Embedded and Ambient Systems

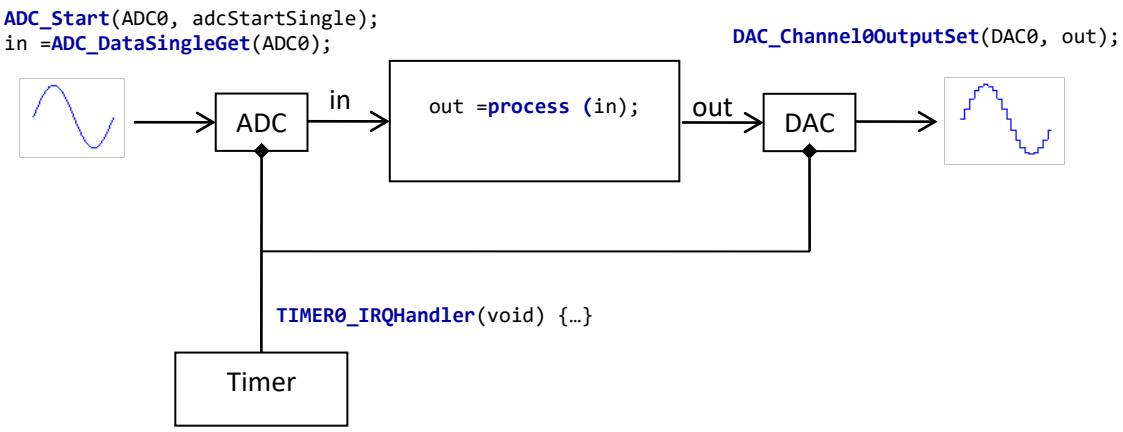
Exercise 6 topics

SW architecture for measurement data acquisition system

During the exercise we will get acquainted with the following topics:

- SW architecture for measurement data acquisition system,
  - sampling timing
  - ADC handling
  - DAC management
- data analysis,
- signal generation,
- filtration.

The block diagram of the system to be created is shown in the following figure:



1. Program structure

The software architectures of the real-time data acquisition and processing systems were analyzed in detail in lectures. The following software architecture is used in the exercise:

- Initializing peripherals
  - clock management
  - timer
  - ADC
  - DAC
- Timer IT
  - Retrieve the result of an ADC transformation initiated in a previous Timer IT
  - Start another ADC conversion
  - Data processing

- Output of data processing result on DAC
- Clear Timer IF flag

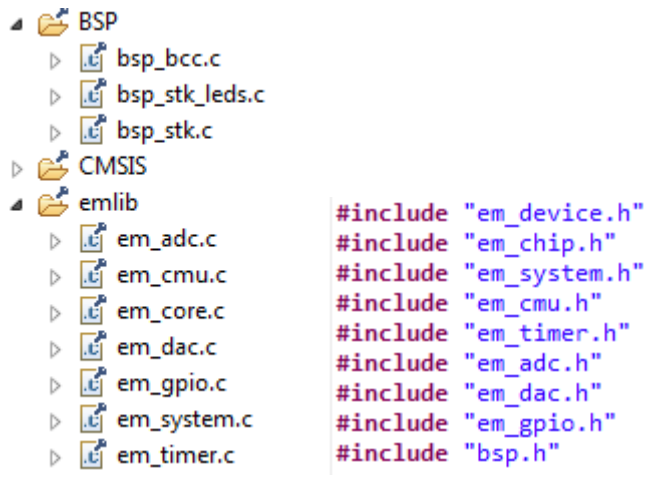
The advantage of the above architecture is that you do not have to wait for the ADC to complete the conversion (conversion will be definitely ended until the next timer IT) and you do not have to specify another interrupt for the AD.

When creating a project, you should immediately add the following c files to the project, and include header files:
To manage the timer, add em_adc.c, em_cmu.c, em_core.c, em_dac.c em_gpio.c, em_system.c, and em_timer.c to your project.[1] and include 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. To manage the LEDs, add bsp_bcc.c, bsp_stk_leds.c, bsp_stk.c to the project[2] and include the bsp.h file.

To speed up your work, you can access the directories containing the C files by clicking the following quick link:

C: \ SiliconLabs \ SimplicityStudio \ v4 \ developer \ sdks \ gecko_sdk_suite \ v1.1 \ platform \ emlib \ src \

C: \ SiliconLabs \ SimplicityStudio \ v4 \ developer \ sdks \ gecko_sdk_suite \ v1.1 \ hardware \ kit \ common \ bsp \



To speed up the work, the following include can be copied into the code:

```
#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] path: SimplicityStudio \ developer \ sdks \ gecko_sdk_suite \ v1.1 \ platform \ emlib \ src \
[2] path: SimplicityStudio \ developer \ sdks \ gecko_sdk_suite \ v1.1 \ hardware \ kit \ common \ bsp \

2. Initialize the CMU (Clock Management Unit)

In connection with the clock distribution network, we perform the following tasks:

- Use of high frequency (48 MHz) clock input: Since we want to perform high speed and precisely timed data acquisition, we use this source.
    - enable external oscillator
    - selecting an external oscillator as the clock source
- Configure a high-frequency generic peripheral clock
    - enable peripheral clock (may be omitted, default enabled)
    - 48 MHz clock division by 4:
        - the ADC can operate at a clock speed of up to 13 MHz
        - it would have been possible to set the prescaler at the ADC, but this is safer, we certainly can't set the ADC clock incorrectly
- Enable GPIO clock
- Enable timer clock
- Enable ADC clock
- Enable DAC clock

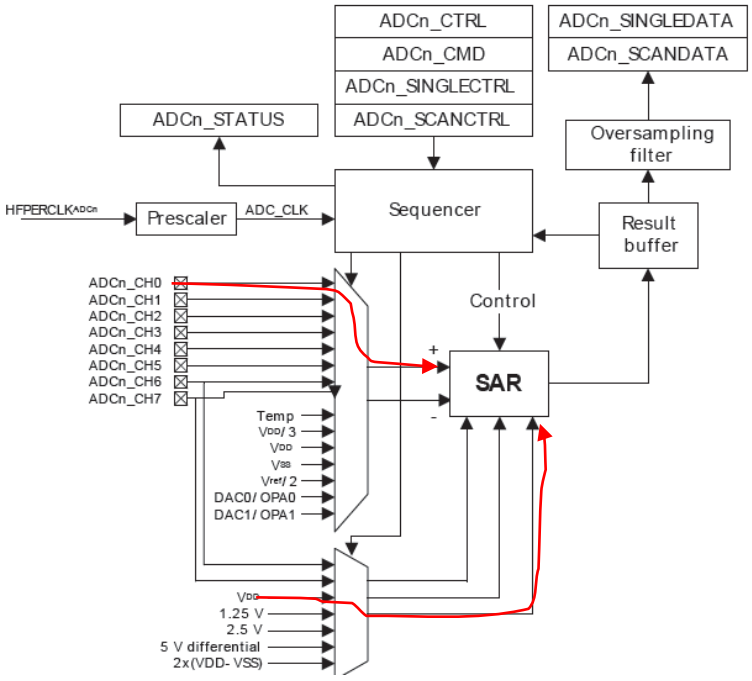
```
// ****************************************
// CMU configuration *
// ****************************************

// 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, boolen 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. Configure ADC (Analog-to-Digital Converter)

The ADC has 8 external input multiplexed channels, of which we will use the CH0 channel. Accordingly, we use the Single conversion mode, for multiple channels it would make sense to use the Scan mode. The reference voltage is 3.3V. Since the peripheral clock is divided to 12 MHz and the maximum clock of the ADC is 13 MHz, the clock is not further divided.

The conversion is done using a successive approximation method, which requires the number of the clock cycles equal to the number of bits (plus overhead, see later).

There are several ways to start a conversion. In the sample code, the conversion is started by setting bit SINGLESTART (bit 0) of register ADC0_CMD to 1. We do this using an appropriate library function.



Initialization is done using the ADC's built-in library function. During initialization, a data structure containing the ADC configuration is passed, and the function sets the ADC parameters accordingly.

The structure containing the parameters to be initialized contains the following fields:

- `ovsRateSel`: over-sampling [default: adcOvsRateSel2]
  - how many samples to be averaged in order to get the ADC result
  - it is irrelevant because this mode is not enabled now
- `lpfMode`: whether we need input low pass filter [default: adcLPFilterBypass]
  - the default value is correct: no input anti-overlapping filter is required
- `warmUpMode`: recovery mode [default: adcWarmupNormal]
  - The internal circuits (eg reference voltage) must be started up for the ADC to operate. These internal circuit units are kept off by default. The rise time is 1µs + 5µs (see data sheet)
  - To keep things simple, the ADC is not turned off: used in adcWarmupKeepADCWarm mode
- `timebase`: the processor handles wake-up times automatically [default: 1]
  - the timebase value must be set so that the ADC clock / timebase division (prescaler) outputs a period time of at least 1 µs.
  - The clock is 12 MHz, so the timebase is set to 16 (16/12MHz > 1 µs), so that any timings are met.
- `prescale`: ADC clock prescaler [default: 0]
  - we do not want to further share the clock signal of the ADC peripheral (12 MHz remains)
  - leave the default at 0.

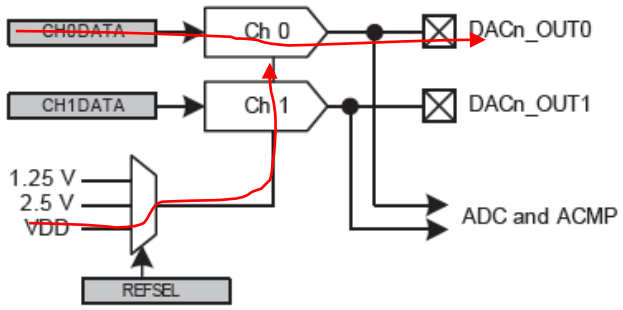
After the general initialization of the ADC peripheral, the channel used (in this case the CH0 channel) must be initialized.

- `prsEnable`: peripheral reflex system [default: false]
  - each peripheral may trigger each other.
- `prsSel`: source of peripheral reflex system
  - irrelevant, not allowed
- `acqTime`: sampling time in ADC clock [default: 1 clock]
  - how long the sample holder circuit should be connected to the signal
  - the default value can be critical, especially for high-impedance sources
- `reference`: reference voltage [default: 1.25V internal reference]
  - it is set to a supply voltage of 3.3 V, so the signal range is larger
- `resolution`: resolution [default: 12 bits]
- `input`: which multiplexed channel to choose [default: CH0]
  - CH0 is appropriate, but this is also confirmed in the code
- `diff`: differential or asymmetric input [default: asymmetric input]
- `leftAdjust`: how to align ADC data in a 32-bit register [default: right]
- `rep`: start another conversion after conversion is complete [default: don't start]
  - default value is correct, we want to start all conversions

```
// ****************************************
// 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. Configure DAC (Digital-to-Analog Converter)

The DAC has 2 outputs, of which we will use the CH0 output. Both outputs are up to 12 bits.



Initialization is done using the DAC's built-in library function. During initialization, a data structure containing a DAC setting is passed, and the function sets the DAC parameters accordingly.

The structure containing the parameters to be initialized contains the following fields:

- `refresh`: refresh interval [default: 8 DAC clock, this is the minimum]
  - the output value is updated with this frequency
- `reference`: reference voltage [default: 1.25 V internal reference]
  - In our code, we use the 3.3 V supply voltage as a reference
- `outMode`: where to connect the output [default: to an output pin]
  - appropriate, we also want to connect directly to an output pin (could be connected to an internal OPA (operational power amplifier) or fed back to an ADC, for example)
- `convMode`: conversion mode [default: continuous]
  - corresponds to the default. It could be e.g. turn off between samples, then an external electronics should hold the value.
- `prescale`: presenter [default: 0]
  - According to the data sheet, the clock is max 1 MHz, so a 16 division is se tat the prescaler.
- `lpEnable`: low pass filter [default: disabled]
  - we do not want an output filter, it meets the default value
- `ch0ResetPre`: prescaler on channel CH0 [default: disabled]
  - appropriate
- `outEnablePRS`: enable peripheral reflex system [default: disabled]
- `sineEnable`: enable sinus generation [default: disabled]
- `diff`: differential output [default: disabled]

After the general initialization of the DAC peripheral, the channel used (in this case the CH0 channel) must be initialized.

- `enable`: enable [default: false]
  - we need to enable it, set it to be true

- `prsEnable`: enable peripheral reflex system [default: false]
- `prsSel`: see previous point
- `refreshEnable`: enable auto-update [default: false]
    - we update the output by writing to the register, otherwise no update is required

```
// ****************************************
// 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. Configure a timer

We initialized the timer in Exercise 5, so now we will only highlight the differences required for the specific task.

To configure the timer using its library functions, follow these steps:

- setting the peripheral clock divider
- enable timer clock
- create the parameter structure required for initialization
    - the prescaler is reset to the appropriate value
- reset the timer
- we set the TOP value
- delete any pending interrupts
- Interrupt is enabled
    - Enable at the timer peripheral
    - Enable Timer Interrupt at NVIC i.e., at the core

The above initialization process is implemented by the following program code.

We need to specify a constant FS that gives the sampling frequency. As the peripheral clock is 12 MHz, the TOP value of the counter (for FS=5 kHz, for example):

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

```
#define FS 5000 // Specify the sampling frequency in Hz
#define TIMER_DIV (12000000/FS-1)// division ratio

 // ***************************************
 // 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 initialization *
 // ******************************
 BSP_LedsInit();
```

Default values used to initialize the timer.

```
#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, / * From 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. * / \
 }
```

To handle the interrupt, the following functions must be implemented in the program code:

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

uint32_t process(uint32_t in) {
 uint32_t out;

// this is where signal processing comes in, e.g. output = input
 out = in;
 return out;
}


// ***************************************
// * TIMERIRQ *
// ***************************************

uint32_t ADC_data_in, DAC_data_out;
uint32_t TimerCnt;


voidTIMER0_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);
 }

}
```

In the timer interrupt routine:

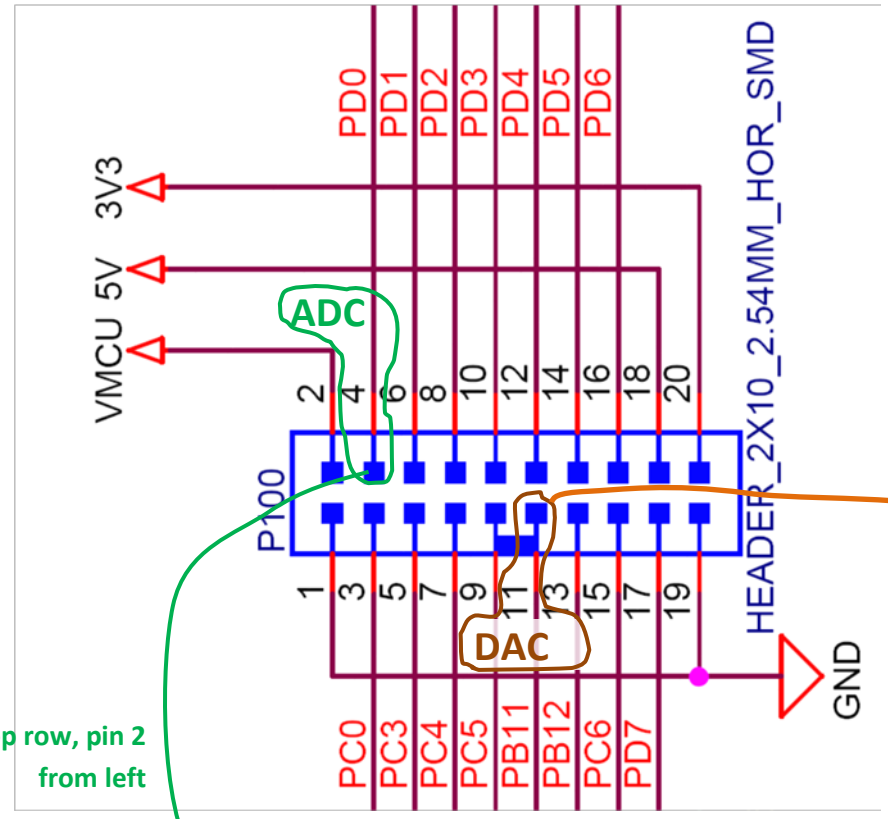
- read the result of the ADC conversion started during the previous timer IT.
    - `ADC_data_in = ADC_DataSingleGet (ADC0);`
- We are starting a new AD transformation
    - `ADC_Start (ADC0, adcStartSingle);`
- we call the function that performs the data processing
    - `DAC_data_out = process (ADC_data_in);`
- we send the final result of the data processing to the DAC
    - `DAC_Channel0OutputSet (DAC0, DAC_data_out);`
- Clear the Timer Interrupt flag
- A LED flashes every second: it indicates operation

6. Mapping input and output legs

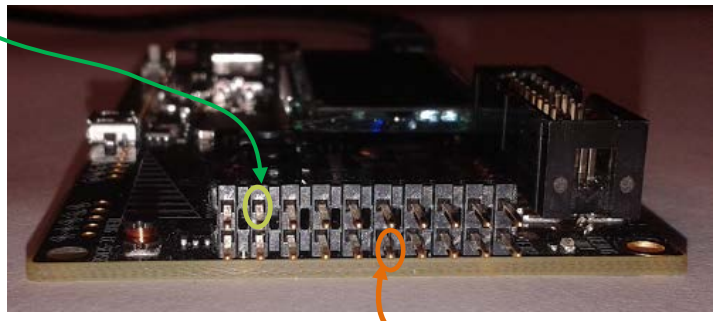
The output and input pins are located on pins PB11 (DAC0 output) and PD0 (ADC_CH0 input) according to the following documentation:

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.

The pins on the microcontroller expansion connector can be identified as follows:



top row, pin 2 from left



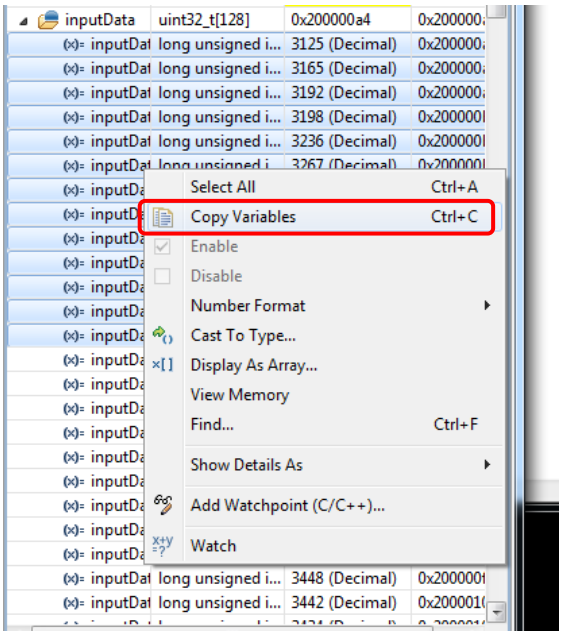
bottom row, pin 5 on the right

7. Display data

During the measurement, the sampled and processed data will be saved in separate arrays, and by exporting the data in a file, we can also display the data graphically using a Python script. The script is available on the subject website for exercises.

The steps are as follows:

- Select the numeric values to save to the file in the debug window that contains the variables
  - Select the first item, then press the shift key to select the last item.
- Right-click and select Copy Variables (saves the data to the clipboard).
- Open a text file, copy the copied data one by one with CTRL + V, and save it as a directory called ADCdata.txt to display_data.py.
- Double-click to run display_data.py
  - Note: You can change the file name by editing display_data.py. Right-click display_data.py Open with: Notepad ++
- The window must be closed before running the display program again.



Tasks

1. Display input data:
    a. Continuously save the input data in a circular array of 128 samples.
    b. Run the program
    c. keep touching the ADC input with your finger,
    d. stop the program (then you can release the input),
    e. save the data to the ADCdata.txt file,
    f. display by running display_data.py.
2. Generate a saw signal and output it to the DAC
    a. Generation of the saw signal can be achieved by incrementing a phase variable. The value of the variable is incremented by a given value at each sampling rate and output directly to the DAC.
    b. Help: The 12-bit DAC expects a value of 0… 4095. If a variable is incremented by dX during each sampling, the full range is run over a sampling rate of 4095 / dX. If you want to generate a saw signal with frequency f0, it runs from 0 to 4095 during the FS / f0 rate. So:

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

    x = x + dX;
    Let the frequency be f0 = 100Hz
    c. Measure the signal back: connect the DAC output to the ADC input using the wires assigned by the practice leader. **ATTENTION!!!**Check several times very carefully to see if you are connecting appropriate pins (count down several times, check with your partner as well) because accidentally connecting two outputs can cause board damage. You may want to connect the cable to the DAC output first, as it is harder to find.
    d. Control saturation of the signal if it is greater than 3000 LSB
    e. Control the signal to saturation if it is less than 100 LSB
    f. Generate a triangle signal.
    g. Perform exponential averaging of the measured signal such that the cut-off frequency is fc = 50 Hz. The interference voltage of the mains is measured.
        i. as a reminder:
        $y_n = y_{n-1} + (1-\alpha)(x_n - y_{n-1})$
        where$\alpha = e^{-\frac{1}{T \cdot f_s}}$
        the time constant:$T = \frac{1}{2\pi f_c}$

Appendix: Python code that displays data

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

fname= 'ADCdata.txt'
fs= 5000.0

print "fs =% f" % (fs)
Ts= 1/fs
main= open(fname)
ln= fO.readline ()
arr= np.array ([])

while not(ln==''):
 tab1 = ln.find ('\ t') # Find your first tab
 tab2 = ln.find ('\ t', tab1+1) # find second tabulator
 tab3tab = ln.find ('\ t', tab2+1) # if it is separated by a plain tab
 tab3par = ln.find ('(') # if the type is specified with a zodiac sign
 if (tab3par<0):
 tab3par = tab3tab
 tab3 = min(tab3tab, tab3par)
 numberRaw = ln [tab2+1: tab3] # the raw number
 if flax(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 = flax(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 ()
```