20. IIR filtering

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.

20.1. The theory of FIR filtering

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* is the result of filtering in one step *k*. The coefficients are determined using more complex algorithms than the inverse Fourier transform used for FIR coefficients h_m.

The procedure to calculate the output from a filter is presented in Fig. 19.1. 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 first convolution involves input samples *x* and coefficients *a*, the second convolution involves past results *y* and coefficients *b*. The new result y_k is sent to the DAC as the result of filtering, and also stored to be used in successive calculations.

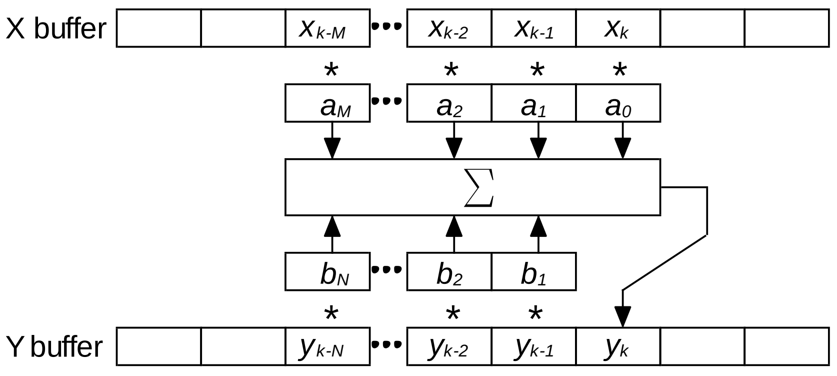

Figure 20.1: Graphical representation of the IIR filtering

The implementation of IIR filtering is based on the previous example on FIR filtering. The listing of the program is given bellow. Functions for the initialization of peripherals are the same as used in chapter on FIR filtering, and are not discussed again.

There are two circular buffers declared as global variables, since they must be reachable from the interrupt function. The buffer for input samples *X* is declared as integer, while buffer for the output results *Y* is declared as float. The reason is the required precision; with IIR filters the numerical errors caused by the use of integer variables 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 *X1*, *X2* and *Y1*, *Y2* to allow two signals to be filtered simultaneously. The lengths of buffers are exaggerated.

The procedure to determine the values of coefficients *a* and *b* can be complex, and is out of the scope of this text. 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, the listing of the main part of the program is given below.

```
#include "stm32f4xx.h"
#include "stm32f4xx_rcc.c"
#include "stm32f4xx_adc.c"
#include "stm32f4xx_dac.c"
#include "stm32f4xx_gpio.c"
#include "stm32f4xx_tim.c"
#include "dd.h"
#include "math.h"

short int X1[1024], X2[1024], k = 0;
float Y1[1024], Y2[1024];
#define pi 3.14159

// declare and init IIR weights: 4th order, Chebishew, Low Pass, reference [1]
// +/- 0.5%, -3dB at 0.025 (250Hz) 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 ()  {

  GPIOEinit ();
  ADCinit_T5_CC1_IRQ();
  DACinit();
  TIM5init_TimeBase_CC1(8400);           // 8400 == 100us == 10kHz

  while (1) {                            // endless loop
  };
}
```

The program starts with the inclusion of CMSIS files, and proceeds to the declaration section, where the circular buffers are defined. This is followed by the declaration and the initialization of coefficients *a* and *b*. The execution starts with the section "main()", where the peripherals are configured by four calls of the functions. Next the processor enters the empty infinite loop.

The important stuff again happens in the interrupt function, listed below.

```
void ADC_IRQHandler(void)
{
  GPIOE->BSRRL = BIT_8;
  X1[k] = ADC1->DR;                                                            // 4
  X2[k] = ADC2->DR;                                                            // 5
  float conv = 0;
  for (int m = 0; m<5; m++)                                                    // 7
```

```
    conv += a[m] * X1[(k - m) & 1023];                                                     // 8
  for (int n = 1; n<5; n++)                                                                // 9
    conv += b[n] * Y1[(k - n) & 1023];                                                     // 10
  Y1[k] = conv;                                                                            // 11
  Y2[k] = (float)X1[k];                                                                    // 12
  DAC->DHR12R1 = (int)Y1[k];                                                               // 13
  DAC->DHR12R2 = (int)Y2[k];                                                               // 14
  k++; k &= 1023;                                                                          // 15
  GPIOE->BSRRH = BIT_8;
}
```

Here results from the two ADCs are first stored in the circular buffer in lines 4 and 5. This is followed by the calculation 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, lines 7 to 10. The complete calculation is performed using floating point arithmetic to assure the required precision, and the result is stored in the output circular buffers in line 11. The unfiltered input signal is copied to the other output buffer for comparison in line 12. Lastly, current values from the output buffers are copied to DACs in lines 13 and 14, and the pointer to circular buffers is updated in line 15.

All calculations and sample management is fenced into two statements to make a bit at port E high at the beginning of calculation and to return the same bit 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