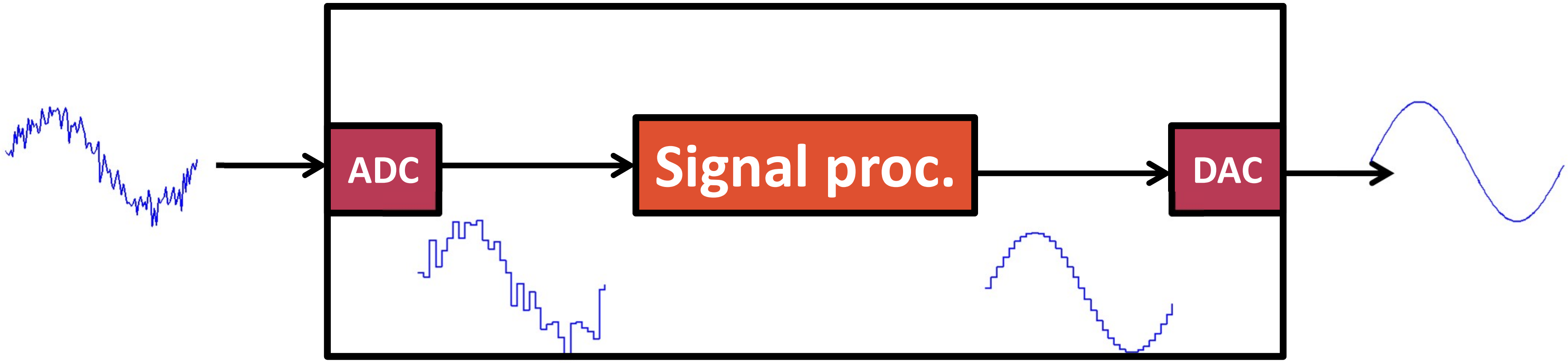
Embedded Software Development
2024. 11. 25.

SW architecture of data processing systems

Méréstechnika és
Információs Rendszerek
Tanszék

SW architecture of data processing system

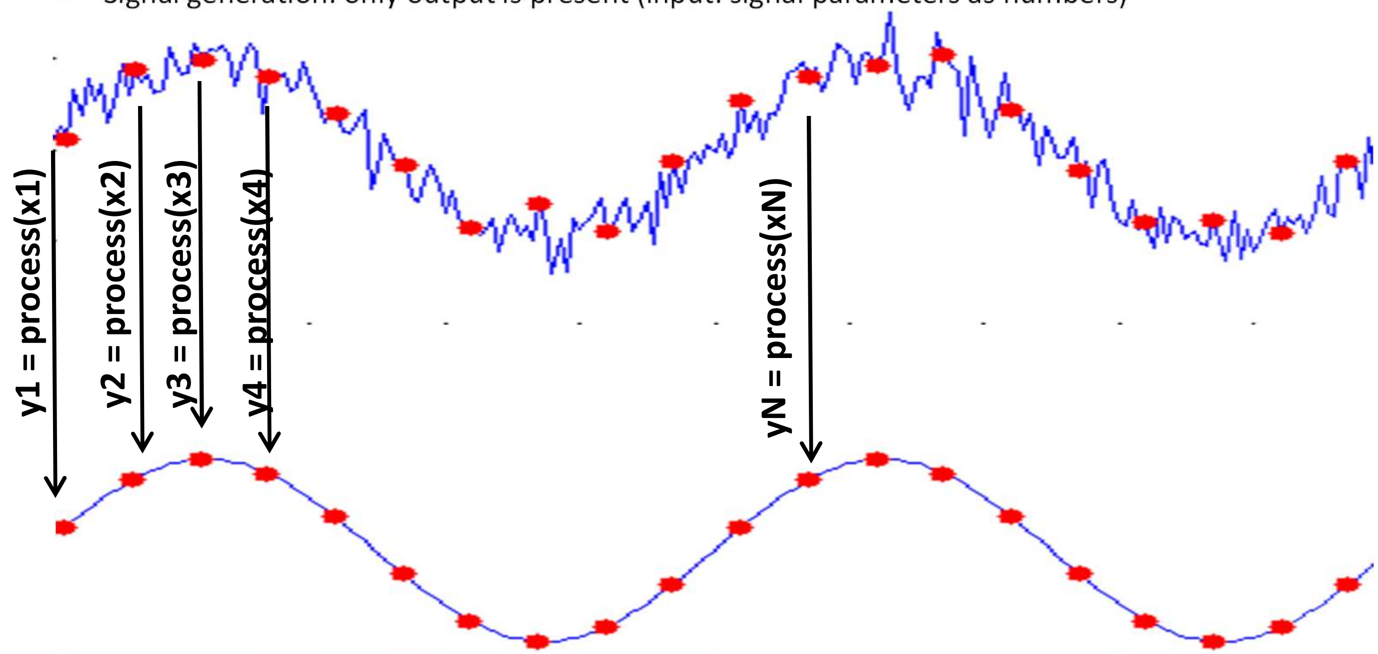
- Model of operation:



- Based on data access and data handling two main architectures can be distinguished:
  - Sample-based data processing
  - Block of data-based data processing

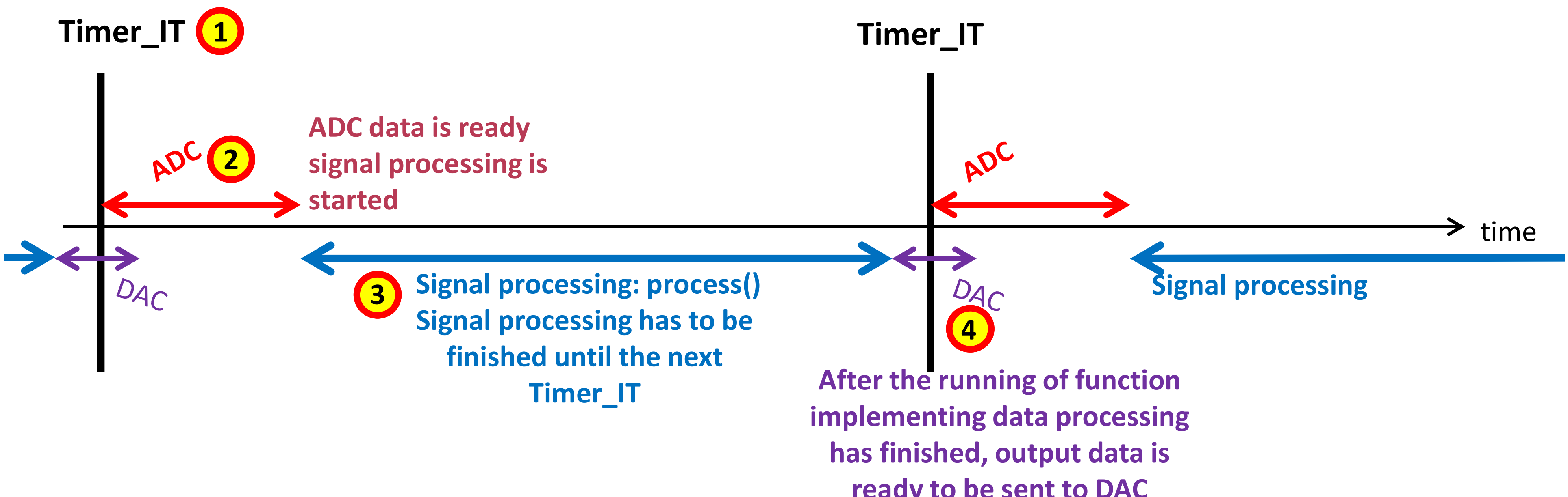
Sample-based data processing

- In every sampling time instant the signal processing routine processes a new sample, i.e. receives a new sample and generates a new output (an other sample)
  - Remarks, notes:
    - Samples my be received (and generated after processing) from more, simultaneously processed signals
    - In general the presence of input/output signal is not required in a data processing system
      - Measurement of signal parameter: only input is present (output is a number, representing e.g. amplitude)
      - Signal generation: only output is present (input: signal parameters as numbers)



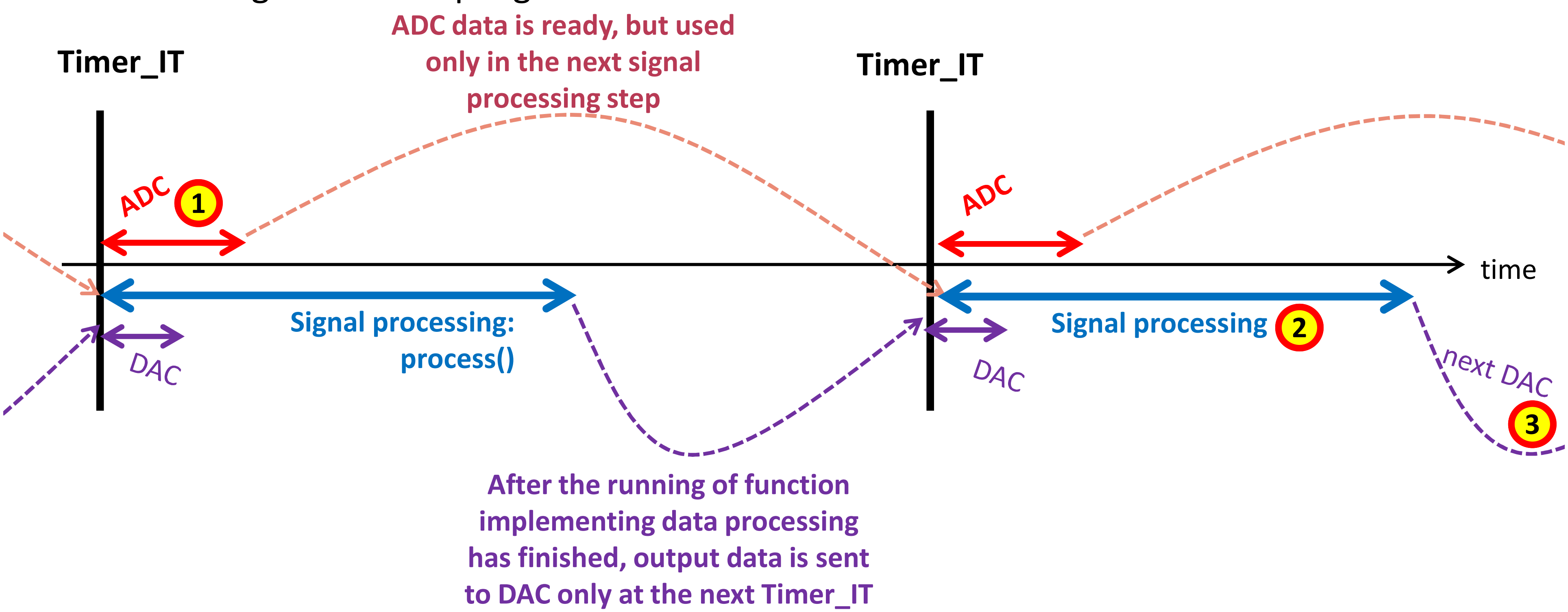
Study of data processing system (1/a)

- At the beginning of Timer_IT, analog-to-digital conversion (ADC) is initiated
- Simple but not resource saving method: end of ADC is waited and just after digital signal processing (DSP) is started. Since DSP has to be finished until the next Timer_IT, time of ADC is wasted.
    - Digital to analog conversion (DAC) is less critical in general since it only has to be initiated and then performed in the background. The result appears at the output independently from us.
- If timing is not critical this simple method can be well applied
- Delay = AD conversion + signal processing + DA conversion
- Processing time < sampling time – AD conversion



Study of data processing system(2/a)

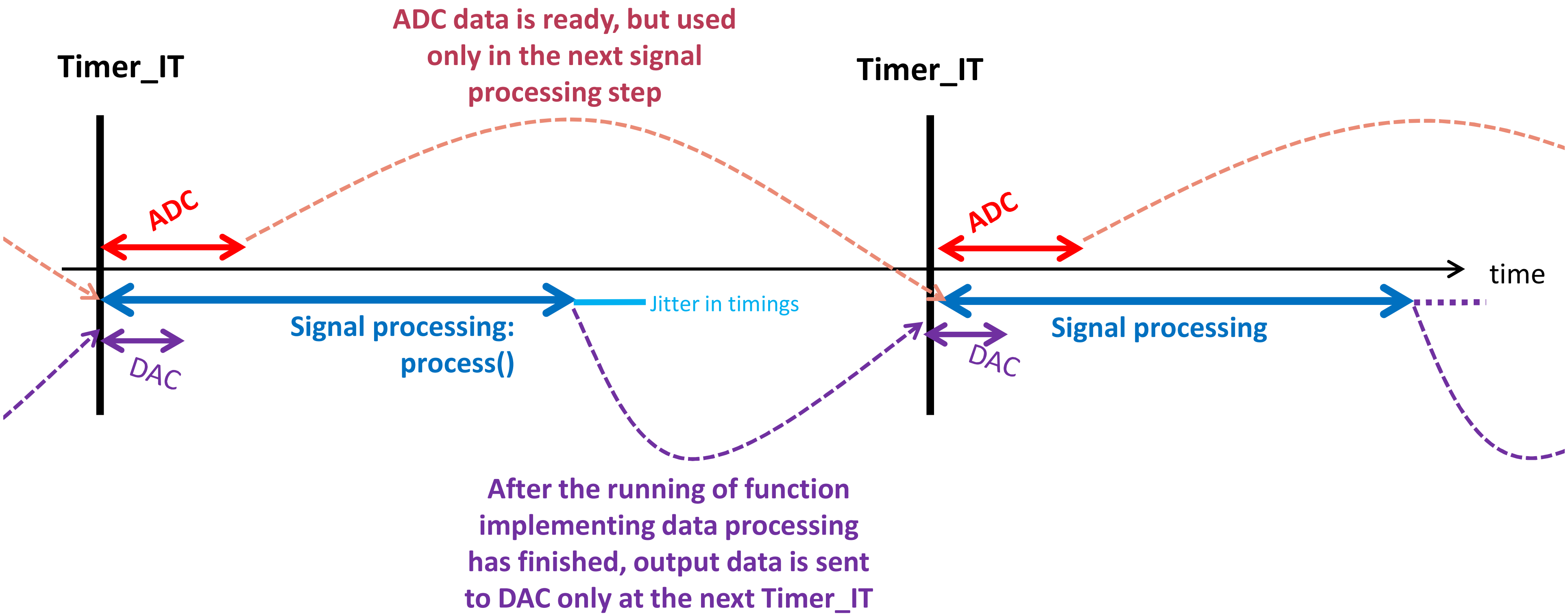
- Signal processing architecture with well-defined timings :
  - Timer-based IT
  - DAC data generated in previous DSP step is being converted into the analog domain
  - Reading ADC result initiated at the previous Timer_IT
  - Initiating new ADC
  - Running signal processing algorithm that processes ADC result
- Delay = 2 * sampling time + DAC delay
- Processing time < sampling time



Study of data processing system (2/b)

- Processed data is ready after return from signal processing function, therefore DAC is possible. But DAC is performed only after next Timer_IT because this way the operation remains deterministic: in some cases runtime of signal processing function may change (due to parameter change or button pushed, etc.) therefore DAC data timing could suffer from jitter. **At a price of extra delay, deterministic behavior is assured.**

Note, that this is the same approach as the one before with more explanation focusing on deterministic timings



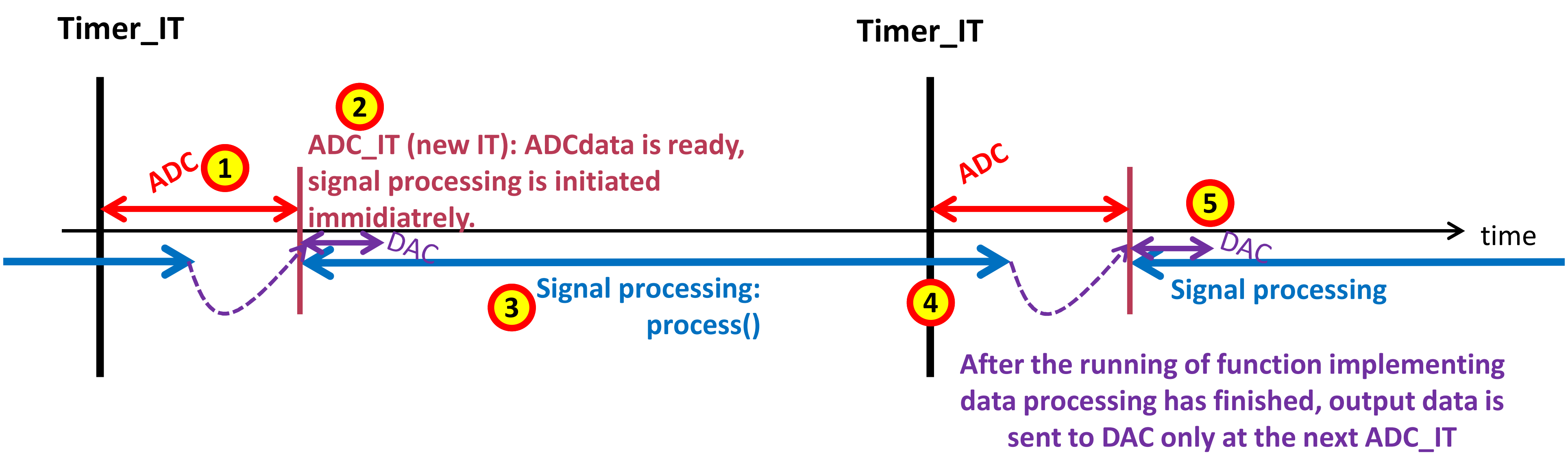
Sample-based signal processing

- Pseudo-code (workframe and processing function):

```
Timer_IT(){
    // result of previous signal processing step is used (sent to DAC)
    writeDAC(ADCout);
     // reading ADC sample (result of ADC initiated at the previous Timer_IT) into ADCin
    ADCin = readADC ();
     // initiation of sampling and ADC of new data: this data is stored in ADC data register
    //and read after next Timer_IT: see ADCin = readADC();
    sampleADCstart();
    // reading ADC sample (result of ADC initiated at the previous Timer_IT) into processing function
    ADCout = process(ADCin);
}


process(data_in){
     .... Some kind of DSP....  E.g.: data_out = data_in * data_in; // simple squaring as an example
     return data_out;
}
```

Study of data processing system (3/a)

- At the beginning of Timer_IT, analog-to-digital conversion (ADC) is initiated
- Readiness of ADC data generates an IT and signal processing is just then initiated. This way delay is shorter but SW architecture is more complicated:
  - A new IT appears in the system
  - It has to be assured that **Timer_IT be capable of interrupting signal processing**: based on the figure below, it is possible that signal processing is going on while sampling should be initiated. Sampling cannot be waited therefore signal processing must be interrupted to initiate ADC.
- Delay = ADC delay + sampling time+ DAC
- Processing time < sampling time



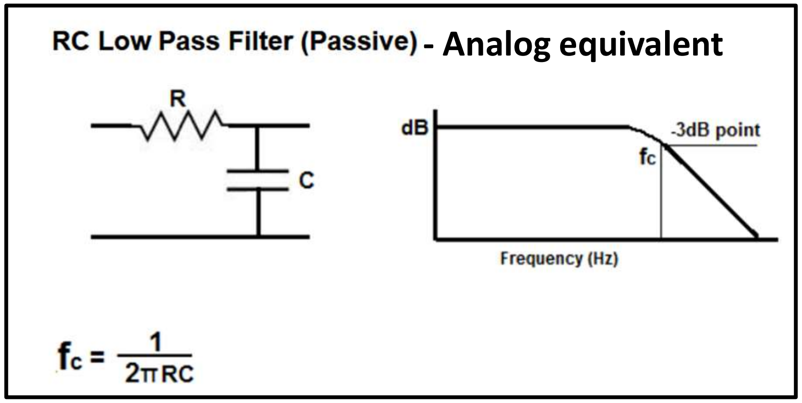
Example: first order IIR filter (Giant Gecko)

- Handling of ADC ands DAC, timing

```
void TIMER0_IRQHandler(void){
        DAC_Channel0OutputSet(DAC0, DAC_data_out);
        ADC_data_in = ADC_DataSingleGet(ADC0);
        ADC_Start(ADC0, adcStartSingle);
        DAC_data_out = process_Filter(ADC_data_in);
        TIMER_IntClear(TIMER0, TIMER_IF_OF);
}
```

Example: first order IIR filter (Giant Gecko)

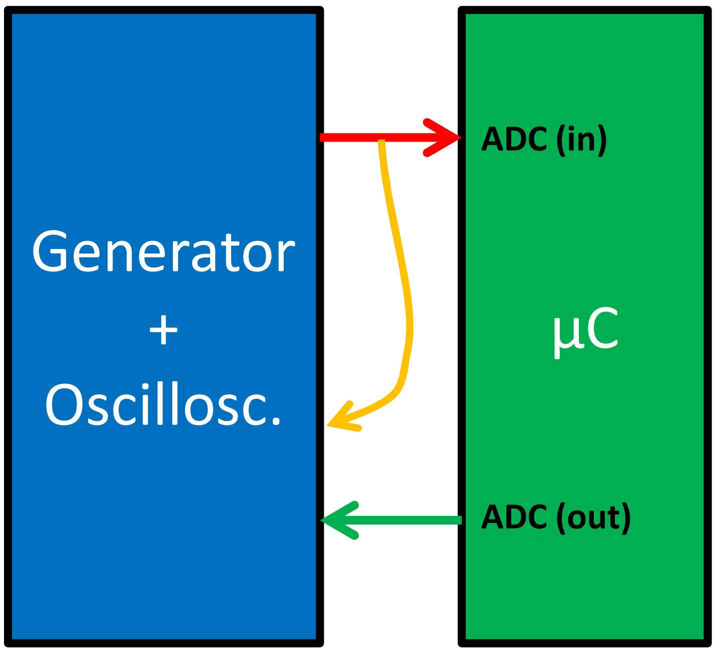
- Data processing: first order filtering
  - Time constant: 1 msec
  - Will be taught later

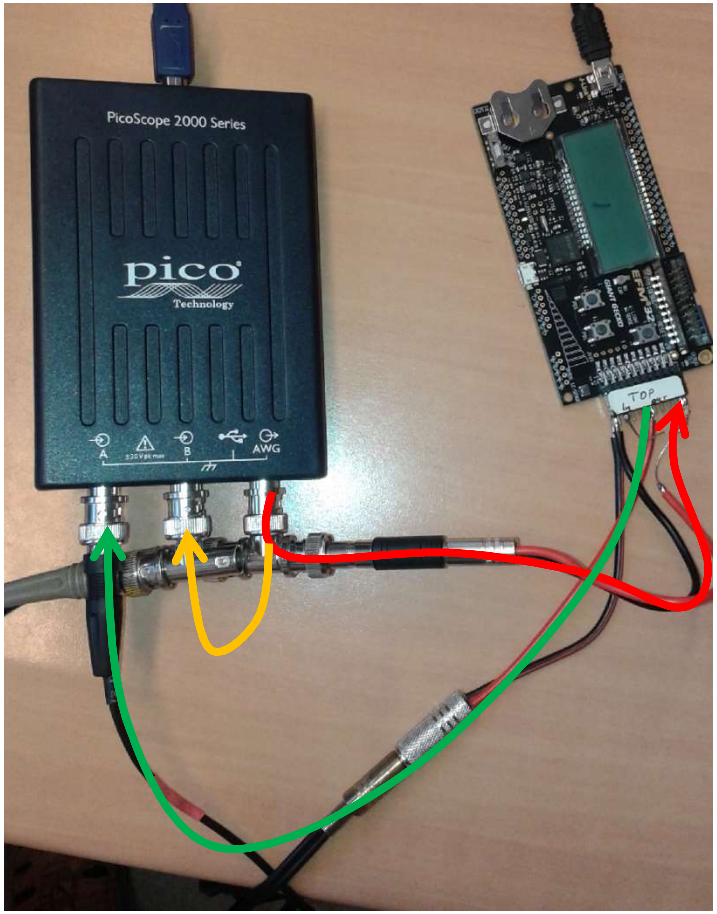



```
uint32_t process_Filter(uint32_t data_in){
        uint32_t data_out;
        float data_in_f;
        float alpha = (1- 0.9802);
        static float y; // static variable since its value has to be preserved between
                           function calls

        data_in_f = (float)data_in; // format conversion
        y = y + alpha*(data_in_f - y); // exponential averaging
        data_out = (uint32_t) y; // format conversion

return data_out; // return of result of filtering
 }
```

Measurement setup

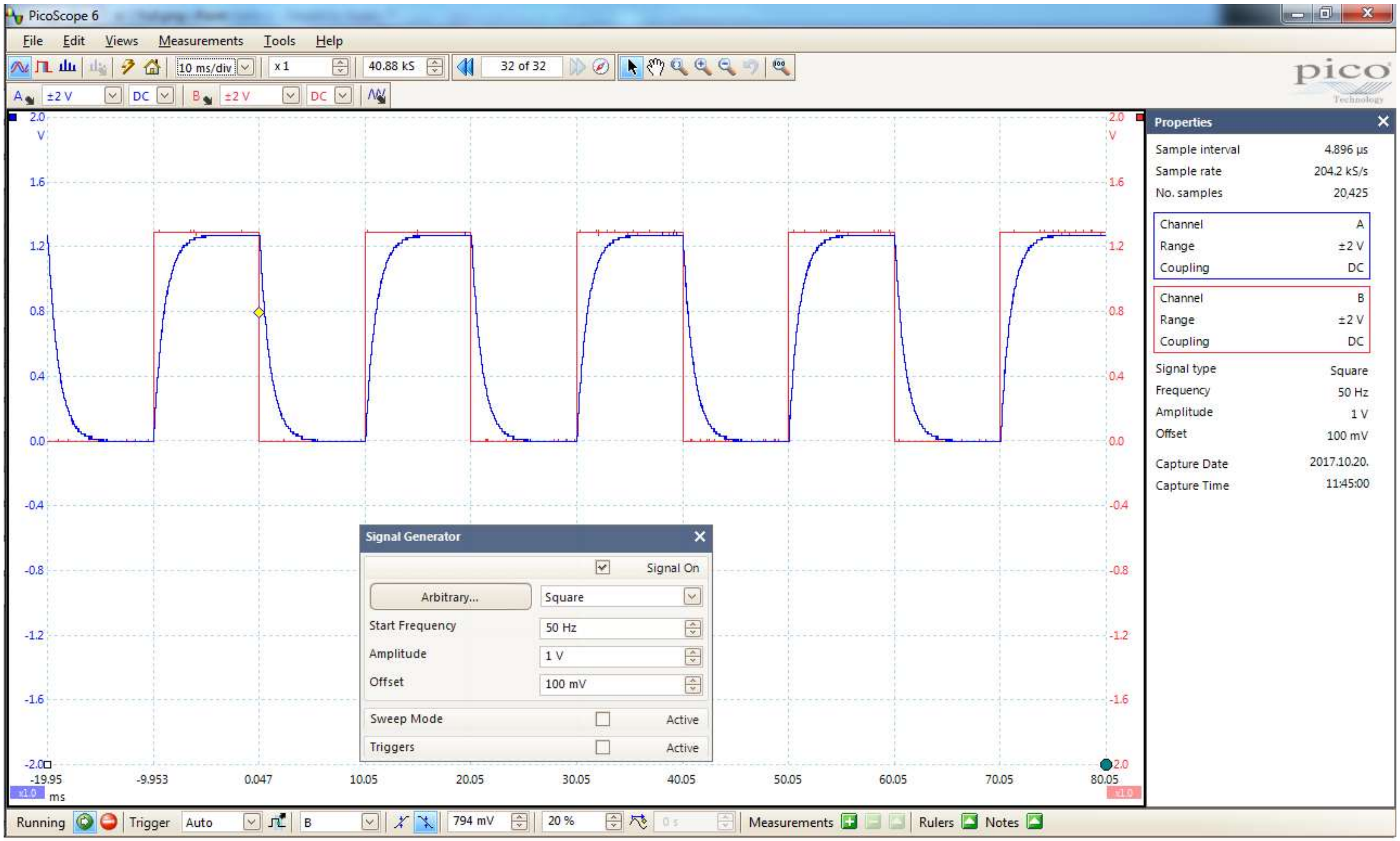
- Measurement: USB oszcilloscope and signal generator (PicoScope)
- Signal generator: board connected to ADC and trigger input





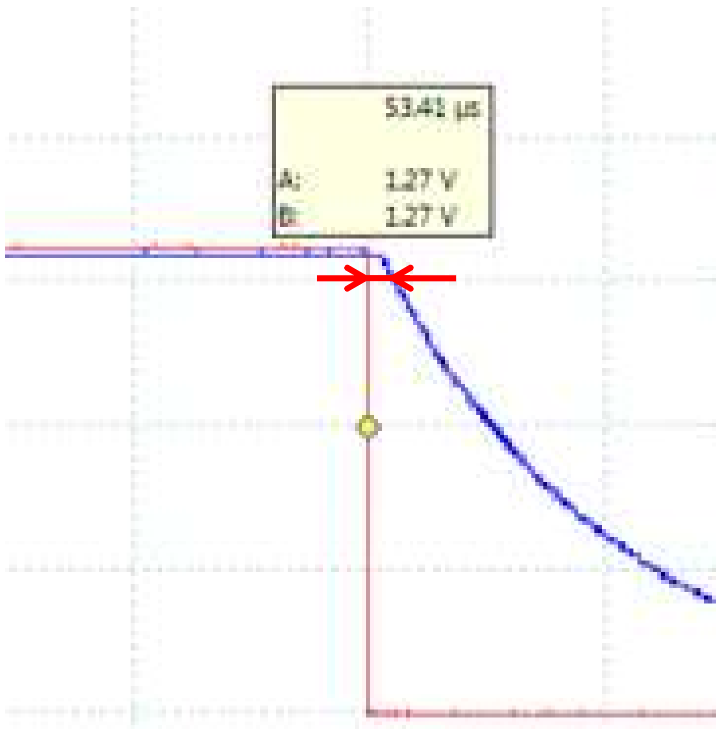
Measurement results

- Excitation: 50Hz square, 0V DC, 1.27V peak value
- Red: excitations (used also as a trigger signal)



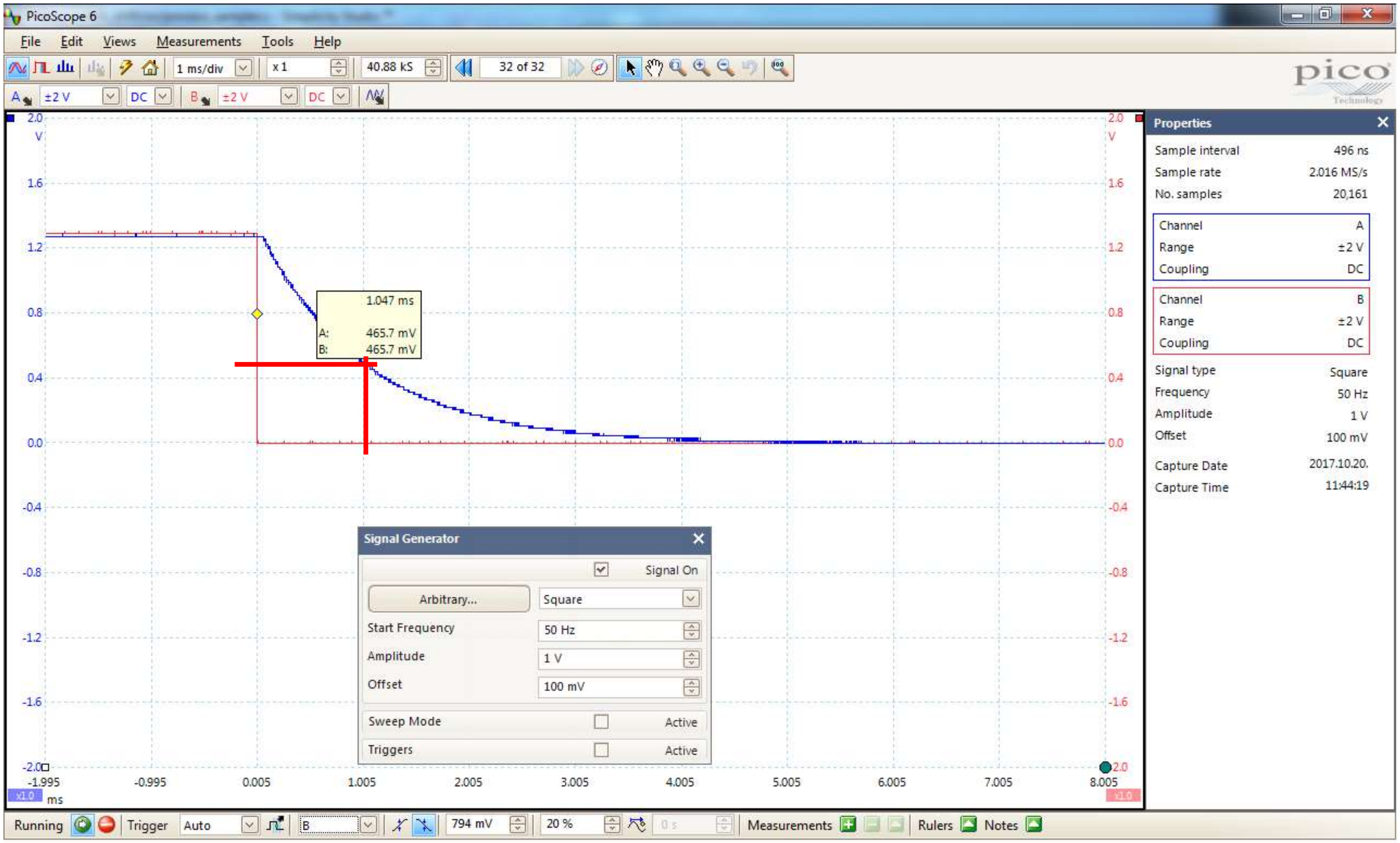
Measurement results

- Delay: ~40µsec (2 samples at 50 kHz, 2*1/50000 sec)
- Source of 2-sample delay:
  - Input data is processed one sampling time later
  - Output data is sent out one sampling time later



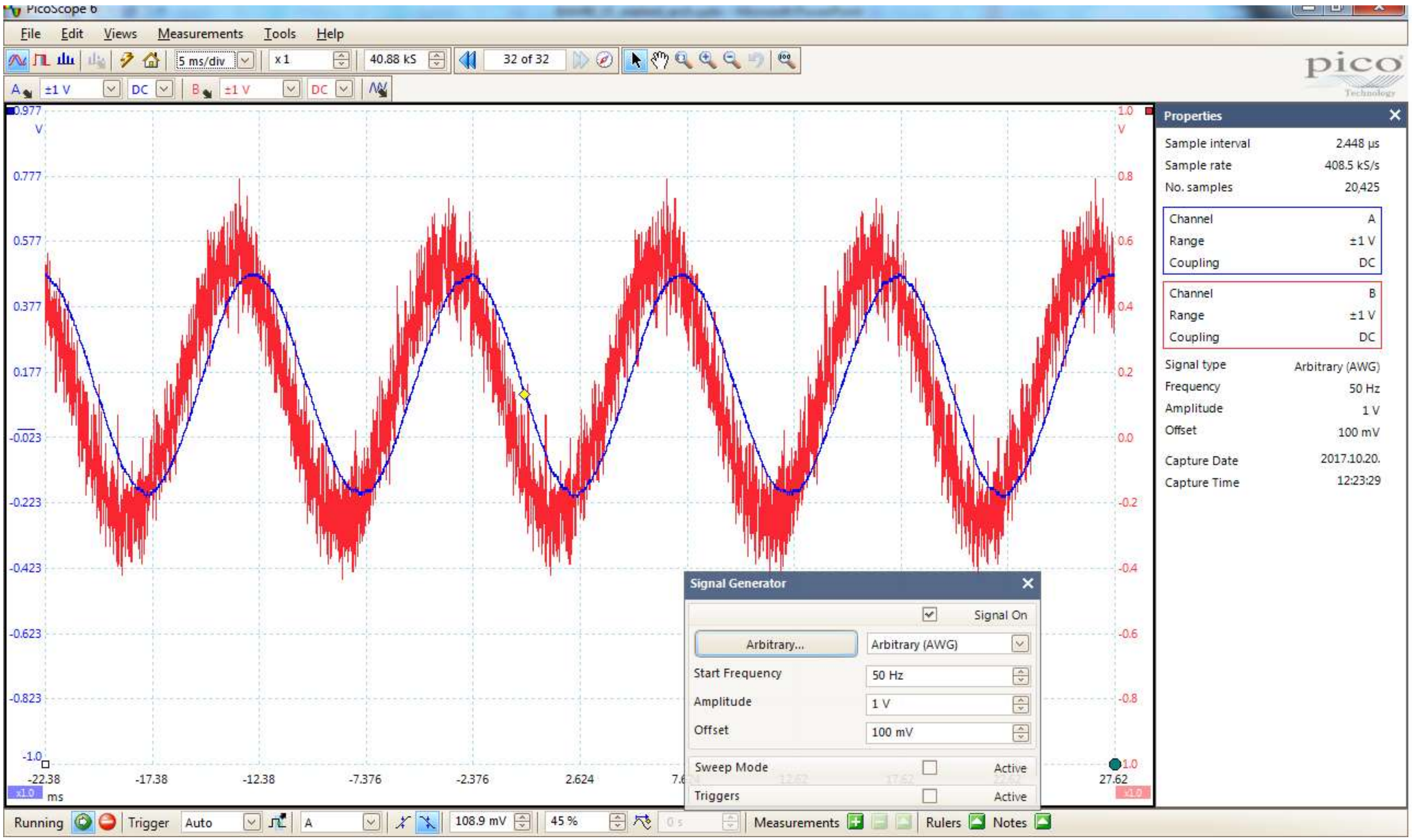
Measurement results

- Time constant: 1.27V*e^{-1}=467mV: where level reduced by factor 1/e
- 40µsec delay be subtracted: as expected (1.047ms-0.040ms)



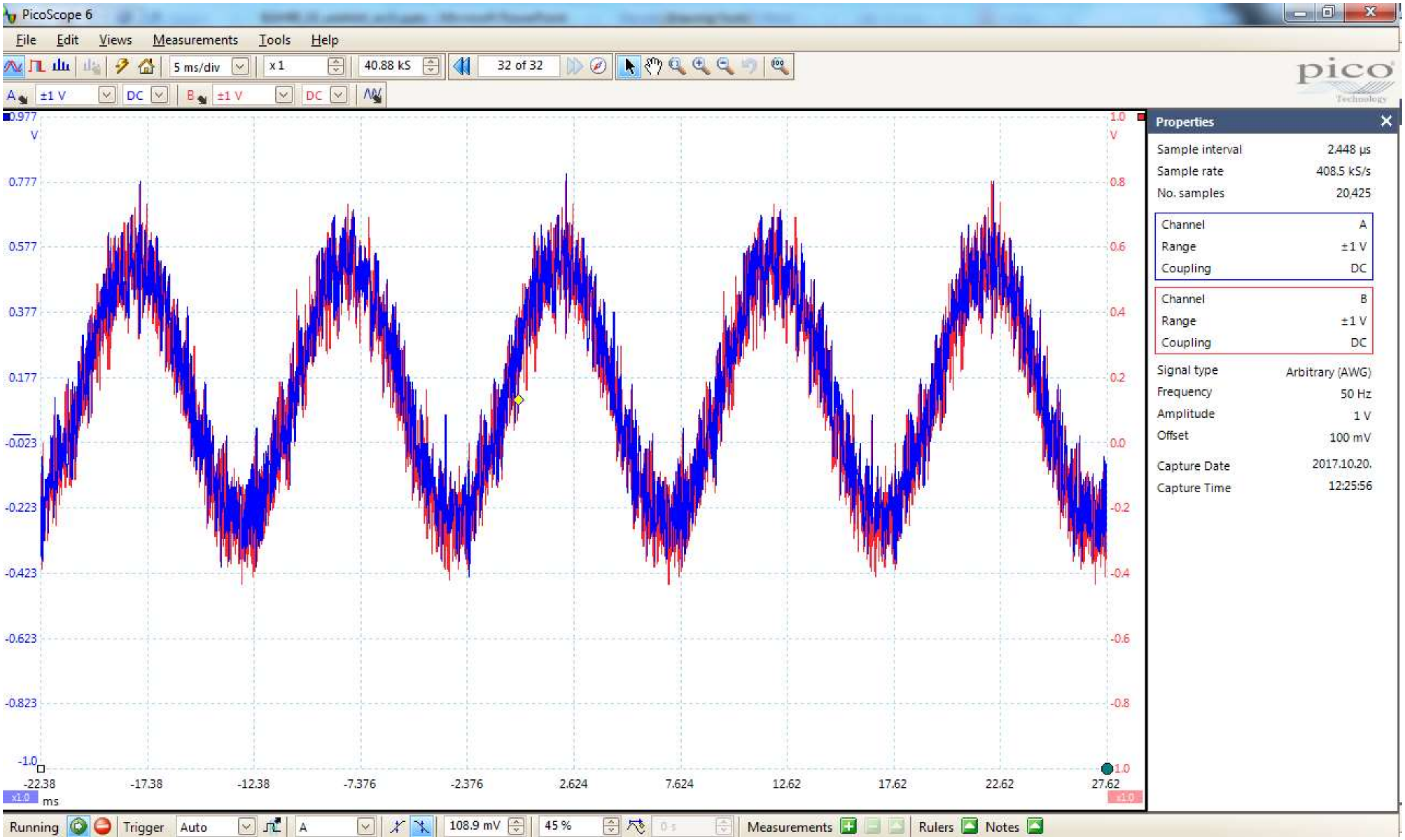
Measurement results

- Filtering of noisy signal (blue signal is filtered)



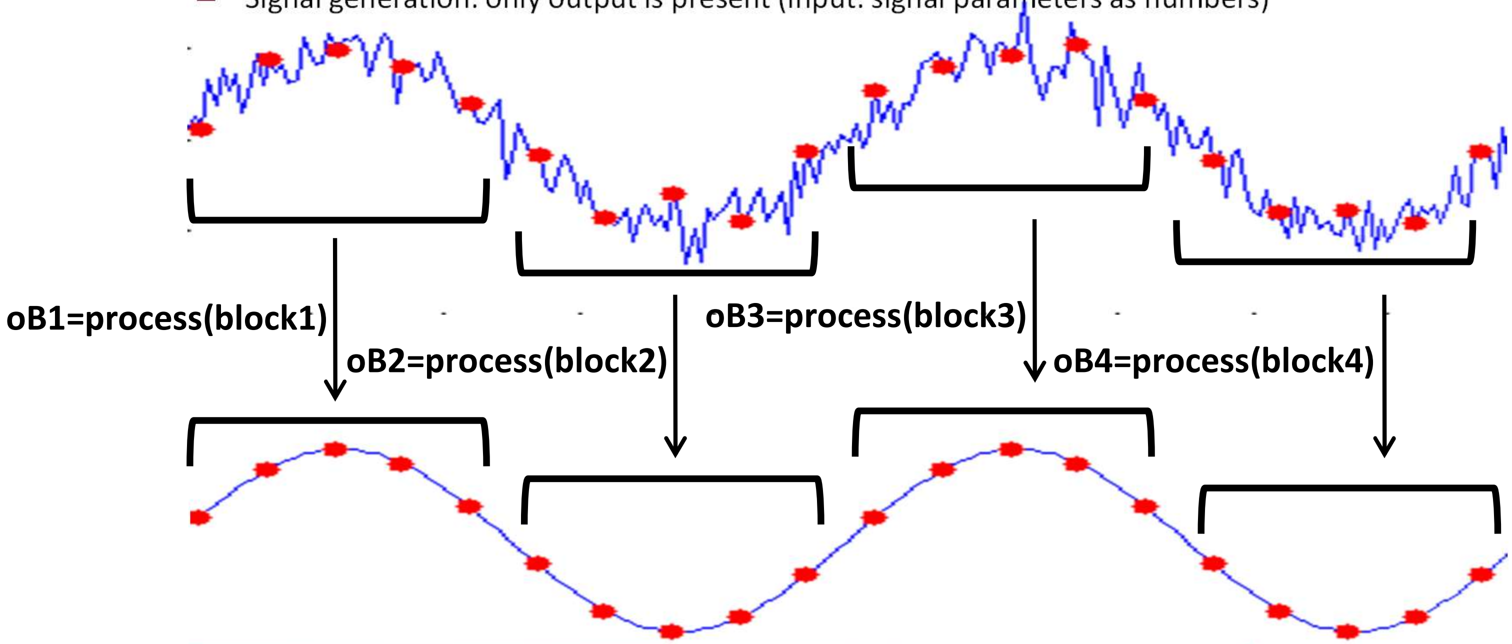
Measurement results

- Filtering of noisy signal: filtering is off, signal just let pass the system



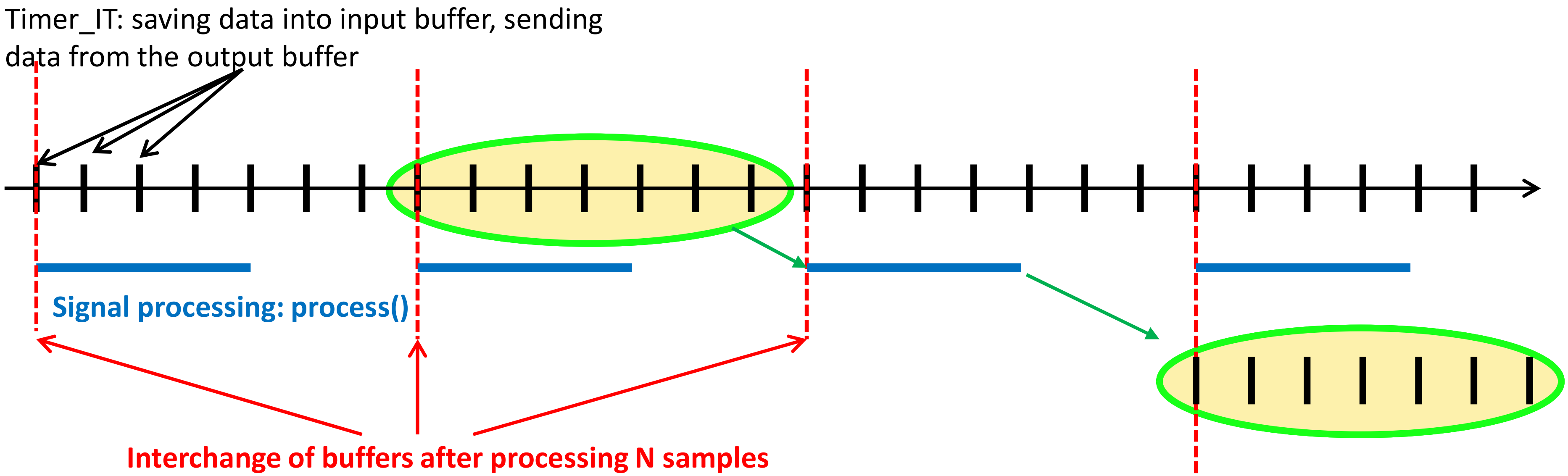
Block of data-based data processing

- Processing function receives blocks of N-data and returns with N-data blocks
- Notes, remarks:
  - Remarks, notes:
    - Samples my be received (and generated after processing) from more, simultaneously processed signals
    - In general the presence of input/output signal is not required in a data processing system
      - Measurement of signal parameter: only input is present (output is a number, representing e.g. amplitude)
      - Signal generation: only output is present (input: signal parameters as numbers)



Examination of data processing system

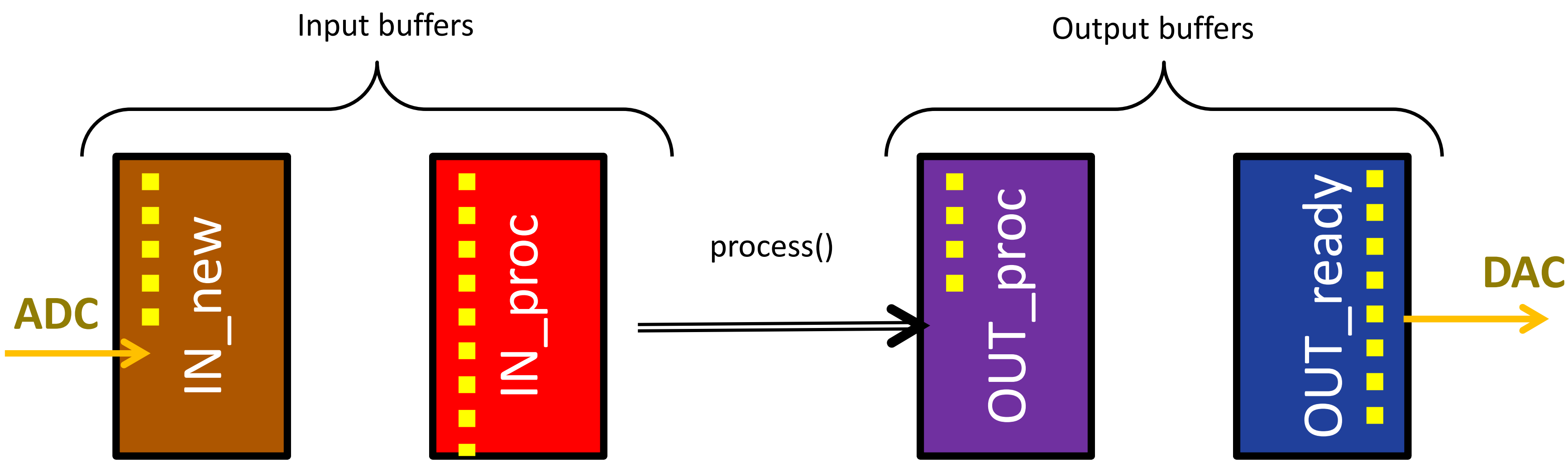
- 1^{st} priority level:
  - Saving data into input buffer and sending data from output buffer (Timer_IT)
  - Signal processing should be interruptable by Timer_IT
- 2^{nd} priority level:
  - Data processing: having N-size buffer, then signal processing has to be finished in N CLK cycles (by the time instant when input and output buffers must be interchanged)
  - Data processing is performed in the main program using a flag to indicate it: therefore Timer_IT will be able to interrupt it
  - There exist low-priority SW ITs where data processing could be implemented in: it is only advised when non-time-critical but long lasting tasks are in the program (e.g. display handling)



Block of data-based data processing

- Buffer handling

```
Timer_IT(){
    block_cntr++;
  // implementation of double buffering: when the end of the actual block is reached, interchange the buffers
    if (block_cntr==BLOCK_SIZE){
        block_cntr = 0;
        exchangeBuffer(IN_new,      IN_proc);     // interchange of input buffers
        exchangeBuffer(OUT_ready, OUT_proc); // interchange of output buffers
  }
```



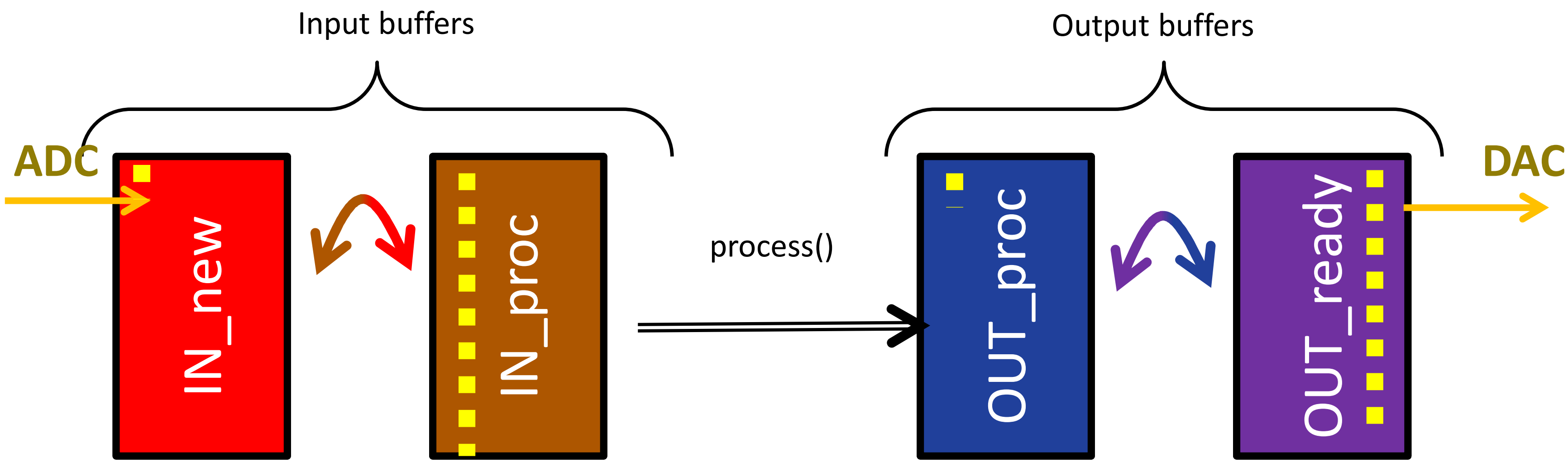
Data is continuously arriving into one of the input buffers from ADC while the content of the other buffer that is full is being processed

Processed data is continuously arriving into one of the output buffers while the content of the other buffer that is full is being sent toward the DAC

Block of data-based data processing

- Buffer handling

```
Timer_IT(){
    block_cntr++;
   // implementation of double buffering: when the end of the actual block is reached, interchange the buffers
    if (block_cntr==BLOCK_SIZE){
        block_cntr = 0;
        exchangeBuffer(IN_new,      IN_proc);     // interchange of input buffers
        exchangeBuffer(OUT_ready, OUT_proc); // interchange of output buffers
   }
```



After processing interchange of buffers is performed (status after interchange is seen)

After processing interchange of buffers is performed (status after interchange is seen)

Block of data-based data processing

Pseudo code (framework and processing function):

```
Timer_IT(){
     block_cntr++;
    // implementation of double buffering: when the end of the actual block is reached, interchange the buffers
     if (block_cntr==BLOCK_SIZE){
          block_cntr = 0;
          exchangeBuffer(IN_new,      IN_proc);    // interchange of input buffers
          exchangeBuffer(OUT_ready, OUT_proc); // interchange of output buffers
         // processing data of the full input buffer and saving processed data into new output buffer
         processStart = true;
    }
     // result of data processing is sent
     writeDAC(OUT_ready[block_cntr] );
     // reading ADC sample (result of the ADC initiated at the previous Timer_IT)
     IN_new[block_cntr] = readADC ();
     // initiating of sampling new ADC: this sample is saved into the data register of ADC
     //and read just after the next Timer_IT: see ADCin = readADC();
    sampleADC();
}

while(1){
    if (processStart)      { processBuffer(IN_proc , OUT_proc); processStart=false;}
}

processBuffer(*in_buff, *out_buff){      // data processing
     .... Some kind of DSP ....  Eg: FFT(in_buff, out_buff); // example (a): FFT
     for (ii=0; ii<BLOCK_SIZE;ii++) out_buff[ii] = in_buff[ii]* in_buff[ii]; // example (b): squaring
}
```

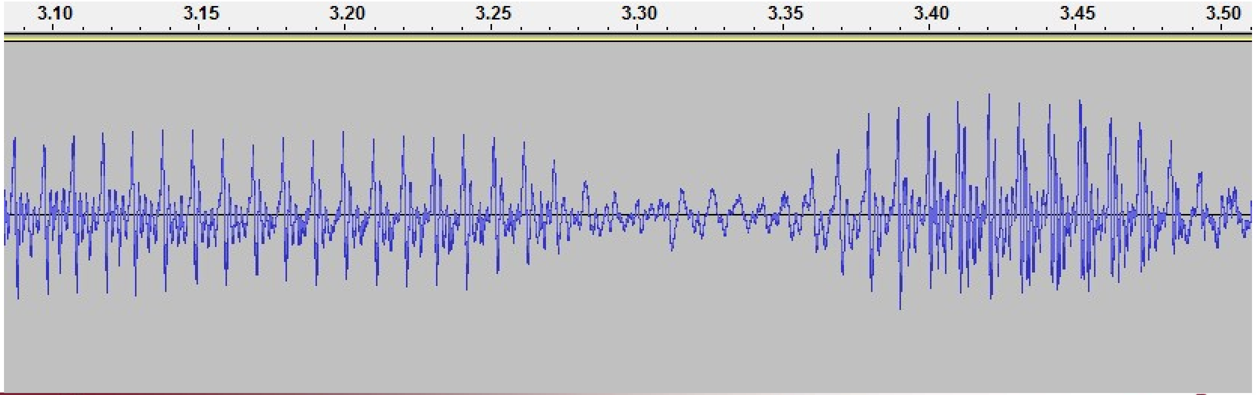
Block of data-based data processing

- Interchange of buffers:
  - Actual data interchange does not happen only the interchange of pointers

```
exchangeBuffer(dataType **buff1,  dataType **buff2){
  dataType tmp    = *buff1;
  *buff1 = *buff2;
  *buff2 = tmp;
}
```

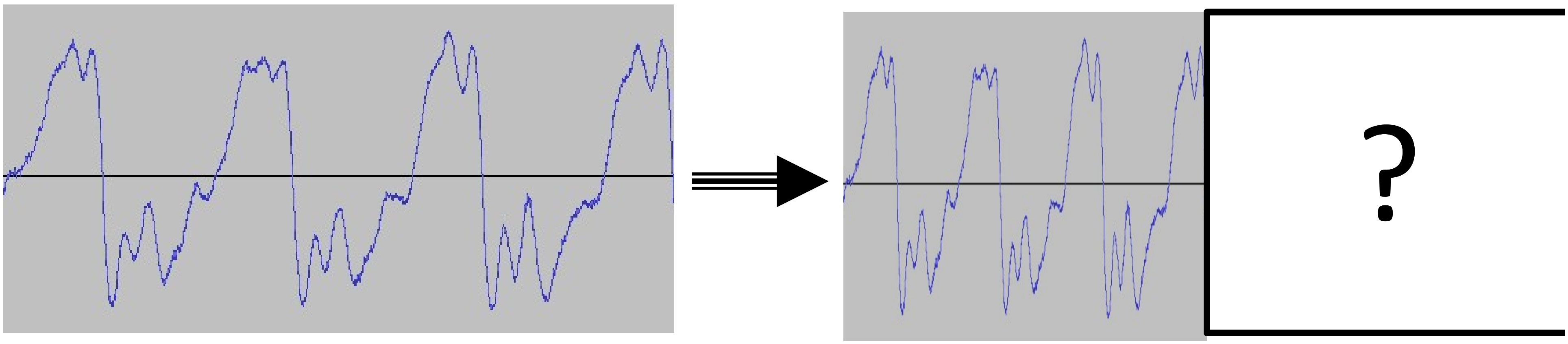
Example

- Pitch-shift algorithm in real time
  - Frequency: an objective measure of the periodicity of voice
  - Pitch: this term corresponds to frequency, but in many cases used to describe the subjective measure of how a voice or music is “high” or “low”
- Goal:
  - Increase pitch (by one octave: doubling the “speed of playback”)
  - Real-time operation
  - Speed of speech should not change
  - Simple algorithm, to be implemented by uC
- A sample function of voice (~400msec duration):



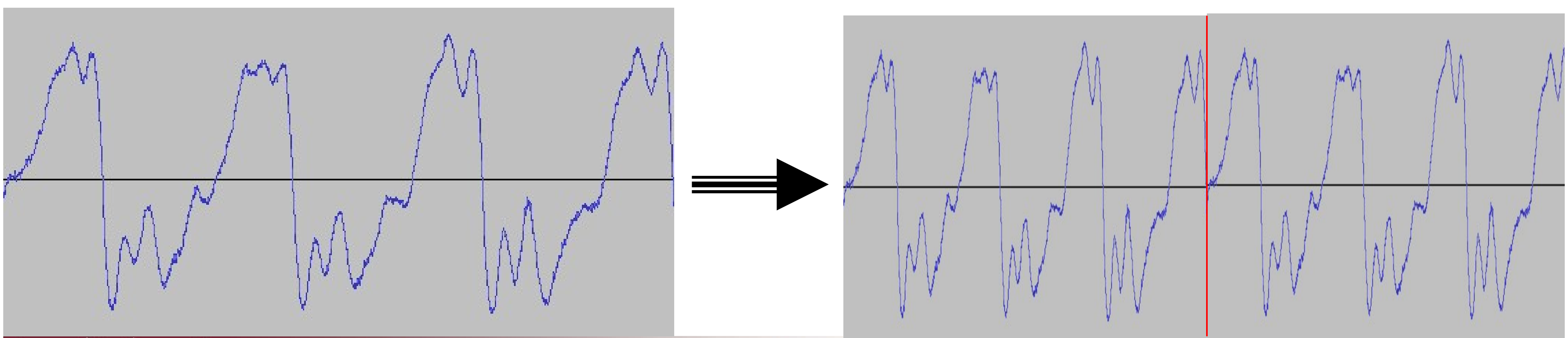
Example

- Algorithm (Step 1)
  - Pitch can be changed by increasing/decreasing play speed
    - One octave increase requires double play speed
  - Problem:
    - The signal will be shorter (since played faster)
    - In real-time cannot work since signal “disappear” too early: buffer is emptied out fast



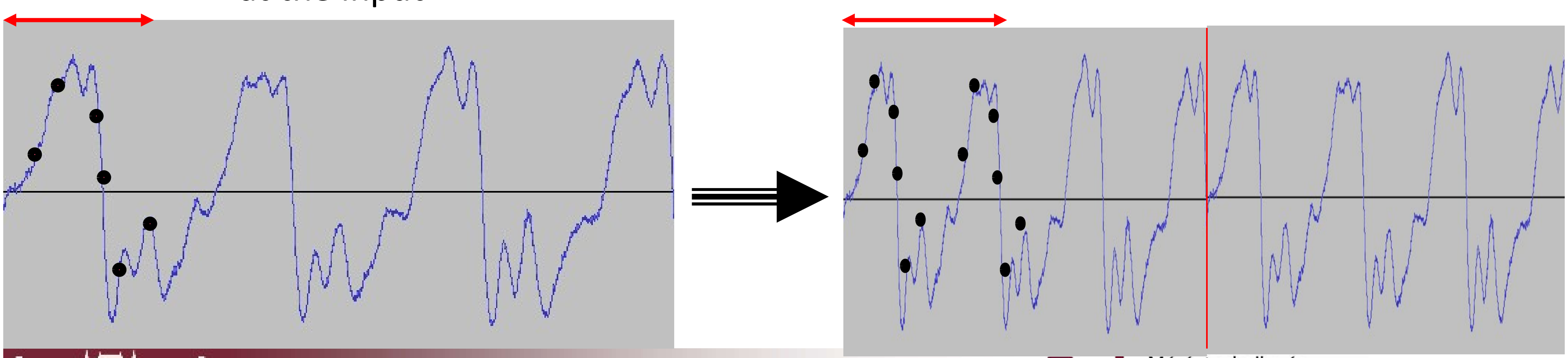
Example

- Algorithm (Step 2)
  - Repeat the whole voice signal→ signal length will not change, therefore can be played in real-time, the buffer will not be emptied too early
  - Problem 1: The cut pieces may not fit correctly (see example: signal would be (expected to be) decreasing after the red line however after repeating it, the signal is increasing after red line). Solution:
    - Windowing the signal providing a smooth transition between the original and repeated one: not too complicated solution but not applied in this example
    - This problem will not cause a subjective error (one person will not hear it as a bad quality voice) and the result is heard to be good. Out ear will "smoothen" it…



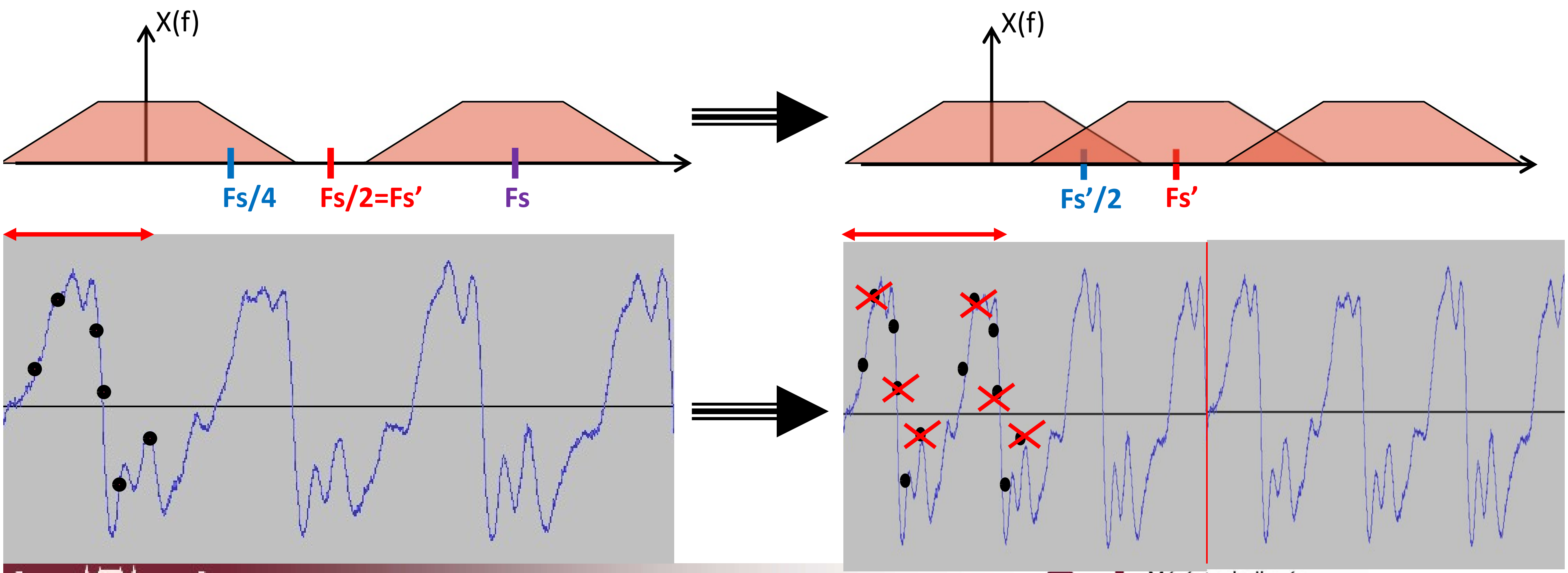
Example

- Algorithm (Step 2)
  - Repeat the whole voice signal→ signal length will not change, therefore can be played in real-time, the buffer will not be emptied too early
  - Problem 2: The sampling frequency of the output signal is twice that of the input one (see figure below)
    - In a DSP system it is required to have the same sampling frequency at the input and output as well. If not, then the frequency difference somehow has to be corrected (e.g. decimation, interpolation, resampling, etc…) since that is a basic requirements for the HW and SW elements of the system. Otherwise some serous difficulties arise to cop with.
    - It is required to have the same amount of data per time unit at the output than at the input



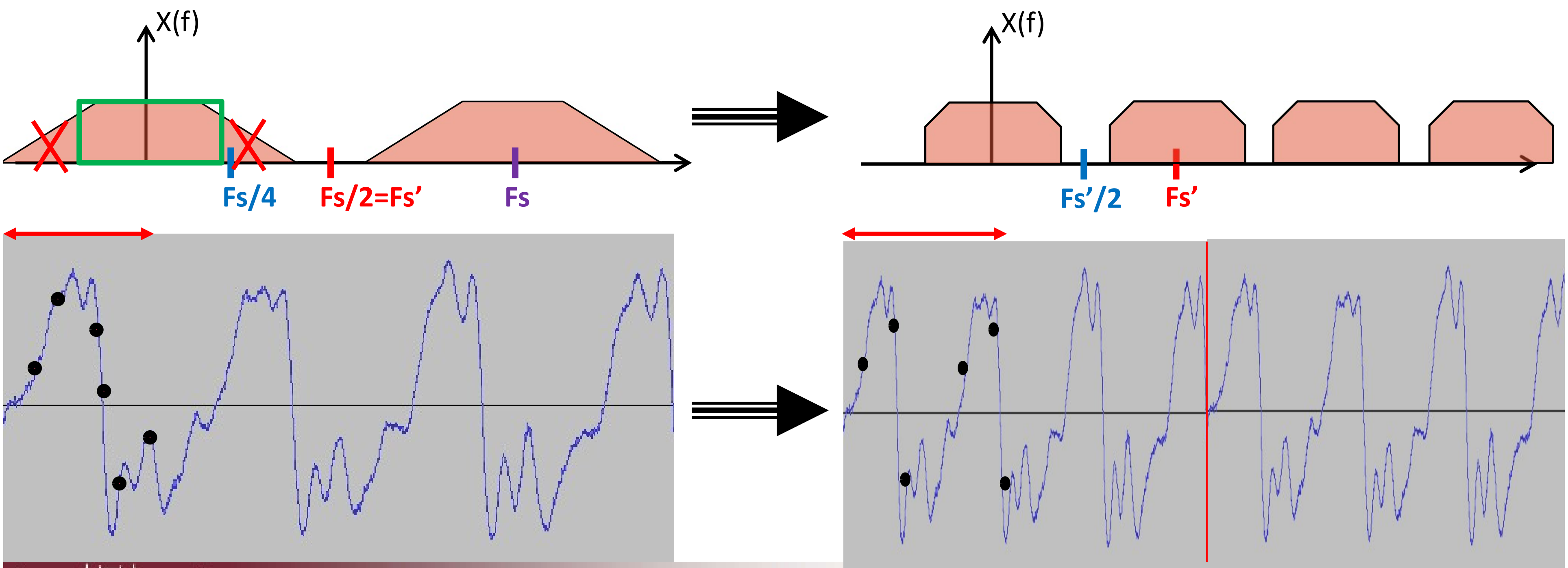
Example

- Algorithm (Step 3)
  - At the output side the number of samples are halved (decimation by factor 2)
  - Simple (but not too good solution): removing every second sample
    - Sampling frequency is halved the spectrum of sampled signal is repeated not by Fs, but Fs'=Fs/2 -> overlapping may occur in spectrum, if the Nyquist sampling criteria is just met since no possibility for decimation in that case.



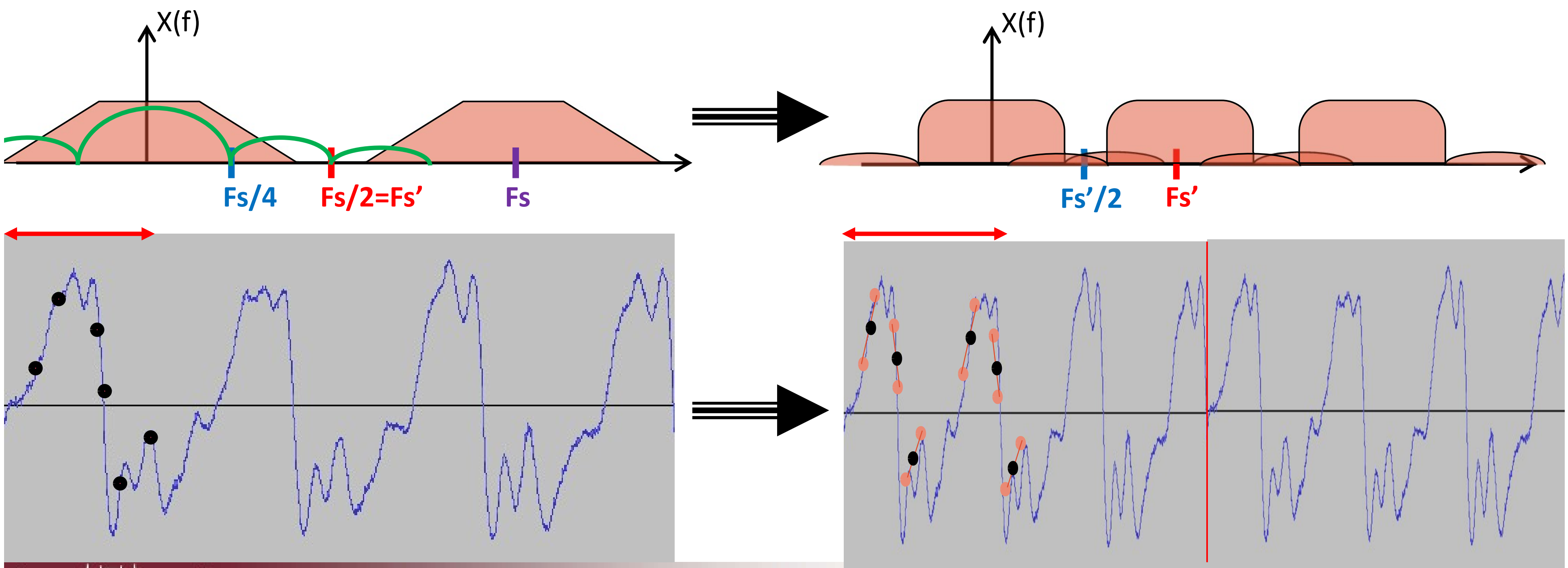
Example

- Algorithm (Step 3)
  - At the output side the number of samples are halved (decimation by factor 2)
  - Correct solution: signal components above Fs/4 are removed by a decimating filter (=simple low-pass filer of cutoff frequency Fs/4) and samples are removed just after filtering



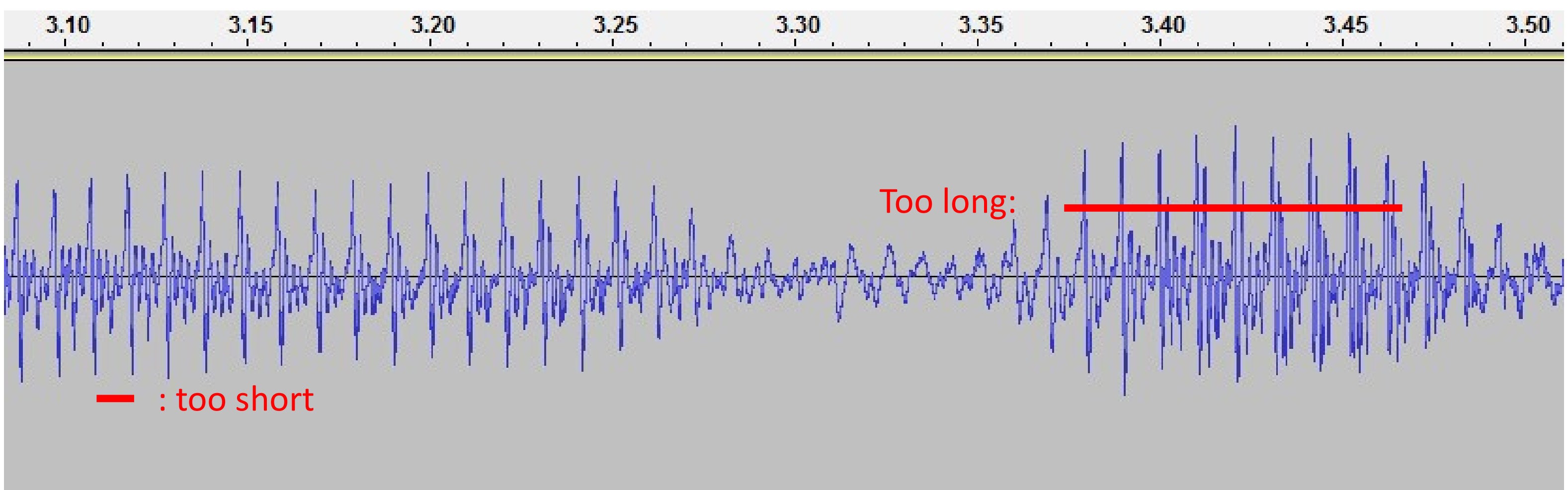
Example

- Algorithm (Step 3)
  - At the output side the number of samples are halved (decimation by factor 2)
  - **Compromise solution: take the average of two neighboring samples and keep only that one instead of the two (this solution is used in the example, averaging is a simple filtering by sin(x)/x frequency response)**



Example: implementation of pitch shift

- Block of data-based data processing:
  - Signal processing shall be block-based since a whole block of data has to be repeated
  - Sampling frequency: 25 kHz (audio signal, our choice)
  - Determination of block size:
    - Not to be too short: inside one tone more than one period should be present to be stored in the data buffer: be longer than 10ms, i.e.: 250 samples
    - Not to be too long: not longer than one whole (to avoid repeatiton of whole): be shorter than 100ms, i.e.: 2500 samples
    - Our choice: 30ms: 0.03*25000 = 750 samples



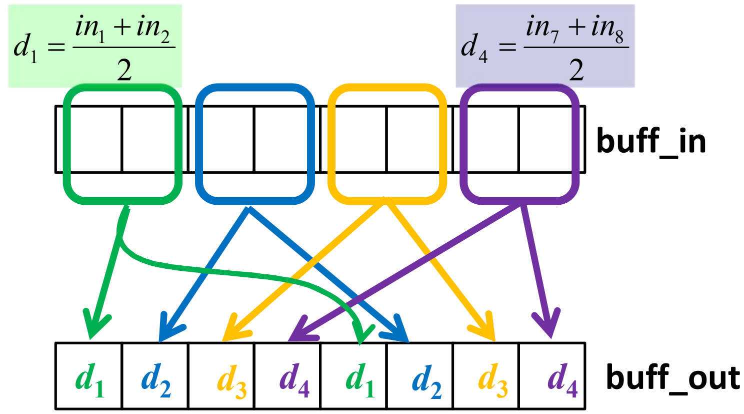
Example: implementation of pitch shift

- Code part of ADC and DAC handling (Giant Gecko)

```
void TIMER0_IRQHandler(void){
        block_pos_cntr++; // position inside the buffer
        if (block_pos_cntr>=N_PITCH_DATA){ // actual buffer is full
                block_pos_cntr = 0;
                // double buffering: buffer storing processed data is full,
                // its content is sent to DAC,
                // the other output buffer will store the newly processed data
                exchangeBuffer(&IN_new,    &IN_proc);  // interchange of input buffers
                exchangeBuffer(&OUT_ready, &OUT_proc); // interchange of output buffers
                processData = true; // a flag is used to indicate that processing can be started
        }
        DAC_Channel0OutputSet(DAC0, OUT_ready[block_pos_cntr]);
        ADC_data_in = ADC_DataSingleGet(ADC0);
        IN_new[block_pos_cntr] = ADC_data_in;
        ADC_Start(ADC0, adcStartSingle);
        TIMER_IntClear(TIMER0, TIMER_IF_OF);
}
```

Example: implementation of pitch shift

- Signal processing:

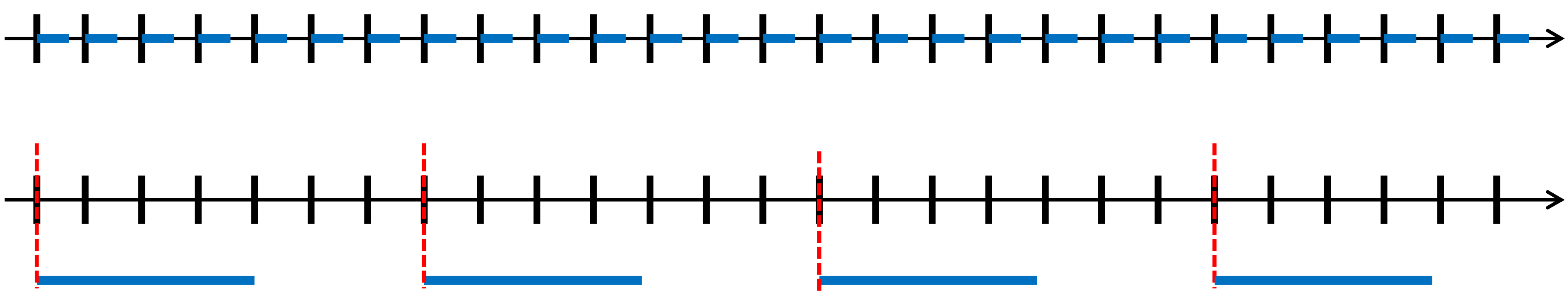



```
while (1) {
        if (processData){
                processData = false;
                process_Pitch(IN_proc , OUT_proc);
        }
 }
```

```
void process_Pitch(uint32_t *buff_in, uint32_t *buff_out){
        uint32_t decim_sample;
        uint16_t sample;
        for (sample=0; sample<N_PITCH_DATA; sample+=2){
  // take the average of every two samples and place the result in position n-th and n+N/2-th
                decim_sample = (buff_in[sample] + buff_in[sample+1])>>1;
                buff_out[sample>>1] = decim_sample;
                buff_out[N_PITCH_DATA/2+(sample>>1)] = decim_sample;
        }
}
```

Data- and block-based DSP

	Data-based	Block-based
Delay (important, e.g.: in control applications)	good	bad
Memory requirement	good	bad (buffering requires extra memory block)
Possibility of reduction of computation complexity	bad	Good (e.g.: filtering by FFT)
Real-time operation	Critical: short timings, timing requirements must be met in sample time	Less critical: timing requirements have to be met in block time (in case of PC practically only block based processing is possible)



Change between data processing modes

- Change between data- and block-based data processing is possible
- The SW architecture must fit to the mode of data processing
  - E.g.: FFT cannot be sample-based
- Change from sample-based to block-based data processing
  - Incoming and outgoing data must be organized into buffers
  - When buffers are full:
    - Interchange of buffers
    - Data processing is started
- Change from block-based to sample based signal processing
  - Data elements of the block are processed one by one
  - Processing function is called sample-wise
  - Result of processing is stored in the output buffer

```
for (ii=0; ii<N; ii++){
        outBuff[ii] = process(inBuff[ii]);
}
```

Handling nonlinearity

Méréstechnika és
Információs Rendszerek
Tanszék

Handling nonlinearity

- Nonlinear functions:
  - Compensating sensor characterisitcs
    - Temperature measurement (NTC, PTC)
    - Nonlinearity of bridges (e.g.: Wheatstone-bridge output)
  - Signal generation
    - Sinusoidal signal: motor control
  - Trigonometrical functions
    - Geometrical calculation
  - Change to decibel scale: log function
  - Root calculation
    - Geometrical calculations
    - RMS (root-mean-square) calculations
  - Division: when no embedded function is available

Table-based storing

- Generating nonlinear functions can be done in an analytical way:
  - Taylor-series approximation
  - Polynomic approximation using other methods
- Problem:
  - Computation need is generally large, especially when convergence is slow
  - Floating-point calculation is required
- Table-based method:
  - Nonlinear function points are stored in a table
  - During function call, the values of the table are used instead of calculations

Table-based sinusoidal

- Sine built up from table using 50 and 500 samples

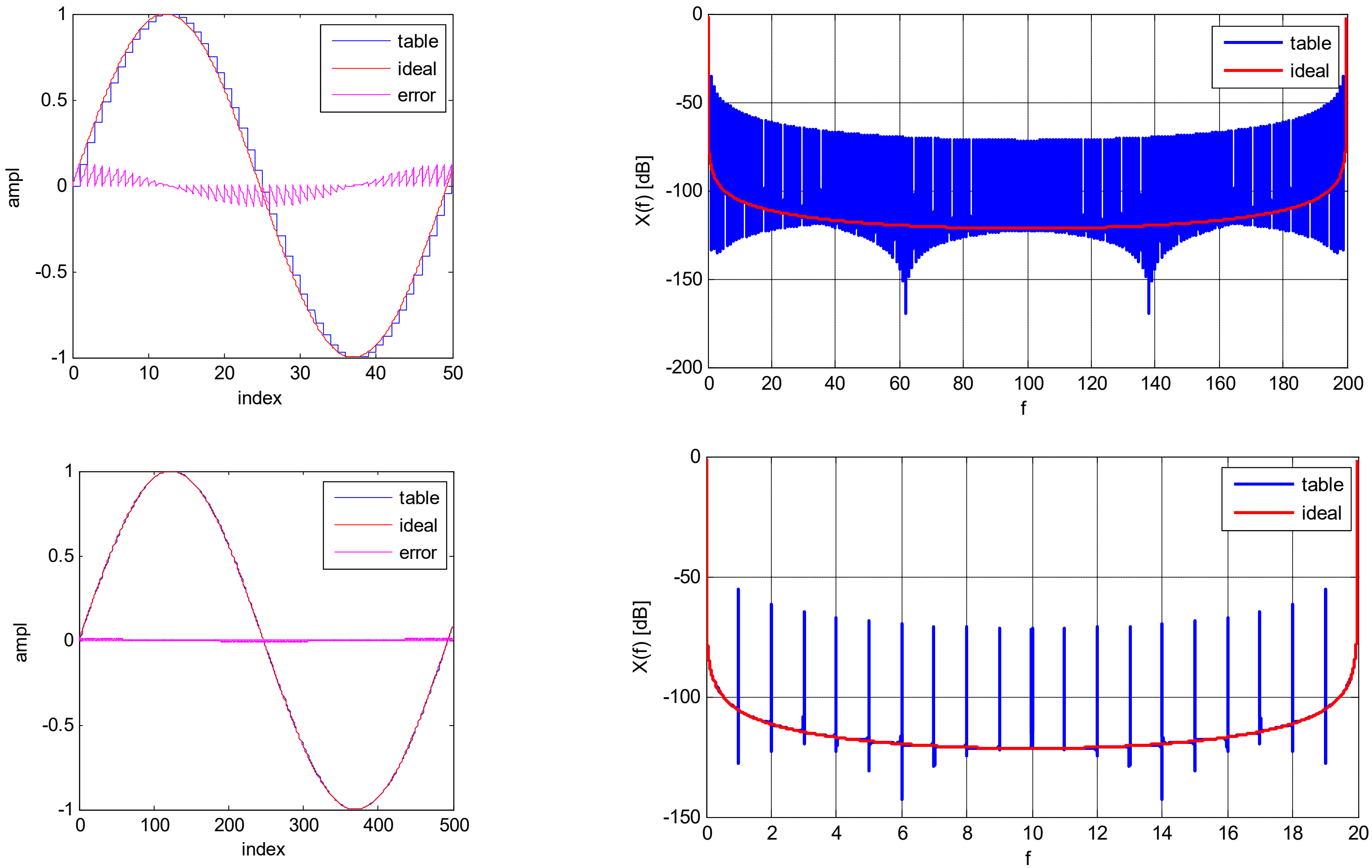
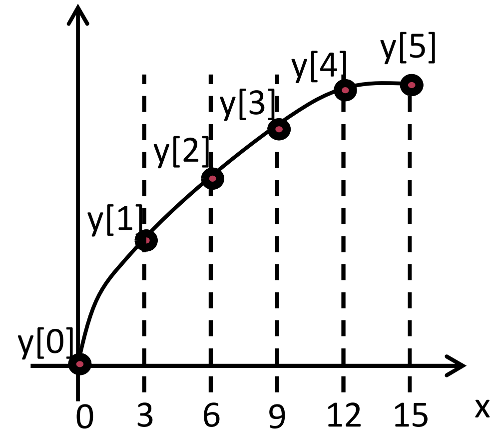


Table-based values with interpolation

- Controversial conditions:
  - Accuracy: requires larger table (larger granularity)
  - Memory size: the smaller table the better
  - Compromise is needed
- Solution:
  - Application of closest samples
  - The approximation of the function is generated by the interpolation between the two samples

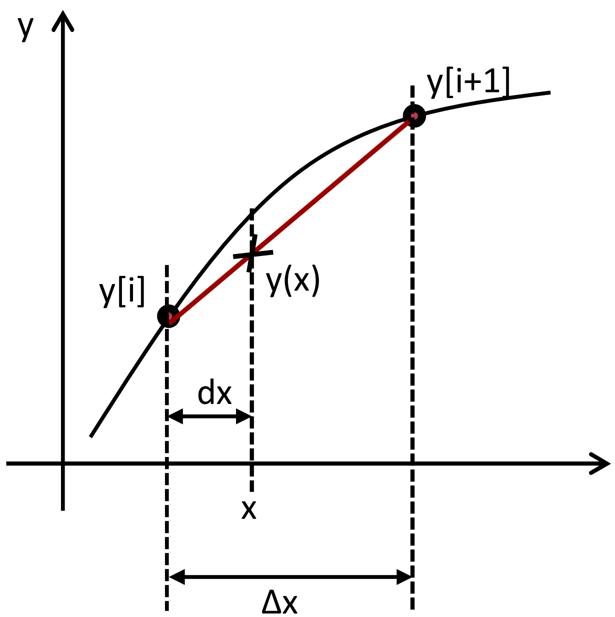
Table-based values with interpolation

- Calculation of linear interpolation
- Quastion: what is the value of y(x) for a given x?



Be y(x) function stored in N+1 points:
• x ϵ [0…n]: input range
• i ϵ [0…N]: table is mapped into how many points
Example: n=15; N=5

Distance of samples in the stored table:
Δx = n/N



1. Calculate which index is the closest from bottom to the input x value:
i = floor(x/Δx) = floor(x/n*N)
y[i] and y[i+1]: the neighboring samples around y(x)

2. Calculate the distance from the applied index of the table
dx = x – i*Δx

3. Interpolation:

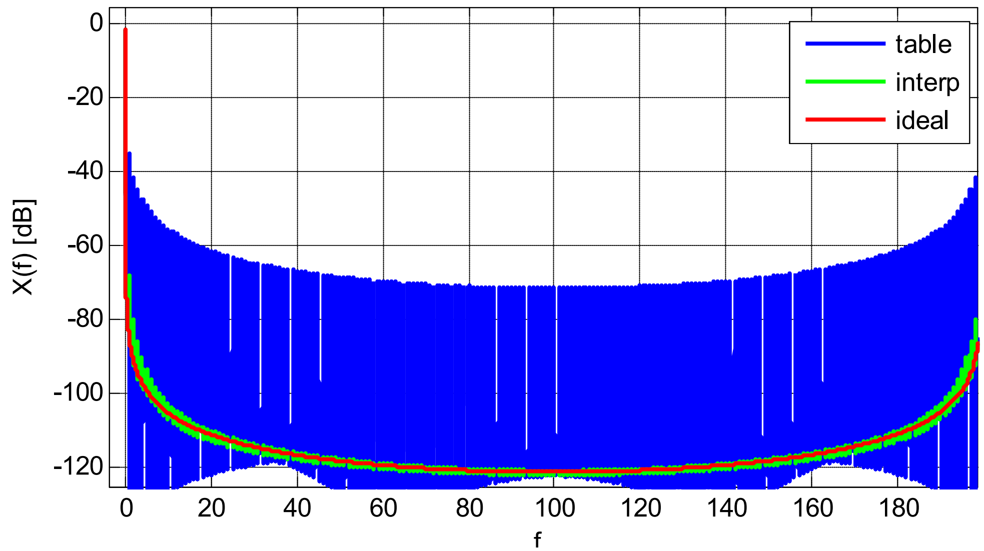
$$y(x) \approx \frac{dx \cdot y[i+1] + (\Delta x - dx) \cdot y[i]}{\Delta x}$$

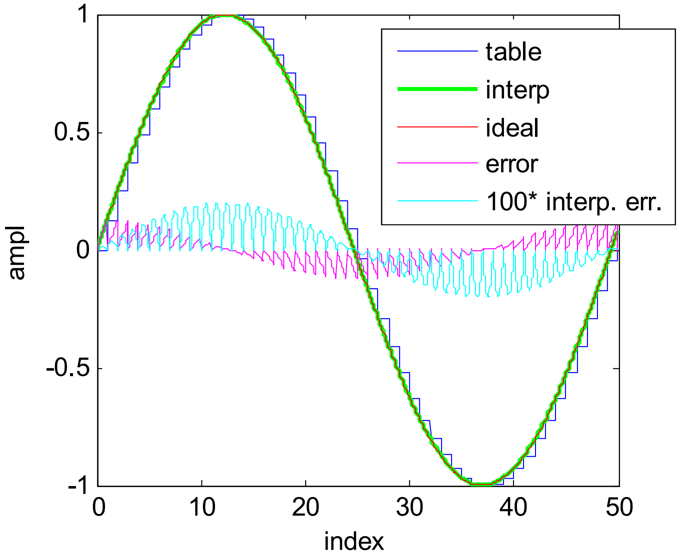
In a different Form (but same content):

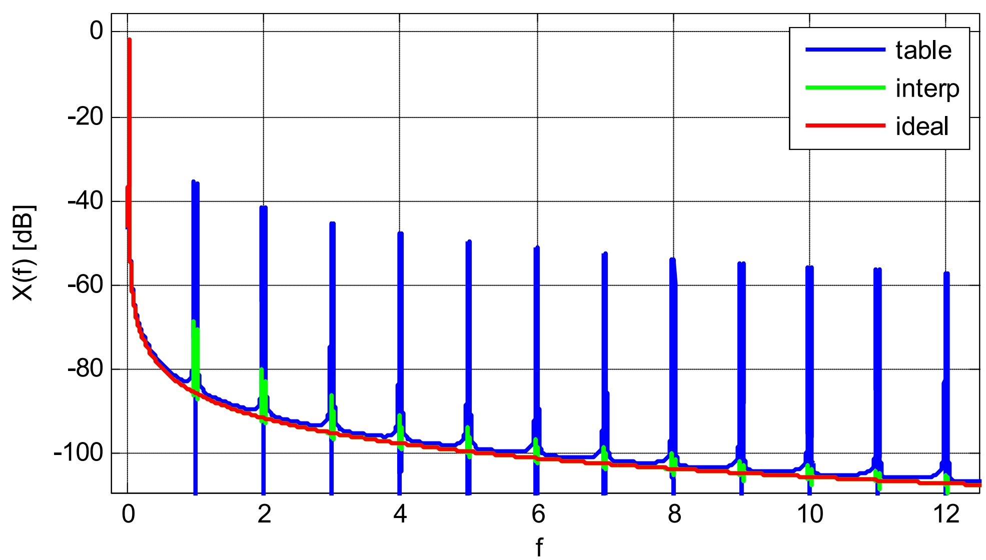
$$y(x) \approx \frac{dx}{\Delta x} y[i+1] + \frac{\Delta x - dx}{\Delta x} y[i]$$

Table-based values with interpolation

- Example: sine in 50 points with linear interpolation
  - The error is reduced by two order: error of linear interpolation by factor of 100 is very close to that of the one considered as ideal







Signal generation

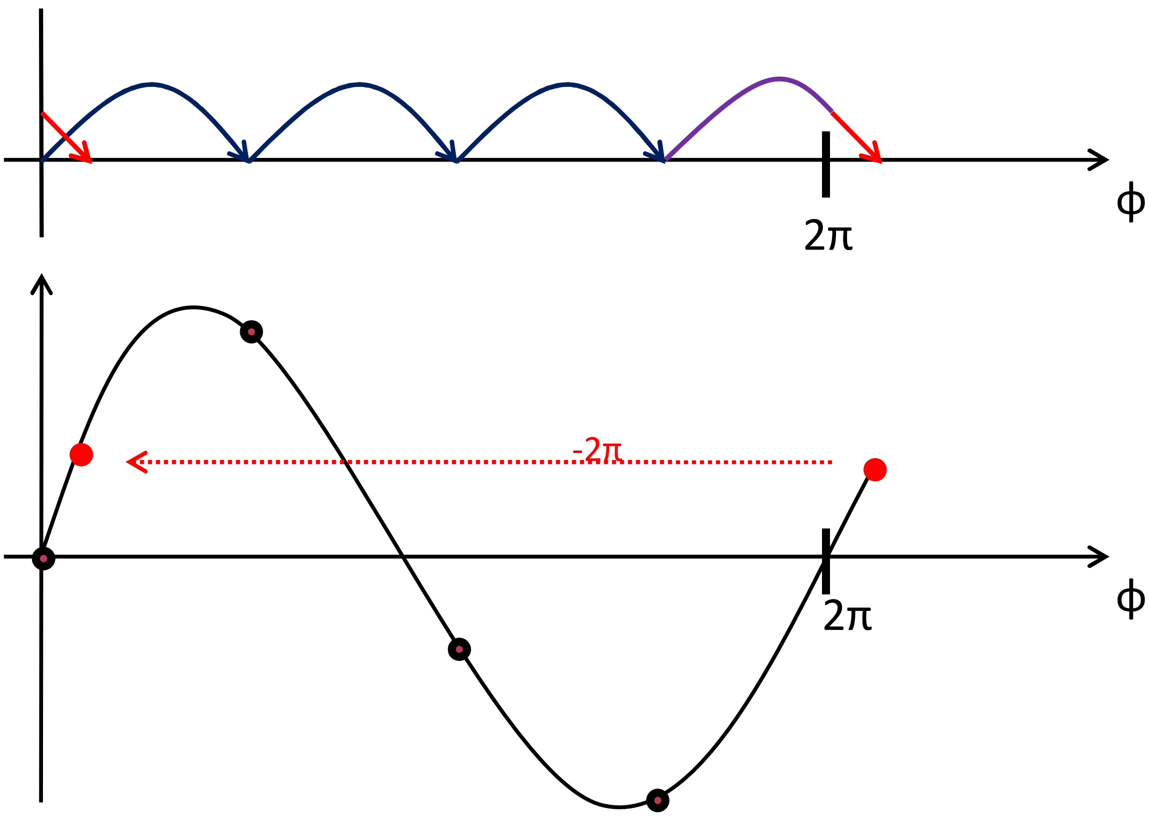
Méréstechnika és
Információs Rendszerek
Tanszék

Signal generation: sinusoidal

- Examined case: generation of sinusoidal signal
- Parameters:
  - Frequency: f
  - Sampling frequency: fs
  - Amplitude: A
  - Time variable: t
- Analitical equation: $x(t) = A*\sin(2\pi*f*t + \phi_0) = A*\sin(\phi(t))$
- Generation of time (not a good solution in practice):
  - t = t + 1/fs; **// in every time instant**
  - Problem: after a while 1/fs increment cannot be added to the actual value of t since the dynamics of number representation is not enough
- In practice better to use the phase
  - Assumption: **sin(…) function and floating point representation of numbers is available**
  - $\Delta\phi = 2\pi*f/fs$
  - $\phi = \phi + \Delta\phi$
  - **if** ($\phi>2\pi$): $\phi = \phi - 2\pi$; **// make it periodic for 2π**
  - $x = A*\sin(\phi)$

Signal generation: sinusoidal

- Phase variable is represented by modulo 2π arithmetic
  - Arbitrarily scalable to any interval, in some cases sin(x) x∊[0…1] interval is used



$\Delta\phi = 2\pi * f/fs$
$\phi = \phi + \Delta\phi$
if $(\phi>2\pi)$: $\phi = \phi - 2\pi$;
$x = A*\sin(\phi)$

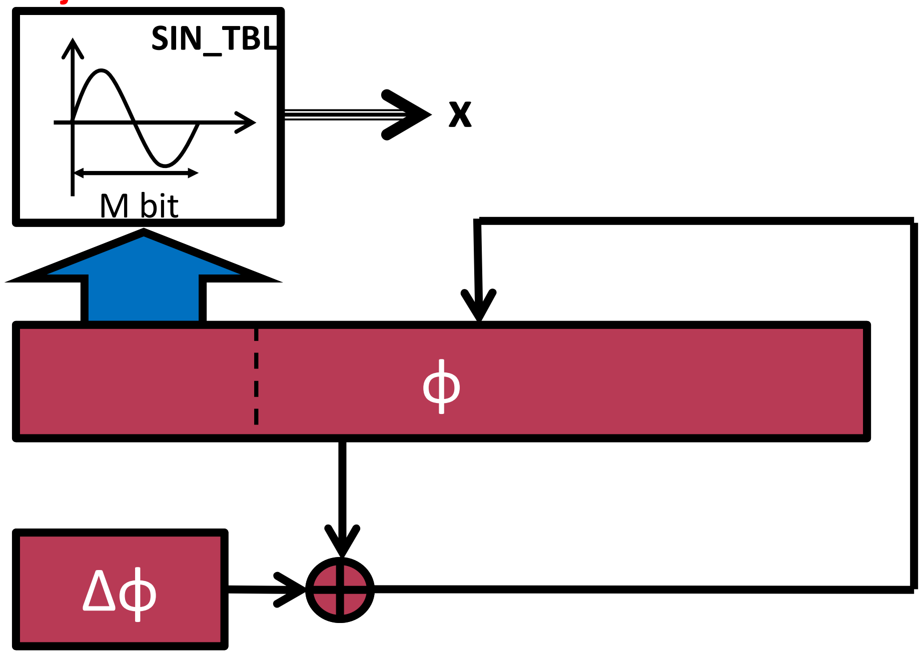
Signal generation: sinusoidal

- **Fixed-point representation**
- N-bit phase variable is used: ϕ
  - Frequency resolution: $\Delta f = fs/2^N$.
    - Example: fs=50kHz, N = 8 bit → $\Delta f = 50kHz/2^8 = 195Hz$
    - Example: fs=50kHz, N = 16 bit → $\Delta f = 50kHz/2^{16} = 0.763Hz$
- Sine-table addressable by M bits is used
- Not always possible to apply the same bit length:
  - The larger the N the better the frequency resolution
  - M should not be too large since consumes too much memory

Assuming Fixed-point (integer) representation results in automatic application of modulo arithmetic

$\Delta\phi = f_{signal}/(fs/2^N)$ // initialization
Algorithm: for every new sample

```
process(){
    ϕ = ϕ + Δϕ  // stepping the phase variable
    addr = ϕ>>(N–M) // calculating address
     x = SIN_TBL[addr]
}
```



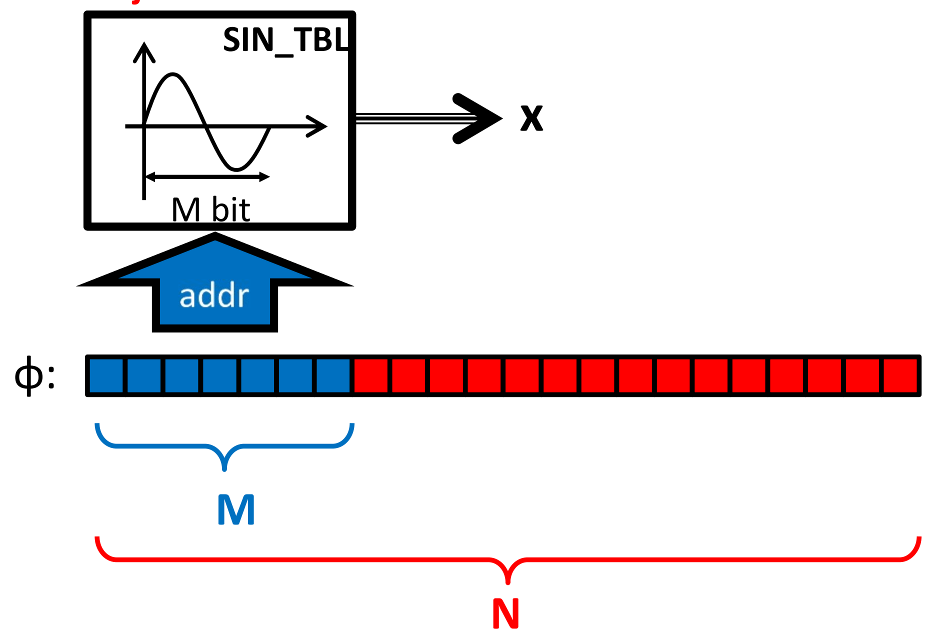
Signal generation: sinusoidal

- **Fixed-point representation**
- N-bit phase variable is used: ϕ
  - Frequency resolution: $\Delta f = fs/2^N$.
    - Example: fs=50kHz, N = 8 bit → $\Delta f = 50kHz/2^8 = 195Hz$
    - Example: fs=50kHz, N = 16 bit → $\Delta f = 50kHz/2^{16} = 0.763Hz$
- Sine-table addressable by M bits is used
- Not always possible to apply the same bit length:
  - The larger the N the better the frequency resolution
  - M should not be too large since consumes too much memory

Assuming Fixed-point (integer) representation results in automatic application of modulo arithmetic

$\Delta\phi = f_{signal}/(fs/2^N)$ // initialization
Algorithm: for every new sample

```
process(){
    ϕ = ϕ + Δϕ  // stepping the phase variable
    addr = ϕ>>(N–M) // calculating address
     x = SIN_TBL[addr]
}
```



Signal generation: sinusoidal

- Example:
  - Sampling frequency: fs = 20kHz
  - Size of sine-table: 8 bit
  - Minimally required resolution: $\Delta f<0.1$ Hz
  - Starting frequency: f_{signal}=200Hz
- Solution:
  - N ≥ 18 bit → $\Delta f=20kHz/2^{18}=20000Hz/ 262144 =0.0763Hz$
  - $\Delta\phi= f_{signal}/\Delta f = f_{signal}/(fs/2^N)= 200/(20000/262144)=2621.4$
  - Integer arithmetic, so $\Delta\phi=2621$ → f_{signal} = 2621* 20000Hz/ 262144= 199.966Hz
  - Frequency error: 0.034Hz

Algorithm:

```
dfi = 2621;
fi = (fi + dfi) & 0x03FFFF; // mask out lower 18 bit: modulo arithmetic
x = SIN_TBL[fi>>10]; // choosing upper 8 bits for addressing
                     // (lower 8 bit could be used for interpolation)
```