19. FIR filtering

Digital filtering of analog signals in real time is easy to implement when one has an ADC, a processor, and a DAC, Fig. 19.1. An example of FIR (Finite impulse Response) filtering will be given.

Figure 19.1: The blocks involved in FIR filtering.

19.1. The theory of FIR filtering

The FIR (Finite Impulse Response) filtering implements a convolution of input signal with predefined coefficients to achieve filtering effect. The convolution formula is given by ([1], chapter 6):

$$y_k = \sum_{m=0}^{M-1} h_m x_{k-m}$$

The coefficients h_m define the properties of the filter, and are calculated as the inverse Fourier transform of the frequency response for the desired filter. Coefficients h_m span for positive and negative indexes m, from –M to +M (theoretically M approaches infinity, but is truncated in practice). This effectively means that in order to calculate the filter response at time k, one should know samples x from current time k, from past (x_{k-1}, x_{k-2}, … , x_{k-m}), and also from the future (x_{k+1}, x_{k+2}, … , x_{k+M}). Since knowing of the future is not the privilege of ordinary people (or processors), we cannot implement convolution formula directly, but have to resort to a delayed calculation by modifying/rearranging the convolution formula:

$$y_k = \sum_{m=-M}^{M} h_m x_{k-m-M}$$

This is graphically presented in Fig. 19.2. Input samples *x* are stored in a (circular) buffer one per box in the drawing, and samples from the past with indexes from $k - 2M$ to k are used to calculate convolution. The result y_k used as current output from the filter is actually a delayed version of the filtered input signal *x*. The delay depends on the number of coefficients used in convolution.

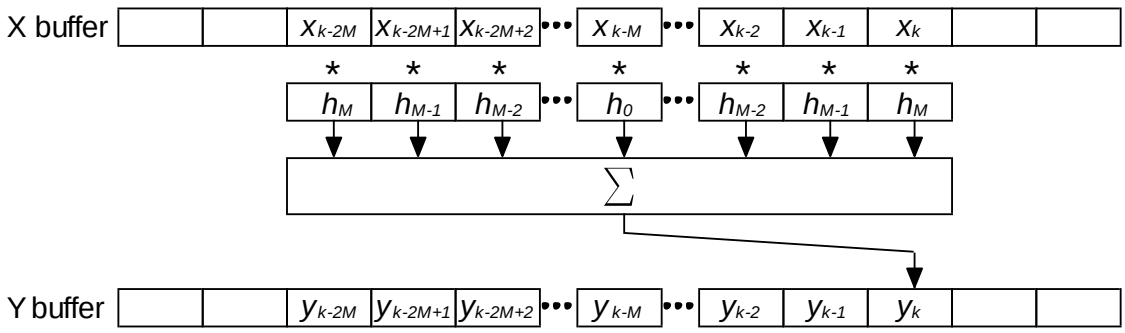

Figure 19.2: The samples used in FIR filtering

The coefficients h_m for a low pass FIR filter are given by (again [1], chapter 6):

$$h_m = 2\frac{f_c}{f_s} \cdot \frac{\sin\left(2\pi m \frac{f_c}{f_s}\right)}{2\pi m \frac{f_c}{f_s}} ,$$

where f_s stands for the sampling frequency, and f_c for the corner frequency of the filter. The sharpness of the filter is better for a large number of coefficients (*M*). However, a large number of coefficients imply many multiplications in the convolution formula, so one has to choose *M* carefully.

The attenuation might not be the expected one due to the limited number of coefficients used, but can be improved by gradually reducing values of coefficients close to index *M*; the process is known as "windowing". A common window function is a raised cosine, named by its inventor von Hann window.

$$h_{m\,windowed} = h_m \cdot \frac{1 + \cos(\pi \cdot \frac{m}{M-1})}{2}$$

19.2. FIR filtering in real time - software

FIR filtering can be implemented in real time. Input signal must be sampled at regular time intervals, and this can be achieved as described in chapter 13. The timer can start the acquisition, and then the sampled value can be stored into a circular buffer *X*. The buffered samples can be used in calculation of the convolution formula following the acquisition of each new sample, and the result of convolution can be converted back to analog signal using a DAC or stored into another circular buffer *Y* for further use. The important thing is that the calculation of convolution must be performed immediately after a new sample of the input signal is available, therefore within the interrupt function. The calculation must be finished before the next interrupt request; this additionally limits the number of coefficients h_m in the convolution formula since time is spent for every multiplication.

Coefficients used in convolution formula stay the same throughout the filtering process, and should be calculated only once prior to the filtering.

The listing of the program is given in Fig. 3. It commences with the inclusion of CMSIS files. Next two circular buffers *X1* and *X2*, and a pointer *k* are declared; these must be global variables since they are used in the interrupt function and must retain their content from one execution of the interrupt function to another. The length of circular buffers is exaggerated, but shows that a long buffer is not a problem for the microcontroller. The buffers are declared as short integers, 16 bit signed values; enough to hold the result from an ADC. Next an array for coefficients *h* is declared, and is composed of "float" values. These coefficients have values of less than one, and must be declared as floating point numbers, apparently. Again, the length of this array is exaggerated.

```
#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"
#define pi 3.14159

short int X1[1024], X2[1024], k = 0;                                          // 11
float h[1024];                                                                // 12

int main ()  {
  h[0] = 2.0 * 100.0 / 10000.0;         // central weight: 2 x fc / fs       // 15
  for (short m = 1; m < 64; m++)                                              // 16
```

```
    h[m] = (h[0] * sin(pi * m * h[0])) / (pi * m * h[0]);       // other weights              // 17
  for (short m = 1; m < 64; m++)                                                              // 18
    h[m] = (h[m] * cos(pi / 2 * m / 63.0));                     // windowing                  // 19

  GPIOEinit ();                                                                               // 21
  ADCinit_T5_CC1_IRQ();                                                                       // 22
  DACinit();                                                                                  // 23
  TIM5init_TimeBase_CC1(8400);          // 8400 == 100us == 10kHz                             // 24

  while (1) {                           // endless loop                                       // 26
  };
}
```

The body of the program starts in line 15. Coefficients are calculated in lines 15 to 17. As we can see from the formula above the values of coefficients are symmetrical around the central one with index 0, therefore we only need to calculate coefficients with positive indexes *m* and the one with index *m*=0, altogether 64 coefficients. Lines 18 and 19 implement the windowing.

Peripherals are configured and initialized by calls in lines 21 to 24. The functions called are the same as used in previous chapters. They turn on two ADCs and two DACs, define their properties and configure associated port pins. The configuration also starts a timer to issue periodic Start Conversion pulses for the ADC at 100μs time intervals, and enables interrupt requests from the ADC.

The microcontroller is now ready to start filtering and the program continues into an endless loop where it wastes time. The important stuff happens in the interrupt function listed below.

```
void ADC_IRQHandler(void)
{
  GPIOE->BSRRL = BIT_8;
  X1[k] = ADC1->DR;
  X2[k] = ADC2->DR;
  float conv = (float)X1[(k - 100) & 1023] * h[0];                                            // 6
  for (int m = 1; m<64; m++)                                                                  // 7
    conv += h[m] *(float)(X1[(k - 100 + m) & 1023] + X1[(k - 100 - m) & 1023]);               // 8
  DAC->DHR12R1 = (int)conv;                                                                   // 9
  DAC->DHR12R2 = X1[(k - 100) & 1023];                                                        // 10
  k++; k &= 1023;                                                                             // 11
  GPIOE->BSRRH = BIT_8;
}
```

When a new sample is ready in the ADC the interrupt function is called. Results from both ADCs are stored into circular buffers *X1* and *X2* at location pointed to by the pointer *k*. Next a float variable *conv* is declared in line 6, and a product of the central weight *h[0]* and central sample *X1[(k-100) & 1023]* is stored into it. Here we assume that an offset of 100 from the current sample is sufficient to cover the length of the convolution; the offset should be bigger than *M*. The AND function within the square brackets is used to avoid accessing the circular buffer beyond its boundaries as explained in chapter 13.

The rest of the convolution is implemented within the 'for' statement, lines 7 and 8, for coefficients with indexes from 1 to including 63. Coefficient values are symmetrical around the central coefficient, and it would be a waste of time to make two separate multiplications of input samples with the same value of coefficient. It is better to add the two (symmetrical) input samples first, and then multiply the sum by the coefficient. The input samples are indexed as *[k – 100 + m]* and *[k – 100 - m]* to emphasize the symmetry, and the AND function is used to keep the pointer within the circular buffer.

Once the convolution is calculated the result is converted to integer form and sent to the first DAC. The unfiltered original signal is sent to the second DAC for comparison. This signal must be equally delayed as the filtered one; the central sample as used in convolution is sent.

The two GPIOE writes are used to make a pulse at port E, pin 8, and the pulse can be used to determine the time needed to execute the interrupt function (12μs in this case).

The speed of execution can be improved slightly by using integer variables and integer mathematical operations. As stated before the coefficients have values less than one and cannot be converted to integers. However, they can be multiplied by 65536 (which is equivalent for shifting the coefficient values for 16 bits to the left) and then converted to integers. Since we now have integer coefficients and integer samples of input signals all calculations can be performed in integer arithmetic. Since coefficients are 65536 times too big now the convolved results are also65356 times too big, and have to be divided by 65536 (shifted by 16 bits to the right) to obtain the correct value.

Only few lines of program need to be changed to implement the idea. The declaration of the array with coefficients *h[1023]* is changed to "int" in line 11.

The calculation of coefficient values in "main" part of the program is changed. First a value of the central coefficient is calculated as "float", but is immediately converted to integer and stored into the array of coefficients at index 0. Next coefficients for indexes from 1 to 63 are calculated, all multiplied by 65536. In order to implement the windowing the calculated coefficients are again converted to floating point numbers, multiplied by the weight, and converted back to integers, see the listing below.

```
float h0 = 2.0 * 100.0 / 10000.0;
  h[0] = (int)(h0 * 65536);                 // central weight: 2 x fc / fs
  for (short m = 1; m < 64; m++)
    h[m] = (int)((h0 * 65536 * sin(pi * m * h0)) / (pi * m * h0));     // other weights
  for (short m = 1; m < 64; m++)
    h[m] = (int)((float)h[m] * cos(pi / 2 * m / 63.0));                // windowing
```

In the interrupt function the intermediate variable "conv" is declared as integer, and the calculation is performed without explicitly stating the integer arithmetic. This gets done automatically when all variables used are integers. The only difference follows at the point where the result of convolution is sent to the DAC; it first gets divided by 65536 (shift right for 16 bits), and then written to the DAC.

```
  int conv = X1[(k - 100) & 1023] * h[0];
  for (int m = 1; m<64; m++)
    conv += h[m] *(X1[(k - 100 + m) & 1023] + X1[(k - 100 - m) & 1023]);
  DAC->DHR12R1 = X1[(k - 100) & 1023];
  DAC->DHR12R2 = conv >> 16;
```

The execution of the interrupt function using integer arithmetic takes about 9μs.

An interesting FIR function that can be implemented using the FIR filtering is the phase shifting of a signal for 90 degrees; the amplitude of the shifted version remains unchanged. Such FIR function is called the Hilbert transform. The coefficients for a Hilbert transform are given by:

$$h_m = \frac{-1 + (-1)^m}{\pi m}$$

The same program as used for the integer version of FIR filtering can be modified to perform the 90 degree shift, but coefficients must be calculated following the formula above, see listing bellow.

```
  h[0] = 0;
  for (short m = 1; m < 64; m++)
    h[m] = (int)(65536.0 * (-1.0 + cos(pi * m)) / (pi * m));           // coefficients
  for (short m = 1; m < 64; m++)
    h[m] = (int)((float)h[m] * cos(pi / 2 * m / 63.0));                // windowing
```

Note that here also the coefficients are multiplied by 65535, and that the result of convolution must be divided by the same factor.

[1] The Scientist and Engineer's Guide to Digital Signal Processing