Beágyazott és Ambiens Rendszerek

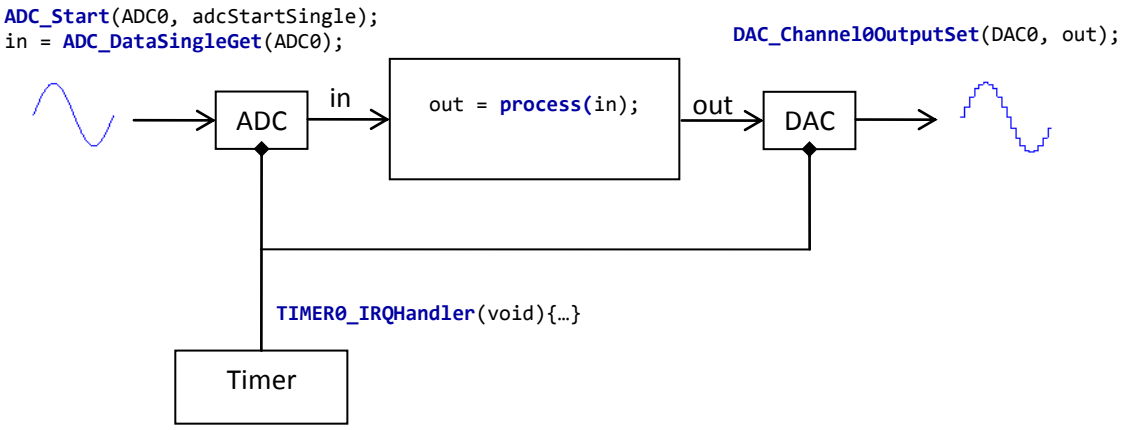
6. gyakorlat tematikája

SW architektúra mérésadatgyűjtő rendszerre

A gyakorlat során a következő témakörökkel ismerkedünk meg:

- SW architektúra mérésadatgyűjtő rendszerre,
  - mintavételezés időzítése
  - ADC kezelése
  - DAC kezelése
- adatelemzés,
- jelgenerálás,
- szűrés.

A létrehozandó rendszer blokkdiagramja az alábbi ábrán látható:



1. Programfelépítés

A valós idejű adatgyűjtő és –feldolgozó rendszerek szoftverarchitektúráit előadásokon részletesen elemeztük. A gyakorlat során a következő szoftverarchitektúrát alkalmazzuk:

- Perifériák inicializálása
  - órajel-menedzsment
  - időzítő
  - ADC
  - DAC
- Időzítő megszakítás (Timer IT)
  - Előző Timer IT-ban indított ADC átalakítás eredményének beolvasása
  - Újabb ADC konverzió elindítása
  - Adatfeldolgozás

- Adatfeldolgozás eredményének kiadása DAC-n
- Timer IF flag törlése

A fenti architektúra előnye, hogy nem kell várni az ADC konverzió befejezésére (a következő timer IT-ig biztosan befejeződik), és nem kell újabb megszakítást megadni az AD-hez.

A projekt létrehozása során már a kezdetben érdemes a következő c fájlokat hozzáadni a projekthez, illetve header fájlokat include-olni:
Az időzítő kezeléséhez hozzá kell adni a projekthez az em_adc.c, em_cmu.c, em_core.c, em_dac.c em_gpio.c, em_system.c és em_timer.c fájlokat[1], és include-olni kell az em_device.h, em_chip.h, em_system.h, em_cmu.h, em_timer.h, em_adc.h, em_dac.h, em_gpio.h, bsp.h fájlokat. A LED-ek kezeléséhez a projekthez hozzá kell adni a bsp_bcc.c, bsp_stk_leds.c, bsp_stk.c[2], és include-oljuk a bsp.h fájlt.

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\

A munka gyorsítása érdekében a következő include-ok bemásolhatók a kódba:

```
#include "em_device.h"
#include "em_chip.h"
#include "em_system.h"
#include "em_cmu.h"
#include "em_timer.h"
#include "em_adc.h"
#include "em_dac.h"
#include "em_gpio.h"
#include "bsp.h"
```

[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\

2. CMU (Clock Management Unit) inicializálása

Az órajelelosztó hálózat kapcsán a következő feladatokat végezzük el:

- Nagyfrekvenciás (48 MHz-es) órajelbemenet használata: mivel nagysebességű és pontosan időzített adatgyűjtést szeretnénk végrehajtani, ezért használjuk ezt a forrást.
  - külső oszcillátor engedélyezése
  - külső oszcillátor kiválasztása órajelforrásként
- Nagyfrekvenciás általános periféria-órajel konfigurálása
  - periféria órajel engedélyezése (kimaradhat, default engedélyezett)
  - 48 MHz-es órajel leosztása 4-el:
    - az ADC max 13 MHz-es órajellel képes üzemelni
    - lehetett volna a leosztást az ADC-nél is kiválasztani, de így biztonságosabb, biztosan nem tudjuk az ADC órajelét rosszul megadni
- GPIO órajel engedélyezése
- Timer órajel engedélyezése
- ADC órajel engedélyezése
- DAC órajel engedélyezése

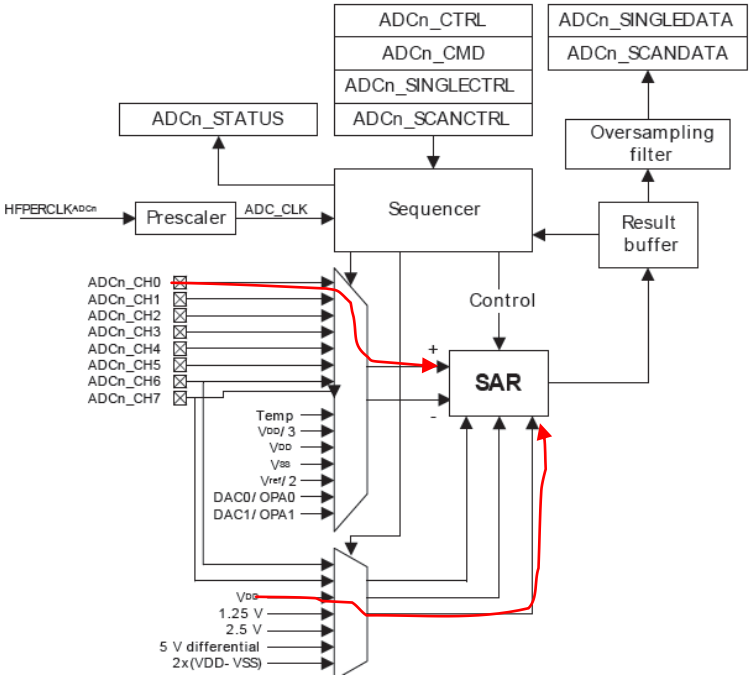
```
//*********************************************
//              CMU configuration              *
//*********************************************

//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);
```

3. ADC (Analog-to-Digital Converter) konfigurálása

Az ADC 8 külső bemeneti multiplexált csatornával rendelkezik, ebből mi egy, a CH0 csatornát fogunk használni. Ennek megfelelően a Single conversion üzemmódot használjuk, több csatorna esetén lenne értelme a Scan üzemmód használatának. Referenciafeszültségként a 3.3V-os tápfeszültséget használjuk. Mivel a periféria órajelet 12 MHz-re leosztottuk, az ADC maximális órajele pedig 13 MHz, így az órajelét tovább nem osztjuk.

A konverzió szukcesszív approximációs módszerrel történik, amely lényegében a bitszámmal megegyező órajelet igényel (plussz overhead, lásd később).

A konverzió több módszerrel indítható. A mintakódban az ADC0_CMD regiszter SINGLESTART (0. bit) bitjének 1-esbe állításával indítjuk a konverziót. Ezt megfelelő könyvtári függvény segítségével tesszük meg.



Az inicializáció az ADC beépített könyvtári függvénye segítségével történik. Az inicializálás során egy ADC beállítását tartalmazó adatstruktúrát adunk át, és a függvény ez alapján állítja be az ADC paramétereit.

Az inicializálandó paramétereket tartalmazó struktúra a következő mezőket tartalmazza:

- `ovsRateSel`: túl-mintavételezés [default: adcOvsRateSel2]
  - hány minta átlagolásával kapjuk meg az ADC eredményt
  - értéke lényegtelen, mert ezt az üzemmódot külön engedélyezni kellene
- `lpfMode`: kell-e a bemenetre aluláteresztő szűrő [default: adcLPFilterBypass]
  - a default érték megfelel: nem kell bemeneti átlapolásgátló szűrő
- `warmUpMode`: feléledési mód [default: adcWarmupNormal]
  - Az ADC működéséhez el kell indulnia a belső áramköröknek (pl. referenciafeszültség). Ezeket a belső áramköri egységeket lekapcsolva tartjuk alapbeállításként. A feléledési idő 1µs+5µs (lásd adatlap)
  - Egyszerűség kedvéért az ADC-t nem kapcsoljuk le: adcWarmupKeepADCWarm módban használjuk
- `timebase`: a processzor a feléledési időket automatikusan kezeli, ehhez [default: 1]
  - a timebase értéket úgy kell beállítani, hogy az ADC órajel/timebase leosztás legalább 1 µs-es periódusidőt adjon ki.
  - Az órajel 12 MHz, a timebase értéket így 16-ra állítjuk, így az esetleges időzítések biztosan teljesülnek.
- `prescale`: ADC órajel prescaler [default: 0]
  - nem szeretnénk tovább osztani az ADC periféria órajelét (marad 12 MHz)
  - hagyjuk a default 0 értéken.

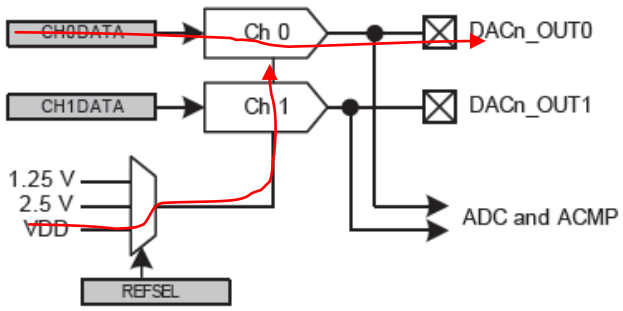
Az ADC periféria általános inicializálását követően inicializálni kell a használt csatornát (jelen esetben a CH0 csatornát).

- `prsEnable`: peripherial reflex system [default: false]
  - az egyes perifériák egymást triggerelhetik.
- `prsSel`: peripherial reflex system forrása
  - lényegtelen, nem engedélyezzük
- `acqTime`: mintavételi idő ADC órajelben megadva [default: 1 órajel]
  - a mintavevő-tartó áramkör mennyi ideig legyen a jelre rákapcsolva
  - a default érték megfelelő, főleg nagyimpedanciás források esetén lehet kritikus
- `reference`: referenciafeszültség [default: 1.25V belső referencia]
  - a 3.3 V-os tápfeszültségre állítjuk, így nagyobb a jeltartomány
- `resolution`: felbontás [default: 12 bit]
- `input`: melyik multiplexált csatornát választjuk [default: CH0]
  - A CH0 megfelelő, de ezt a kódban is megerősítjük
- `diff`: differenciális vagy aszimmetrikus bemenet [default: aszimmetrikus bemenet]
- `leftAdjust`: az ADC adat hogyan legyen igazítva a 32 bites regiszterben [default: jobbra]
- `rep`: a konverzió befejeztét követően kezdődjön-e újabb konverzió [default: ne kezdődjön]
  - default érték megfelel, minden átalakítást mi szeretnénk indítani

```
//*****************************************
//              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);
```

4. DAC (Digital-to-Analog Converter) konfigurálása

A DAC 2 kimenettel rendelkezik, ebből a CH0 kimenetet fogjuk használni. Mindkét kimenet maximum 12 bites.



Az inicializáció a DAC beépített könyvtári függvénye segítségével történik. Az inicializálás során egy DAC beállítását tartalmazó adatstruktúrát adunk át, és a függvény ez alapján állítja be a DAC paramétereit.

Az inicializálandó paramétereket tartalmazó struktúra a következő mezőket tartalmazza:

- `refresh`: frissítési intervallum [default: 8 DAC órajel, ez a minimum]
  - ilyen gyakorisággal frissül a kimeneti érték
- `reference`: referenciafeszülség [default: 1.25 V belső referencia]
  - Mi a kódunkban a 3.3 V-os tápfeszültséget használjuk referenciaként
- `outMode`: hova kapcsolódjon a kimenet [default: egy kimenei lábra]
  - megfelelő, mi is közvetlenül egy kimeneti lábra szeretnénk kötni (lehetne pl. belső OPA-ra kötni vagy ADC-re visszacsatolni)
- `convMode`: konverziós üzemmód [default: folyamatos]
  - megfelel a default. Lehetne pl. kikapcsolni a minták között, ekkor egy külső elektronikának kellene tartani az értéket.
- `prescale`: előosztó [default: 0]
  - Adatlap alapján az órajel max. 1 MHz lehet, így 16-os előosztót állítunk be.
- `lpEnable`: kimeneti aluláteresztő szűrő [default: tiltva]
  - nem szeretnénk kimeneti szűrőt, megfelel a default érték
- `ch0ResetPre`: prescaler a CH0 csatornán [default: tiltva]
  - megfelelő
- `outEnablePRS`: peripheral reflex system engedélyezése [default: tiltva]
- `sineEnable`: szinuszgenerálás engedélyezése [default: tiltva]
- `diff`: differenciális kimenet [default: tiltva]

A DAC periféria általános inicializálását követően inicializálni kell a használt csatornát (jelen esetben a CH0 csatornát).

- `enable`: engedélyezés [default: false]
    - engedélyeznünk kell, true-ra állítjuk
- `prsEnable`: peripheral reflex system engedélyezése [default: false]
- `prsSel`: lásd előző pont
- `refreshEnable`: automatikus frissítés engedélyezése [default: false]
    - regiszterbe történő írással frissítjük a kimenetet, egyébként nem kell frissítés

```
  //*****************************************
  //              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);
```

5. Időzítő konfigurálása

Az időzítő inicializálásával az 5. gyakorlaton foglalkoztunk, ezért most csak a konkrét feladathoz szükséges eltéréseket emeljük ki.

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.

Külön meg kell adnunk egy FS konstanst, amely a mintavételi frekvenciát adja meg. Mivel a periféria órajel 12 MHz, így a számláló TOP értéke (5 kHz esetén pl. 1200-as osztás kell):

$$TOP_{TIMER} = \frac{12000000\ Hz}{FS} - 1 = \frac{12000000\ Hz}{5000\ Hz} - 1 = 2399$$

```
#define FS 5000   // mintavételi frekvencia megadása Hz-ben
#define TIMER_DIV  (12000000/FS-1) // osztási arány

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

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. */      \
  }
```

A megszakítás lekezeléséhez az alábbi függvényeket kell implementálni a programkódban:

```
//  ****************************************
//  *            process data              *
//  ****************************************

uint32_t process(uint32_t in){
    uint32_t out;

//  ide jön a jelfeldolgozás, pl. kimenet=bemenet
    out = 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);
    }

}
```

A timer megszakításrutinban:

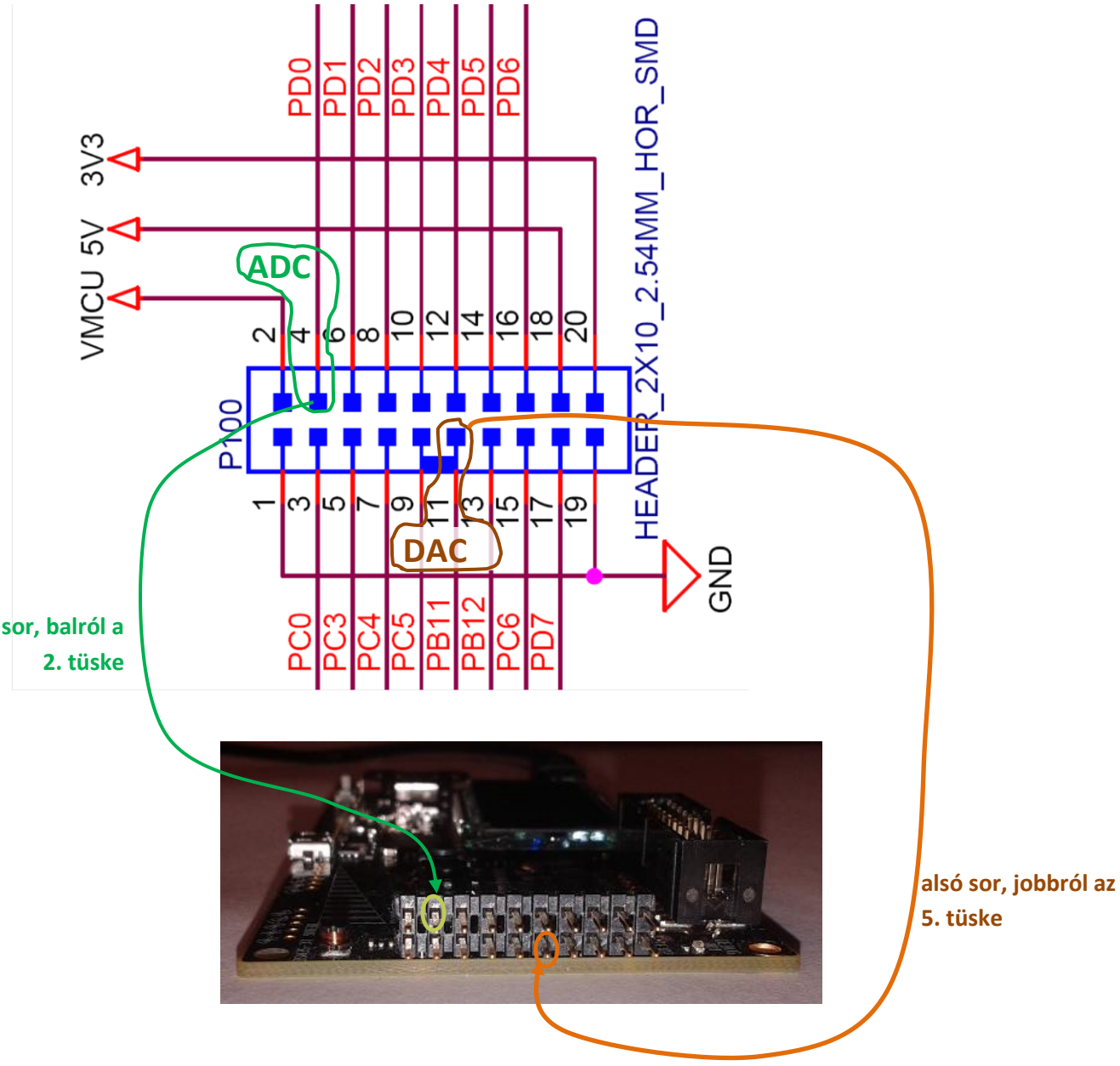
- beolvassuk az előző timer IT során elindított ADC konverzió eredményét.
    - `ADC_data_in = ADC_DataSingleGet(ADC0);`
- Új AD átalakítást indítunk el
    - `ADC_Start(ADC0, adcStartSingle);`
- meghívjuk az adatfeldolgozást végző függvényt
    - `DAC_data_out = process(ADC_data_in);`
- kiadjuk az adatfeldolgozás végeredményét a DAC-n
    - `DAC_Channel0OutputSet(DAC0, DAC_data_out);`
- Töröljük a Timer megszakítás flaget
- Másodpercenként LED-et villogtatunk: jelezzük a működést

6. Ki- és bemeneti lábak feltérképezése

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:

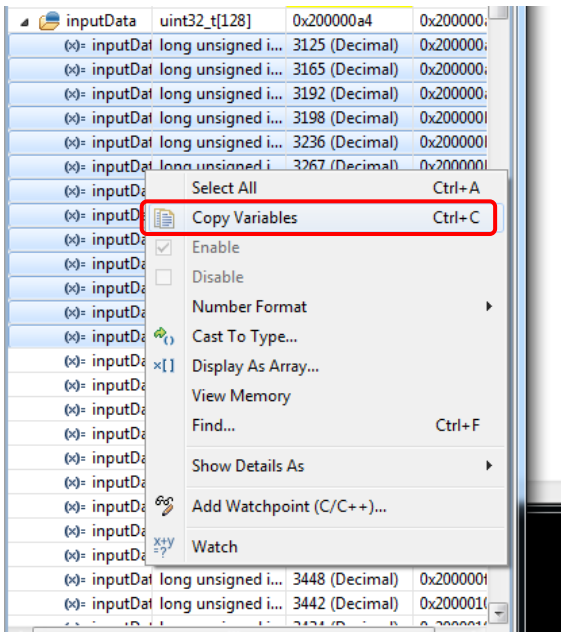


7. Adatok megjelenítése

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.
- 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.py programmal egy könyvtárba.
- Dupla kattintással futtassuk a display_data.py programot
  - Megjegyzés: a display_data.py programot szerkesztve módosíthatjuk a fájlnevet. Jobb gombbal a display_data.py programra kattintva Open with: Notepad++
- A megjelenítő program újrafuttatása előtt az ablakot be kell zárni.



Feladatok

1. Jelenítsük meg a bemeneti adatokat:
    a. Mentsük el folyamatosan a bemeneti adatokat egy 128 mintát tartalmazó cirkuláris tömbben.
    b. Futtassuk a programot,
    c. érintsük meg folyamatosan ujjunkkal az ADC bemenetet,
    d. állítsuk le a programot (ez után engedhetjük el a bemenetet),
    e. mentsük ki az adatokat az ADCdata.txt fájlba,
    f. jelenítsük meg a display_data.py program futtatásával.
2. Generáljunk fűrész jelet, és adjuk ki a DAC-n
    a. A fűrészjel generálása egy fázisváltozó növelésével érhető el. A változó értékét minden mintavételi ütemben növeljük egy adott értékkel, és közvetlenül adjuk ki a DAC-n.
    b. Segítség: a 12 bites DAC 0…4095 értéket vár. Ha egy változót minden mintavétel során dX értékkel növelünk, akkor 4095/dX mintavételi ütem alatt futjuk át a teljes tartományt. Ha egy f0 frekvenciájú fűrészjelet szeretnénk generálni, akkor az FS/f0 ütem alatt fut fel 0-ról 4095-ig. Tehát:

$$dX = 4095 * \frac{f0}{FS}$$

    x=x+dX;
    Legyen a frekvencia: f0=100Hz
    c. Mérjük vissza a jelet: csatlakoztassuk a gyakorlatvezető által kiosztott vezetékek segítségével a DAC kimenetet az ADC bemenetre. **FIGYELEM!!!** Többször is ellenőrizzük le nagyon alaposan, hogy jó lábakat kötünk-e össze (többször számoljuk le, ellenőrizze le a társunk is), mert ha véletlenül két kimenetet kötünk össze, az a kártya tönkremenetelét okozhatja. Érdemes először a kábelt a DAC kimenethez csatlakoztatni, mert azt nehezebb megkeresni.
    d. Vezéreljük telítésbe a jelet, ha annak értéke nagyobb, mint 3000 LSB
    e. Vezéreljük a jelet telítésbe, ha annak értéke kisebb, mint 100 LSB
    f. Generáljunk háromszög jelet.

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.plot(t, arr,'.-')
plt.grid(True)
plt.xlabel('t [ms]')
plt.ylabel('x(t)')
plt.show()
```