20. IIR filtering, on-line

The IIR filtering is very similar to FIR filtering as far as the implementation in the microcontroller is concerned. An example of a program for IIR (Infinite Impulse Response) filtering will be given.

Mathematically, the IIR filtering is expressed as:

$$y_k = \sum_{m=0}^{M} a_m x_{k-m} + \sum_{n=1}^{N} b_n y_{k-n}$$

Here coefficients (weights) are marked a_m and b_n, x are input samples and y are the results of filtering. The coefficients are determined using more complex algorithms than the inverse Fourier transform used for FIR coefficients. In general, *M* is not equal to *N*.

The procedure to calculate the output from a filter is presented in Fig. 1. There are two (circular) buffers involved, one for input samples *x* and one for output results *y*. The new result y_k is composed by adding together two convolutions. The upper convolution involves the input samples *x* and coefficients *a*, the lower convolution involves former results *y* and coefficients *b*. The new result y_k is also sent to the DAC as the result of filtering.

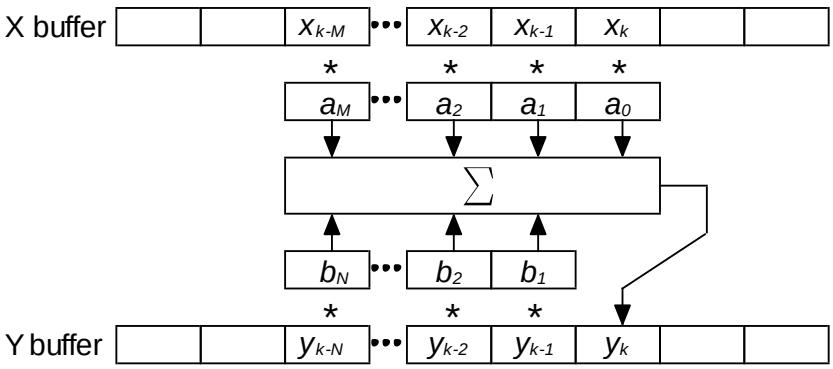

Figure 1: Graphical representation of the IIR filtering

The implementation of filtering in the microcontroller is based on the implementation of the FIR filter. The listing of the program is given in Fig. 2. Functions for the initialization of peripherals are the same as used in chapter on FIR filtering, and are not repeated here.

There are two circular buffers defined as global variables, since they must be reachable from the interrupt function. The buffer for input samples x1 is declared as integer, while the buffer for the output results y1 is declared as float. The reason is the required precision; with IIR filters the numerical errors caused by the use of integer results may lead to poor filter performance or even to numerical instabilities and oscillations. The output buffer could also be integer, but it should be up-

scaled as were the coefficients in the former example on FIR filtering. There are two pairs of such circular buffers to allow two signals to be filtered simultaneously. The lengths of buffers are exaggerated.

Luckily for us, tables with coefficients exist. For the purpose of this programming example the coefficient values are copied from reference [1]. A fourth order filter with Chebishew characteristics is implemented. The corner frequency of 0.025 of the sampling frequency and 0.5% ripple in the pass-band are used. The coefficients are:

m/n	0	1	2	3	4
a_m	1.504626e-5	6.018503e-5	9.027754e-5	6.018503e-5	1.504626e-5
b_n		3.725385e0	-5.226004e0	3.270902e0	-7.705239e-1

These values are coded within the declaration section at the beginning of the program.

Within the main function the ADC and DAC are initialized, the timer is used to assure periodic sampling and generation of signals, and the interrupt controller NVIC is enabled for interrupt requests from the ADC. Following this the microcontroller continues with the execution of the

```
#include "stm32f4xx.h"

int   x1[4096], x2[4096], xyPtr;  // declare input circular buffers
float y1[4096], y2[4096];         // declare output circular buffers

// declare and init IIR weights: 4th order, Chebishew, Low Pass, [1]
// 0.5%, -3 dB at 0.025 (250 Hz here) of sampling frequency
float a[5] = {1.504626e-5, 6.018503e-5,  9.027754e-5, 6.018503e-5,  1.504626e-5};
float b[5] = {0          , 3.725385e0 , -5.226004e0 , 3.270902e0 , -7.705239e-1};

int main ()  {
  GPIO_setup();                   // GPIO set-up
  DAC_setup();                    // DAC set-up
  ADC_setup();                    // ADC set-up
  Timer2_setup();                 // Timer 2 set-up
  NVIC_EnableIRQ(ADC_IRQn);       // Enable IRQ for ADC in NVIC

  // waste time - indefinite
  while (1) {
    if (GPIOE->IDR & 0x0001) GPIOA->ODR |=  0x0040;   // LED on
    else                     GPIOA->ODR &= ~0x0040;   // else LED off
  };
}

// IRQ function
void ADC_IRQHandler(void)         // this takes approx 6us of CPU time!
{
  GPIOE->ODR |=  0x0100;          // PE08 up
  x1[xyPtr]  = ADC1->DR;          // pass ADC -> circular buffer x1
  x2[xyPtr]  = ADC2->DR;          // pass ADC -> circular buffer x2
  float conv = 0;                           // declare and init sum
  for (int i = 0; i < 5; i++)               // for koefs 0 to 4
    conv += a[i] * x1[(xyPtr-i) & 4095];    // convolve inputs
  for (int i = 1; i < 5; i++)               // for koefs 1 to 4
    conv += b[i] * y1[(xyPtr-i) & 4095];    // convolve outputs
  y1[xyPtr] = conv;                         // save filtered result
  y2[xyPtr] = (float)x1[xyPtr];             // save original
  DAC->DHR12R1  = (int)y1[xyPtr];           // filtered -> DAC
  DAC->DHR12R2  = (int)y2[xyPtr];           // original -> DAC
  xyPtr = (xyPtr + 1) & 4095;     // increment pointer to circulat buffer
  GPIOE->ODR &= ~0x0100;          // PE08 down
}
```

Figure 3:TheA listing of a program to implement IIR filtering

endless loop to waste time (and periodically check switches and control LEDs).

The important stuff again happens in the interrupt function. Here results from the two ADCs are first stored in the circular buffer. This is followed by the calculation of two convolutions for one input signal only, one convolution for current and past input samples and coefficients a_m, and one convolution for the past results of filtering and coefficients b_n. The complete calculation is performed using floating point arithmetic to assure the required precision, and the result is stored in the output circular buffer. The unfiltered input signal is copied to the other output buffer for comparison. Lastly, current values from the output buffers are copied to DACs, and the pointer to circular buffers gets updated.

All calculations are sample management is enclosed into two statements to make a bit at port E high at the beginning of calculation and to return the same bit to low at the end of calculation. This bit can be used to determine the time needed to execute the interrupt function, which is about 6μs for this example.

[1] Steven W. Smith: The Scientist and Engineer's Guide to Digital Signal Processing