19. FIR filtering, on-line

A digital filtering of analog signals in real time is easy to implement when one has an ADC, a processor, and a DAC, Fig. 1. An example of FIR filtering will be given.

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

Figure 1:The blocks involved in FIR filtering.

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

The coefficients h_m define the properties of the filter, and are calculated as the inverse Fourier transform of the desired frequency characteristics of the desired filter. Coefficients h_m calculated using this method span for positive and negative indexes m, from –M to +M (and in theory M approaches infinity). This effectively means that in order to calculate 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. 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 required delay depends on the number of coefficients used in convolution.

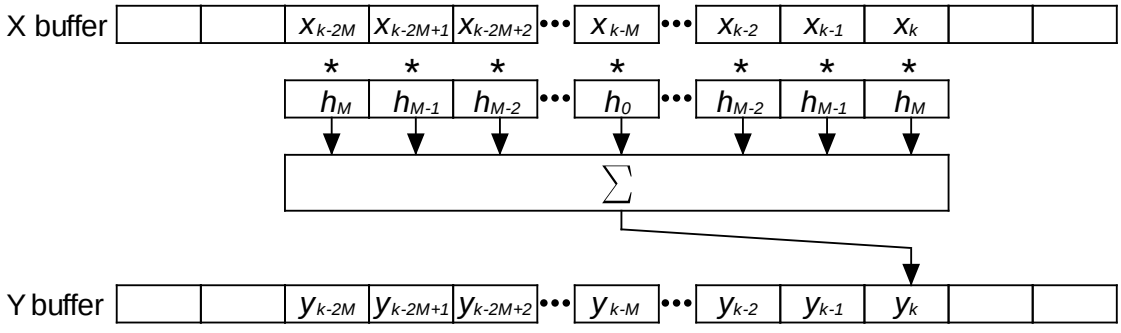

Figure 2: The samples used in FIR filtering

The coefficients h_m for a low pass FIR filter are given by:

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

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. However, a large number of coefficients imply many multiplications in the convolution formula, so one has to choose *m* carefully.

The attenuation at frequencies above the corner frequency can be poor, but can be improved by progressively reducing the values of coefficients close to index *M*; the process is known as "windowing". A common window function is a raised cosine, named von Hann window.

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

FIR filtering can be implemented in real time. Input signal must be sampled at regular time intervals, and this can be achieved by timer built into the microcontroller. The timer can start the acquisition, and then the sampled value can be stored into a circular buffer *x*, as it was already explained in experiment 13, "The use of circular buffer". The buffered samples can be used to calculate the convolution immediately after 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 limits the number of coefficients in the convolution formula, since every multiplication takes some time.

The coefficients used in convolution formula stay the same throughout the filtering, and should be calculated once prior to the filtering.

The listing of the program is given in Fig. 3. Two circular buffers x1 and x2, and a pointer xPtr are declared first; 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 this microcontroller. The buffers are declared as integers, 32 bit signed values. In order to store the result from ADCs a "short" (16 bits) would suffice and would save RAM. Next an array for coefficients *w* is declared, and is composed of "float" values. Following the formula above the coefficients will have the value of less than one, and must therefore be floating point numbers. Again, the length of this array is exaggerated.

In the main part of the program the peripherals are first initialized. The initialization is hidden in functions called from the main, and is the same as it was in previous chapters. The functions themselves are listed at the end of this chapter. The initialization turns on two ADCs and two DACs, defines their properties and associated port pins. The initialization also starts the timer to issue periodic Start Conversion pulses for the ADC at 100μs time intervals, and enables interrupt requests from the ADC.

The main program is continued by calculating the values of coefficients used in convolution. As we can see from the formula above the values of coefficients are symmetrical around the central

coefficient with index 0, therefore we need to calculate coefficients only for positive indexes *m*, and this is done in the next three program lines. There are 63 coefficients for positive indexes and a central one for index 0. Next two lines implement the windowing function.

The microcontroller is now ready to start filtering and the program continues into an endless loop where it wastes time.

```
#include "stm32f4xx.h"
#include "math.h"
#define pi 3.14159

int x1[4096], x2[4096], xPtr;   // declaration of circular buffers
float w[1024];                  // declaration of FIR weights

int main ()  {
  GPIO_setup();                 // GPIO clock enable, digital pin definitions
  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

  w[0] = 2.0 * 100.0 / 10000.0;
  for (short k = 1; k < 64; k++)                          // FIR weights
    w[k] = (w[0] * (sin(pi * k * w[0])) / (pi * k * w[0]));
  for (short k = 1; k < 64; k++)                          // windowing, Hanning
    w[k] = (w[k] * cos(pi/2 * k / 62.0));

  // 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)       // PASS takes approx 42us of CPU time!
{
  GPIOE->ODR |=  0x0100;        // PE08 up
  x1[xPtr]  = ADC1->DR;         // pass ADC -> circular buffer x1
  x2[xPtr]  = ADC2->DR;         // pass ADC -> circular buffer x2
  float conv = (float)x1[(xPtr - 100) & 4095] * w[0];    // take central weight
  for (int k = 1; k < 64; k++)                           // convolve the rest
    conv += w[k] * (x1[(xPtr - 100 + k) & 4095] + x1[(xPtr - 100 - k) & 4095]);
  DAC->DHR12R1  = (int)conv;                             // result   -> DAC
  DAC->DHR12R2  = x1[(xPtr - 100) & 4095];               // original -> DAC
  xPtr = (xPtr + 1) & 4095;     // increment pointer to circulat buffer
  GPIOE->ODR &= ~0x0100;        // PE08 down
}
```

Figure 3: A listing of a program to implement FIR filtering

The important stuff happens in the interrupt function. When a new sample is ready in the ADC, the interrupt function is called. Results from both ADCs are first stored into circular buffers at location pointed to by a pointer xPtr. Next a float variable is declared and a product of the central weight w[0] and past central sample x1[(xPtr-100) & 4095] 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; it must be bigger than *M*. The AND function within the pointer into the circular buffer is used to avoid reads out of the circular buffer boundaries as explained in chapter 13.

The rest of the convolution is implemented within the “for” statement 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 separate multiplications of input samples with the same value of

coefficient. It is better to add the two input samples first, and then multiply the sum by the coefficient. The input samples are indexed as [xPtr – 100 + k] and [xPtr – 100 - k] 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 and sent to DAC for conversion. 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 to the DAC.

The two GPIOE functions are used to make a pulse at port E, and the pulse can be used to determine the time needed to execute the interrupt function (42μs in this case).

The speed of execution can be improved by using integer variables and integer mathematical operations. Only few lines of program shown in Fig. 3 need to be changed, and those are shown in Fig. 4. The declaration of the array with coefficients is changed to "int". As stated before the coefficients have values less than one and cannot be directly 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 stored as integers. Since we now have integer coefficients and integer samples of input signals all calculations can be performed by integers. As expected, this produces the result of convolution which is 65356 times too big, and has to be divided by 65536 (shifted by 16 bits to the right) to obtain the correct result.

```
int w[1024];                    // declaration of FIR weights


// in main

  float w0 = 2.0 * 100.0 / 10000.0;
  w[0] = (int)(w0 * 65536);
  for (short k = 1; k < 64; k++)                              // FIR weights
    w[k] = (int)(w0 * 65536 * (sin(pi * k * w0)) / (pi * k * w0));
  for (short k = 1; k < 64; k++)                              // windowing, Hanning
    w[k] = (int)((float)w[k] * cos(pi/2 * k / 63.0));


// in IRQ function

  int conv = x1[(xPtr - 100) & 4095] * w[0];             // take central weight
  for (int k = 1; k < 64; k++)                           // convolve the rest
    conv += w[k] * (x1[(xPtr - 100 + k) & 4095] + x1[(xPtr - 100 - k) & 4095]);
  DAC->DHR12R1  = conv >> 16;                            // result   -> DAC
```

Figure 4: Changes to the program from Fig. 3 to implement filtering in integer

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 position 0. Next coefficient values 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.

In the interrupt function the intermediate variable "conv" is declared as integer, and the calculation is performed in a regular way without explicitly stating the integer arithmetic. This gets done automatically when all variables 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.

The improvement in speed is significant; the execution of the interrupt function using integer arithmetic takes only 10μs.

An interesting function that can be implemented using the FIR filtering is the shifting of a signal with unknown frequency for 90 degrees. Such 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 used, but the coefficients must be calculated following the formula above, Fig. 5. Note that here also the coefficients are multiplied by 65535, and that the result of convolution must be divided by the same factor.

```
// in main

  w[0] = 0;
  for (short k = 1; k < 64; k++)                    // FIR weights
    w[k] = (int)(65536.0 * (-1.0 + cos(pi * k)) / (pi * k));
  for (short k = 1; k < 64; k++)                    // windowing, Hanning
    w[k] = (int)((float)w[k] * cos(pi/2 * k / 63.0));
```

Figure 5: The calculation of coefficients to implement a Hilbert transform

Figure 6 shows the function for the initialization of the peripherals, which were moved out of the main program in Fig. 3.

```
void ADC_setup(void)    {
  RCC->APB2ENR |= 0x00000100;   //  clock for ADC1
  RCC->APB2ENR |= 0x00000200;   //  clock for ADC2
  ADC->CCR      = 0x00000006;   // Regular simultaneous mode only
  ADC1->CR2     = 0x00000001;   // ADC1 ON
  ADC1->SQR3    = 0x00000002;   // use PA02 as input
  ADC2->CR2     = 0x00000001;   // ADC1 ON
  ADC2->SQR3    = 0x00000003;   // use PA03 as input
  GPIOA->MODER |= 0x000000f0;   // PA02, PA03 are analog inputs

  ADC1->CR2    |= 0x06000000;   // use TIM2, TRG0 as SC source
  ADC1->CR2    |= 0x10000000;   // Enable external SC, rising edge
  ADC1->CR1    |= 0x00000020;   // Enable ADC Interrupt for EOC
}

void DAC_setup(void)    {
  RCC->APB1ENR |= 0x20000000;   // Enable clock for DAC
  DAC->CR      |= 0x00010001;   // DAC control reg, both channels ON
  GPIOA->MODER |= 0x00000f00;   // PA04, PA05 are analog outputs
}

void GPIO_setup(void)   {
  RCC->AHB1ENR |= 0x00000001;   // Enable clock for GPIOA
  RCC->AHB1ENR |= 0x00000010;   // Enable clock for GPIOE
  GPIOE->MODER |= 0x00010000;   // output pin PE08: time mark
  GPIOE->MODER |= 0x00040000;   // output pin PE09: toggle
  GPIOA->MODER |= 0x00001000;   // output pin PA06: LED D390
}

void Timer2_setup(void) {
  RCC->APB1ENR |= 0x0001;       // Enable clock for Timer 2
  TIM2->ARR     = 8400;         // Auto Reload value: 8400 == 100us
  TIM2->CR2    |= 0x0020;       // select TRGO to be update event (UE)
  TIM2->CR1    |= 0x0001;       // Enable Counting
}
```

Figure 6: A listing of function for the initialization

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