 2\pi) \Rightarrow \varphi_{n+1} = \varphi_{n+1} - 2\pi$$

A fenti módszert az alábbi ábra illusztrálja:



Mindez programkódban:

```
phi = phi + 2*PI*fj*Ts;
if (phi>(2*PI)){
      phi = phi - 2*PI;
}

out_tmp =  DC_szint + Amplitudo*arm_sin_f32(phi);
```

Ahhoz, hogy a 12 bites AD átalakítón megfelelően jelenjen meg a jel, meg kell szoroznunk 4096-tal, illetve uint32_t típusúvá kell alakítani:

```
out = (uint32_t) (4096*out_tmp);
```

5. Egyenletes eloszlású fehér zaj generálása lebegő pontos számokkal

Az stdlib.h-ban található rand() függvény integer számokkal tér vissza a $[0…2^{31}]$ tartományban. Ahhoz, hogy az egész számot, lebegő pontos [0…1] tartományba transzformáljuk, le kell osztani a maximális értékkel.

Az így kapott szám középértéke 0.5 lesz. Nulla középértékű és ±0.5 tartományban mozgó (tehát egységnyi csúcstól-csúcs értékű) jelet úgy kapunk, ha 0.5-et kivonunk a számból. Összességében a következő programkód egy adott DC szintű, adott csúcstól-csúcsig vett értékű jelet eredményez.

```
out_tmp = DC_ertek + peak_to_peak* (((float)rand()/2.14748365E9)-0.5);
```

Ahhoz, hogy a 12 bites AD átalakítón megfelelően jelenjen meg a jel, meg kell szoroznunk 4096-tal, illetve uint32_t típusúvá kell alakítani.

6. Szinusz jel generálása fix pontos számábrázolás esetén

Fix pontos törtszámábrázolás

Fix pontos 1.15 formátumú törtszámábrázolás esetén a 16 bites számok helyiértékei a következő módon alakulnak:

-1 .	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-10}	2^{-11}	2^{-12}	2^{-13}	2^{-14}	2^{-15}

példák:

- 0110 0000 0000 0000 = 0.5+0.25=0.75
- 1110 0000 0000 0000 = –1+0.5+0.25=–0.25
- 0000 0000 0000 0001 = 2^{-15}=$3.0518\cdot10^{-5}$ =0.000030518
- 1000 0000 0000 0001 = 2^{-15}=–1+$3.0518\cdot10^{-5}$ =–0.999969482421875

- 0111 1111 1111 1111 = $\sum_{n=1}^{15} 2^{-n}$ =0.999969482421875 (legnagyobb pozitív szám)
- 1000 0000 0000 0000 =–1 (legkisebb negatív szám)

A fenti példák alapján is látható, hogy a számábrázolási tartomány: [–1, $1–2^{-15}$]=

= [–1, 0.999969482421875].

Valójában a C nyelv nem támogatja a fix pontos törtszám ábrázolási formátumot, csupán megfelelő könyvtári kiegészítésekkel tesszük lehetővé használatát. Az általunk használt környezetben a q15_t típus reprezentálja a fenti számábrázolási formátumot, amely valójában egy 16 bites integer típus az alábbi típusdefiníció alapján:

```
/**
 * @brief 16-bit fractional data type in 1.15 format.
 */
typedef int16_t q15_t;
```

A számok bitreprezentációja tehát sehol nem tartalmazza annak típusát, a mi feladatunk, hogy az adott bitreprezentációt hogyan kezeljük.

Megjelenítés és értékadás során figyelembe kell vennünk, hogy a tört számoknak megfelelő egész típusban tároljuk a számokat, amelyek az alábbi ábra alapján feleltethetők meg egymásnak:

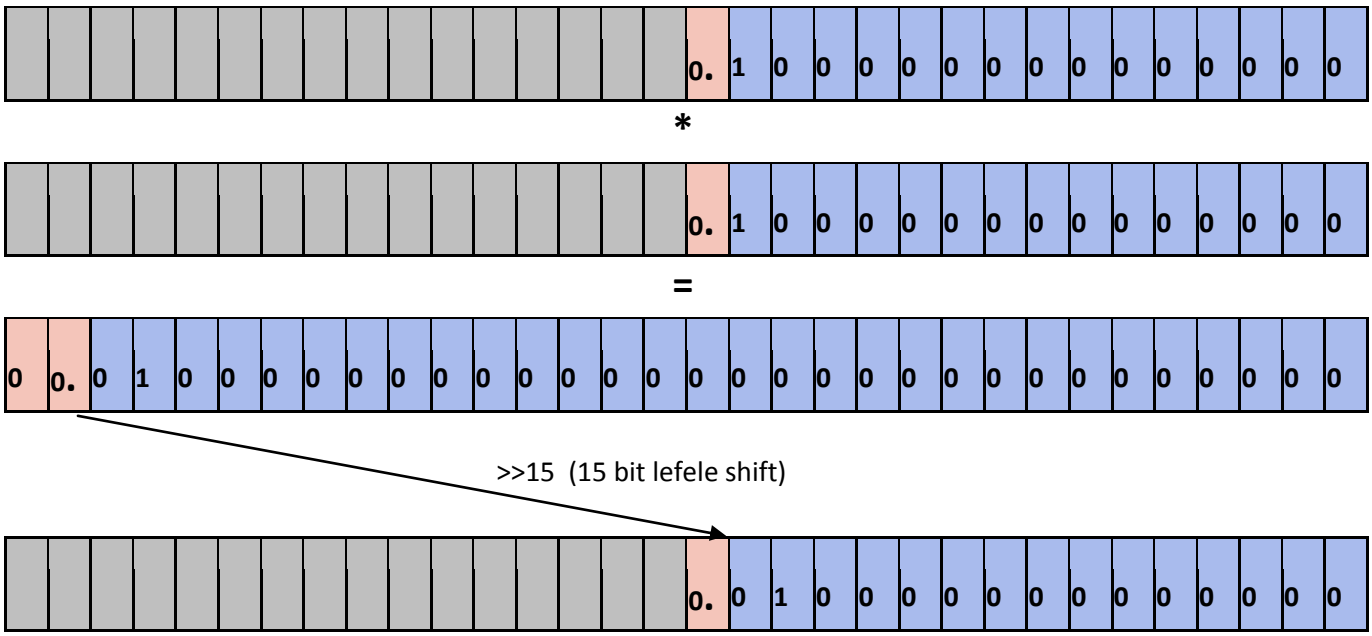
Példák:

- 0.125 értéket szeretnénk adni az x változónak, akkor: x=0.125*32768=4096
- –0.875 értéket szeretnénk adni az x változónak, akkor: x=–0.875*32768=-28672

Visszafele irányba ugyanez az elv érvényesül, tehát ha egy adott q15_t típusú változóban 4096 értéket látunk, akkor az 0.125-ös értéket jelent.

Műveletvégzéseket tekintve az összeadás és kivonás integer számként kezelve is ugyanúgy q15_t típusú számokat eredményez.

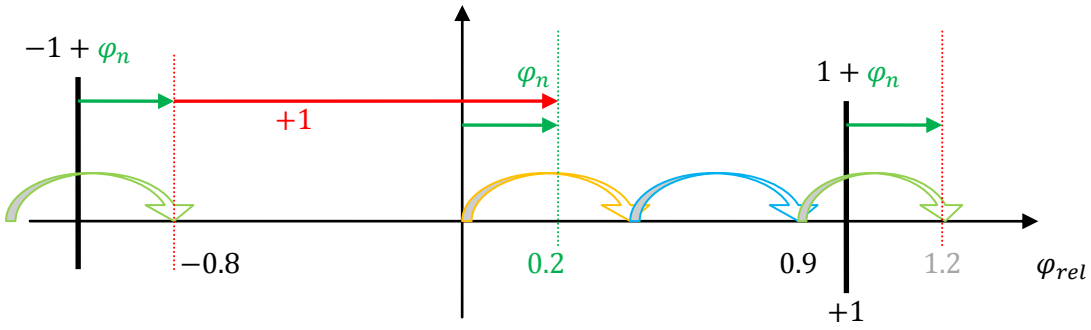
Szorzás esetén vigyázni kell, ugyanis az alábbi példa alapján két q1.15 típusú szám szorzata q2.30 típusú számot eredményez.



Ahhoz, hogy két, q1.15 típusú szám szorzata q1.15 típusú számot eredményezzen, a szorzatot 15 bittel kell lefele irányba shiftelni az alábbi programkódnak megfelelően:

```
q15_t x_q15, y_q15, z_q15;
x_q15 = 0.5*32768;
y_q15 = 0.5*32768;
z_q15 = (x_q15 * y_q15) >> 15;
```

A rendelkezésre álló, q15_t típusokkal dolgozó szinusz függvény a bemeneti fázisszöget [0…1] tartományban várja, ami megfelel a [0…2π] radiánban vett tartománynak. A számábrázolási kényszerek miatt viszont nem tudjuk a 2π értéket ábrázolni, ezért skálázzuk [0…1] intervallumra.
Vegyük észre, hogy a fázist folyamatosan növelve egyszer eléri a 1 értéket, és ekkor átfordul a számláló –1 értéktől. Ha pl. az aktuális fázis 0.9, a növekmény pedig 0.3, akkor a következő ütembeli fázis 1.2 lenne. Az már túlmutat a számábrázolási tartományon, tehát átfordul a változó méghozzá a következő értékre: –1+(1.2–1)=–0.8. A szinusz függvény [0…1] tartományban várja a bemeneti értéket, tehát azt szeretnénk, hogy a bemeneti változó az 1 értéken túljutva 0-tól forduljon át. Ezt úgy érhetjük el, hogy a –1 értéktől átfordult fázisszöghöz 1-et adunk. Az alábbi ábrán látható a módszer illusztrációja:




```
phi_q15 += freq_rel; // step phase
if (phi_q15<0)  //wrap from negative
      phi_q15 += 32768;
```

A szinusz jel DC szintjének és amplitúdójának beállításához a következő programkód használható:

```
out_tmp_q15 = (q15_t) (0.5*32768) + (((int32_t)arm_sin_q15(phi_q15)*9830)>>15);
```

ahol `9830 = 0.3 * 32768`

Ahhoz, hogy a 12 bites DA átalakítón megjeleníthessük a 15 bites számot (előjel bittől eltekintünk, mivel pozitív számokkal dolgozunk), lefele irányba kell shiftelni 3 bittel:

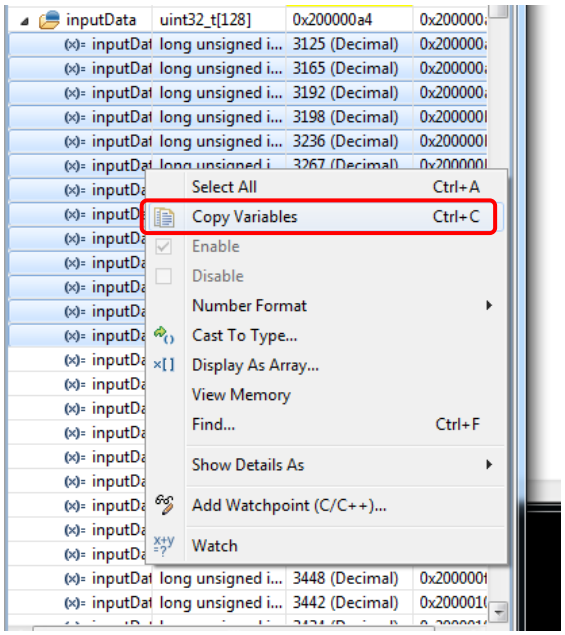
```
out = out_tmp_q15>>3;
```

7. Adatok megjelenítése

A mérés során a mintavételezett és feldolgozott adatokat külön tömbökbe fogjuk menteni, és az ezekben található adatokat egy fájlba kiexportálva az adatokat grafikusan is meg tudjuk jeleníteni egy Python script segítségével. A script elérhető a tárgy honlapján a gyakorlatoknál.

A lépések a következők:

- Jelöljük ki a fájlba mentendő számértékeket a változókat tartalmazó debug ablakban
  - Jelöljük ki az első elemet, aztán a shift gombot nyomva jelöljük ki az utolsó elemet.
  - Első alkalommal jobb gombbal kattintsunk a kijelölt területre, és a Number format menüből válasszuk a Decimal formátumot.
- A kijelölt területre jobb gombbal kattintva a Copy Variables menüpontot válasszuk ki (vágólapra menti az adatokat).
- Nyissunk meg egy szövegfájlt, egy-az-egyben másoljuk bele CTRL+V –vel a kimásolt adatokat, és mentsük le ADCdata.txt néven a display_data_gy7.py programmal egy könyvtárba.
- Dupla kattintással futtassuk a display_data_gy7.py programot
  - Megjegyzés: a display_data_gy7.py programot szerkesztve módosíthatjuk a fájlnevet. Jobb gombbal a display_data_gy7.py programra kattintva Open with: Notepad++
- A megjelenítő program újrafuttatása előtt az ablakot be kell zárni.



Feladatok

1. Állítsuk össze a projektet. Adjuk hozzá az alap jelfeldolgozó keretrendszer működéséhez szükséges c fájlokat.
2. Adjuk a projekthez hozzá a szinuszjel generálásához szükséges c fájlokat, illetve az azokhoz tartozó táblázatokat tartalmazó c fájlt, illetve include-oljuk a megfelelő header fájlokat. Definiáljuk a „ARM_MATH_CM3” változót.
3. Kössük át a fejlesztői kártya DA átalakítóját az AD átalakítóra.
4. Generáljunk szinuszos jelet a következő paraméterekkel:
    a. frekvencia: 50 Hz, DC szint: 0.5 (full scale-hez viszonyítva), amplitúdó 0.3 (full scale-hez viszonyítva)
    b. Vizsgáljuk meg a visszamért jelet.
5. Generáljunk színuszos jelekből összeállított négyszögjelet a következő paraméterekkel:
    a. Az alapharmonikus paraméterei megegyeznek a fent beállítottal, a páratlan felharmonikusok amplitúdói pedig rendre: $A_3 = +\frac{A_1}{3}$, $A_5 = +\frac{A_1}{5}$, $A_7 = +\frac{A_1}{7}$…
    b. Vizsgáljuk meg a visszamért jelet.
6. Generáljunk egyenletes eloszlású fehér zajt a következő paraméterekkel:
    a. DC szint: 0.5 (full scale-hez viszonyítva), peak-to-peak érték 0.1 (full scale-hez viszonyítva)
    b. Vizsgáljuk meg a visszamért jelet.
7. Generáljunk zajos szinuszjelet a következő paraméterekkel:
    a. DC szint: 0.5 (full scale-hez viszonyítva), szinusz jel amplitúdója 0.3 (full scale-hez viszonyítva), zaj amplitúdója 0.1 (full scale-hez viszonyítva)
    b. Vizsgáljuk meg a visszamért jelet.
8. Implementáljunk exponenciális átlagolót. Az átlagolás kimenetét mentsük az inputData_filt változóba.
    a. Az exponenciális átlagolás törésponti frekvenciája fc=100 Hz.
        i. emlékeztetőül:
        $y_n = y_{n-1} + (1-\alpha)(x_n - y_{n-1})$
        ahol $\alpha = e^{-\frac{1}{T \cdot f_s}}$
        az időállandó: $T = \frac{1}{2\pi f_c}$
        tehát más formában: $\alpha = e^{-\frac{1}{T \cdot f_s}} = e^{-2\pi\frac{f_c}{f_s}}$
    b. Változtassuk a jel frekvenciáját f=200 Hz-re, és mérjük meg a szűrt jel amplitúdóját. Vessük össze a következő közelítő képlettel számítható átvitellel:
    $$|H(f)| \approx \left|\frac{1}{1+j\frac{f}{f_c}}\right| = \frac{1}{\sqrt{1+\left(\frac{f}{f_c}\right)^2}}$$
    c.
9. Implementáljunk mozgó ablak átlagolót. Az átlagolás kimenetét mentsük az inputData_filt változóba.
    a. Az átlagolás hossza: N=16 minta.
    b. Az átlagoláshoz az inputData tömbben található adatokat használhatjuk, a buffer cirkularitását elérhetjük a következő módon:

```
#define N_AVG  16
      uint32_t y_avg = 0;
      uint32_t i_avg;
      for (i_avg=0;i_avg<N_AVG;i_avg++){
            y_avg += inputData[(inputData_idx-i_avg)&(N-1)];
      }
      y_avg = y_avg/N_AVG;
```

c. Vizsgáljuk meg az 50 Hz-es jelet!

d. Állítsuk a jel frekvenciáját 200 Hz-re, és mérjük meg az amplitúdóját. Számítsuk ki az átvitelt a következő közelítéssel:

$$|H(f)| = \frac{\sin\left(\pi \frac{f}{f_c}\right)}{\pi \frac{f}{f_c}}$$

ahol $f_c = \frac{f_s}{N}$

e. Számítsuk ki a szűrő első zérushelyét, és állítsuk be a frekvenciát erre az értékre. Keressünk további zérusokat is.

f. Vizsgáljuk meg az átvitelt az első zérushely hez tartozó frekvencia másfélszeresén, és vessük össze a kiszámított értékkel: $|H(f = 1.5 \cdot f_c)| = \frac{1}{1.5\pi}$.

10. Valósítsuk meg a zajgenerátort fix pontos számábrázolással.
    a. DC szint: 0.5 (full scale-hez viszonyítva), peak-to-peak érték 0.1 (full scale-hez viszonyítva)
11. Valósítsuk meg a szinuszgenerátort fix pontos q15 formátumú számábrázolással.
    a. frekvencia: 50 Hz, DC szint: 0.5 (full scale-hez viszonyítva), amplitúdó 0.3 (full scale-hez viszonyítva)
12. Generáljunk zajos szinusz jelet az előző paraméterek alapján.

Melléklet: adatmegjelenítést végző Python kód

```
import numpy as np
from math import *
import matplotlib.pyplot as plt

fname = 'ADCdata.txt'

fs = 5000.0

print "fs = %f" % (fs)

Ts = 1/fs

fO = open(fname)

ln = fO.readline()

arr = np.array([])

while not(ln==''):
    tab1 = ln.find('\t') # elso tabulator megtalalasa
    tab2 = ln.find('\t', tab1+1) # masodik tabulator megtalalasa
    tab3tab = ln.find('\t', tab2+1) # ha sima tabbal van elvalasztva
    tab3par = ln.find('(') # ha zarojellel meg van adva a tipus
    if (tab3par<0):
        tab3par = tab3tab
    tab3 = min(tab3tab, tab3par)
    numberRaw = ln[tab2+1:tab3] # a nyers szam
    if len(numberRaw)>2:
        if numberRaw[0:2]=='0x':
            numberRaw = int(numberRaw,16) # ha hexa
    arr = np.append(arr, float(numberRaw))
    ln = fO.readline() # read a new line

N = len(arr)
t = np.array(range(N)) * Ts * 1e3

plt.subplot(2,1,1)
plt.plot(t, arr,'.-')
plt.grid(True)
plt.xlabel('t [ms]')
plt.ylabel('x(t) [LSB]')
plt.ylim(top=4096, bottom=0)

plt.subplot(2,1,2)
plt.plot(t, arr/4096,'.-')
plt.grid(True)
plt.xlabel('t [ms]')
plt.ylabel('x(t) [Full scale]')
plt.ylim(top=1, bottom=0)

plt.show()
```

Melléklet: kiindulási programkód

```
#include "em_device.h"
#include "em_chip.h"


#include "em_system.h"

#include "em_adc.h"
#include "em_cmu.h"
#include "em_core.h"
#include "em_dac.h"
#include "em_gpio.h"
#include "em_timer.h"

#include "bsp.h"

// ------------------------------




#error("a kovetkezo fajlokat hozza kell adni a projekthez: em_adc.c, em_cmu.c,
em_core.c, em_dac.c em_gpio.c, em_system.c, em_timer.c")
//
c:\SiliconLabs\SimplicityStudio\v4\developer\sdks\gecko_sdk_suite\v2.3\platform\em
lib\src\


#error("a kovetkezo fajlokat hozza kell adni a projekthez: bsp_bcc.c,
bsp_stk_leds.c, bsp_stk.c")
//
c:\SiliconLabs\SimplicityStudio\v4\developer\sdks\gecko_sdk_suite\v2.3\hardware\ki
t\common\bsp\

#error ("a kovetkezo fajlt be kell include-olni: #include ""arm_math.h""")
//
c:\SiliconLabs\SimplicityStudio\v4\developer\sdks\gecko_sdk_suite\v2.3\platform\CM
SIS\

#define N 128
uint32_t inputData[N];
uint32_t inputData_filt[N];
int32_t inputData_idx = 0;


#define FS 5000 // mintaveteli frekvencia megadasa Hz-ben
#define TIMER_DIV (12000000/FS-1) // osztasi arany


int getBP0(void){
      return (!(GPIO_PinInGet(gpioPortB, 9)));
}
int getBP1(void){
      return (!(GPIO_PinInGet(gpioPortB,10)));
}
```

```
// ****************************************
// * process data *
// ****************************************
uint32_t process(uint32_t in){
	uint32_t out;

	//------------------
	// save input data
	inputData_idx = (inputData_idx+1)&(N-1);
	inputData[inputData_idx] = in;
	inputData_filt[inputData_idx] = in;
	//------------------




	return out;
}

// ****************************************
// * T I M E R I R Q *
// ****************************************
uint32_t ADC_data_in, DAC_data_out;
uint32_t TimerCnt;
void TIMER0_IRQHandler(void){
	ADC_data_in = ADC_DataSingleGet(ADC0);
	ADC_Start(ADC0, adcStartSingle);
	DAC_data_out = process(ADC_data_in);
	DAC_Channel0OutputSet(DAC0, DAC_data_out);
	TIMER_IntClear(TIMER0, TIMER_IF_OF);
	TimerCnt++;
	if (TimerCnt>FS)
		{ TimerCnt=0;
			BSP_LedToggle(0);
		}
}


int main(void)
{
  /* Chip errata */
  CHIP_Init();


  //*****************************************
  // CMU configuration *
  //*****************************************
  //void CMU_OscillatorEnable(CMU_Osc_TypeDef osc, bool enable, bool wait);
  CMU_OscillatorEnable(cmuOsc_HFXO, true, true);
  //void CMU_ClockSelectSet(CMU_Clock_TypeDef clock, CMU_Select_TypeDef ref);
  CMU_ClockSelectSet(cmuClock_HF, cmuSelect_HFXO);
  //SystemHFXOClockSet(uint32_t freq); SystemHFXOClockSet(48000000);
  //HFPERCLK: 48MHz/4 = 12MHz
  //void CMU_ClockDivSet(CMU_Clock_TypeDef clock, CMU_ClkDiv_TypeDef div);
  CMU_ClockDivSet(cmuClock_HFPER, cmuClkDiv_4);
  // enable clock signals
  //CMU_ClockEnable(CMU_Clock_TypeDef clock, bool enable);
  CMU_ClockEnable(cmuClock_HFPER, true);
  CMU_ClockEnable(cmuClock_GPIO, true);
```

```
  CMU_ClockEnable(cmuClock_TIMER0, true);
  CMU_ClockEnable(cmuClock_ADC0, true);
  CMU_ClockEnable(cmuClock_DAC0, true);

  //*****************************************
  // ADC configuration *
  //*****************************************
  ADC_Init_TypeDef ADC0_Init = ADC_INIT_DEFAULT;
  //void ADC_Init(ADC_TypeDef *adc, const ADC_Init_TypeDef *init);
  ADC0_Init.prescale = 0;
  // 12MHz ADC0_Init.timebase = 16;
  ADC0_Init.warmUpMode = adcWarmupKeepADCWarm;
  ADC_Init(ADC0, &ADC0_Init);
  ADC_InitSingle_TypeDef ADC0_s_Init = ADC_INITSINGLE_DEFAULT;
  //void ADC_InitSingle(ADC_TypeDef *adc, const ADC_InitSingle_TypeDef *init);
  ADC0_s_Init.reference = adcRefVDD;
  ADC0_s_Init.input = adcSingleInputCh0;
  ADC_InitSingle(ADC0, &ADC0_s_Init);



  //*****************************************
  // DACconfiguration *
  //*****************************************
  DAC_Init_TypeDef DAC_init = DAC_INIT_DEFAULT;
  DAC_init.reference = dacRefVDD;
  DAC_init.prescale = 4; // 2^4=16; max 1MHz
  //void DAC_Init(DAC_TypeDef *dac, const DAC_Init_TypeDef *init);
  DAC_Init(DAC0, &DAC_init);
  DAC_InitChannel_TypeDef DAC_ChInit = DAC_INITCHANNEL_DEFAULT;
  DAC_ChInit.enable = 1;
  //void DAC_InitChannel(DAC_TypeDef *dac, const DAC_InitChannel_TypeDef *init,
unsigned int ch);
  DAC_InitChannel(DAC0, &DAC_ChInit, 0);


  //*****************************************
  // Timer configuration *
  //*****************************************
  TIMER_Init_TypeDef TIMER0_init = TIMER_INIT_DEFAULT;
  //void TIMER_Init(TIMER_TypeDef *timer, const TIMER_Init_TypeDef *init);
  TIMER_Init(TIMER0, &TIMER0_init); TIMER_CounterSet(TIMER0, 0); //
  //__STATIC_INLINE void TIMER_TopSet(TIMER_TypeDef *timer, uint32_t val)
  TIMER_TopSet(TIMER0, TIMER_DIV); // 48MHz/4/x = 12MHz/x
  //__STATIC_INLINE void TIMER_IntClear(TIMER_TypeDef *timer, uint32_t flags);
  TIMER_IntClear(TIMER0, TIMER_IF_OF);
  //TIMER_IntEnable(TIMER_TypeDef *timer, uint32_t flags);
  TIMER_IntEnable(TIMER0, TIMER_IF_OF);
  NVIC_EnableIRQ(TIMER0_IRQn);

  // ******************************
  // *     LED inicializalasa      *
  // ******************************
  BSP_LedsInit();

  // ******************************
  // * nyomogombok inicializalasa  *
  // ******************************
  GPIO_PinModeSet(gpioPortB, 9, gpioModeInput, 0);
```

```
  GPIO_PinModeSet(gpioPortB, 10, gpioModeInput, 0);

  /* Infinite loop */
  while (1) {

        if ((0==inputData_idx)&&getBP0()){
              BSP_LedSet(1);
        } else{
              BSP_LedClear(1);
        }
  }
}
```