Objectives

- To design and implement an FIR band-pass filter with a centre frequency of 20 Hz, a 3-dB bandwidth of 1 Hz and a stop-band attenuation of at least 60 dB. The stop-band is specified as ± 5 Hz of the centre frequency.

Introduction

Laboratory sessions LM1 – LM5 have introduced the tools required for microcontroller based digital control. With this concluding exercise we will apply these techniques in the design of a digital filter. Figure LM6-1 shows the frequency response of a band-pass filter at 20 Hz with a 3-dB bandwidth of 1 Hz (!) and a minimum stop-band attenuation of 60 dB. The stop-band is assumed to include all frequencies below 15 Hz as well as above 25 Hz. Note that this is a reasonably demanding filter specification which might be difficult to satisfy. Remember that an attenuation of 60 dB decreases the amplitude of a signal by a factor 1/1000.

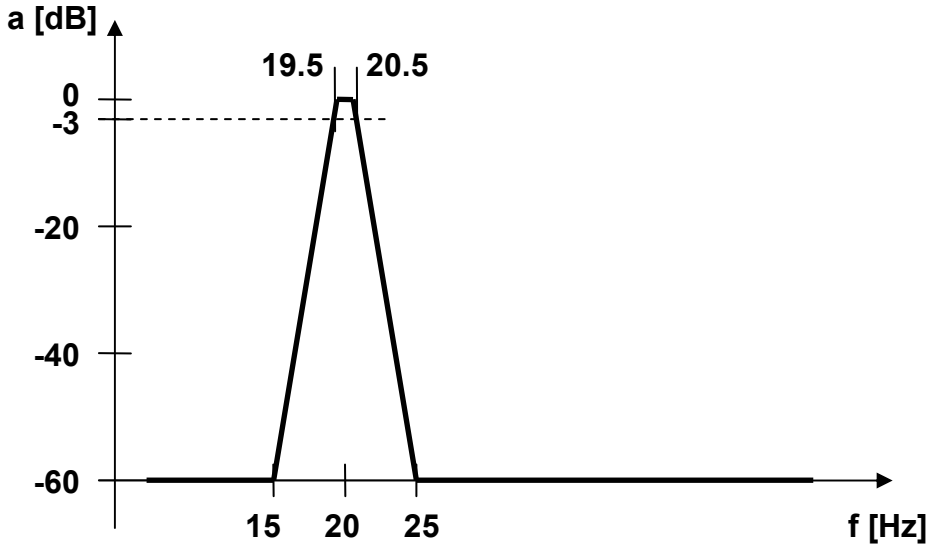


Figure LM6-1 Band-pass filter specification

Design an *Equiripple FIR band-pass filter* with the above filter specification. Recall that an FIR filter is essentially a weighted average of the current and past input values. No feedback of the output is required and the filter is therefore unconditionally stable. The number of elements in the weighted sum *(filter taps)* defines the filter order. Equation LM6.1 is the z-plane transfer function of an N^{th} order FIR filter:

$$T_{FIR}(z) = \sum_{k=1}^{N} b_k z^{-k} \qquad \text{(LM6.1)}$$

The coefficients b_k can be found in a number of ways. A convenient tool is the MATLAB command *fdatool* (Signal Processing Toolbox). Figure LM6-2 illustrates how

fdatool can be used to determine the required filter coefficients. A sample rate of 200 Hz has been chosen (10 times the centre frequency).

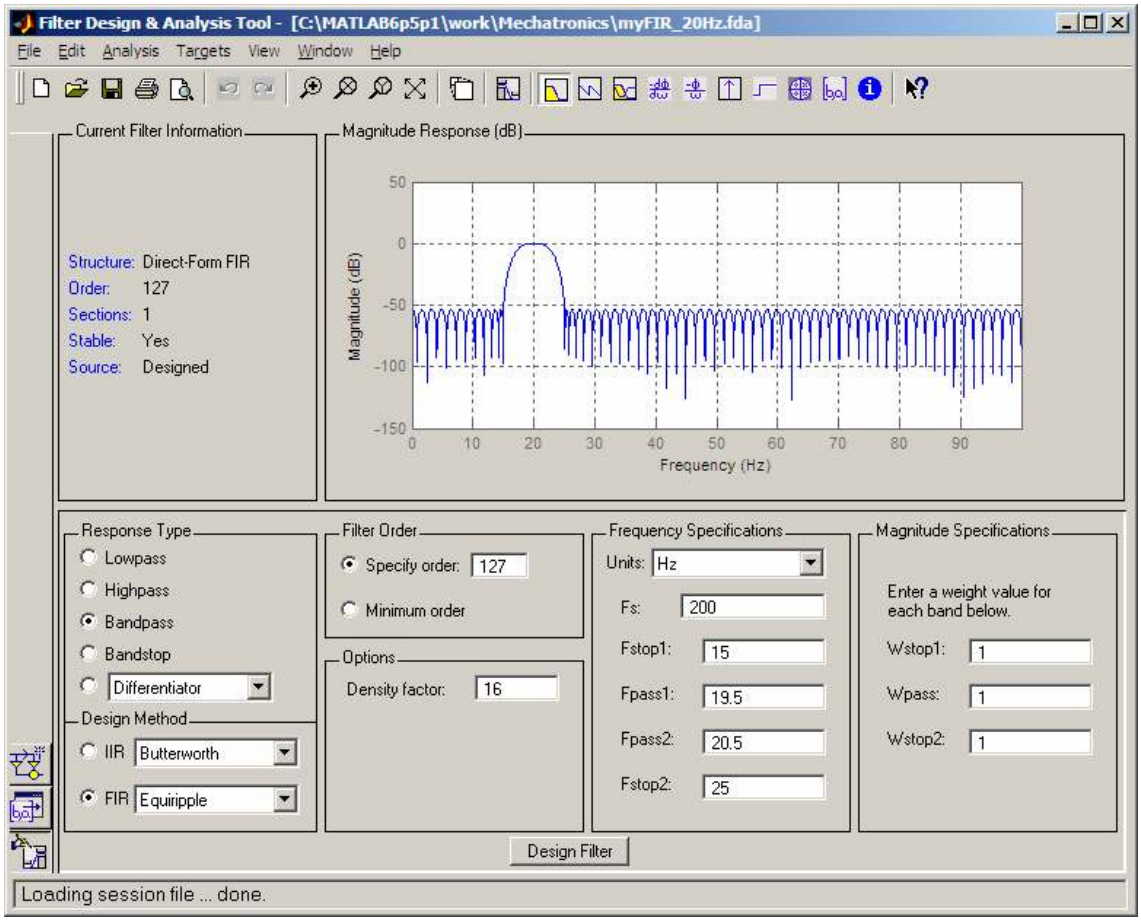


Figure LM6-2 Using MATLAB to determine the filter coefficients (fdatool)

The proposed filter has a filter order of 127. This means that at every sample instance we have to calculate the weighted sum of the 127 most recent input values, i. e. 127 multiplications plus 127 additions. Multiplications are CPU time intensive operations which often require 100s of processor cycles. We should therefore expect the microcontroller to be reasonably busy when evaluating the filter equation. This is the reason why *digital signal processors (DSP)* come with *multiply and accumulate (MAC)* units. A MAC unit can do one multiplication and one addition per processor cycle. The evaluation of a filter with 127 taps would therefore only require 127 processor cycles! The maximum bandwidth of a DSP based system is therefore much larger than what can be achieved with a standard microcontroller. Note, however, that some manufacturers now offer microcontrollers which include MAC units.

Figure LM6-3 shows the z-plane pole-zero map of the filter. Notice that MATLAB has placed 127 *poles* at the origin (p = 0). These poles are *infinitely fast* ($s = -\infty \rightarrow z = e^{js} = 0$) and are therefore essentially ineffective. They correspond to the 'missing' denominator coefficients and have only been included for formal correctness. Recall that the general filter equation can be expressed as

$$T(z) = \frac{\sum_{k=1}^{N} b_k z^{-k}}{1 + \sum_{k=1}^{M} a_k z^{-k}} \qquad \text{(LM6.2)}$$

Setting the denominator coefficients $a_k = 0$, $\forall\ k \in \{1, 2, \ldots, M\}$ turns equation LM6.2 into the FIR filter transfer function LM6.1. For M = N on the other hand, equation LM6.2 can be expressed as

$$T(z) = \frac{\prod_{j=1}^{M} (z - z_j)}{\prod_{j=1}^{M} (z - p_j)} \qquad \text{(LM6.3)}$$

where the zeros z_j are *sums of product terms* of the numerator coefficients b_k and the poles p_j are *sums of product terms* of the denominator coefficients a_k. Assuming $a_k = 0$, $\forall\ k \in \{1, 2, \ldots, M\}$ is therefore equivalent to setting $p_j = 0$, $j \in \{1, 2, \ldots, M\}$, i. e. there are M poles at zero.

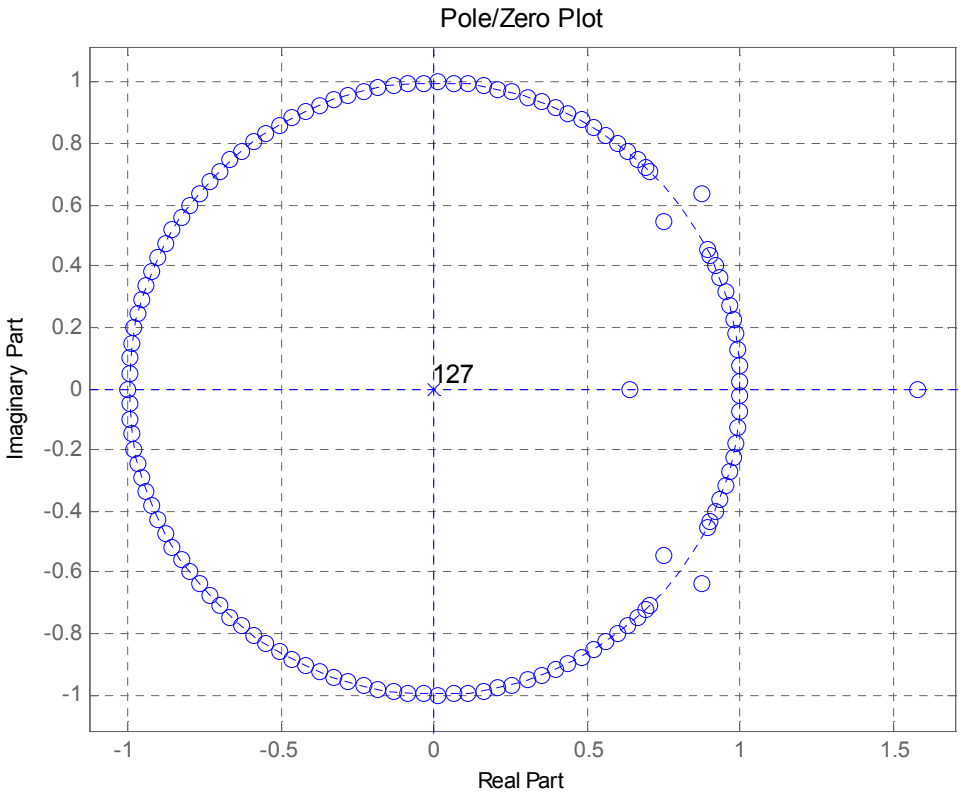


Figure LM6-3 Pole-zero map of an FIR filter with 127 taps

Use *fdatool* to produce the magnitude response as well as the phase response of the filter (Figure LM6-4). Notice that the phase response is linear within the pass-band. This is one of the advantages of an FIR filter.

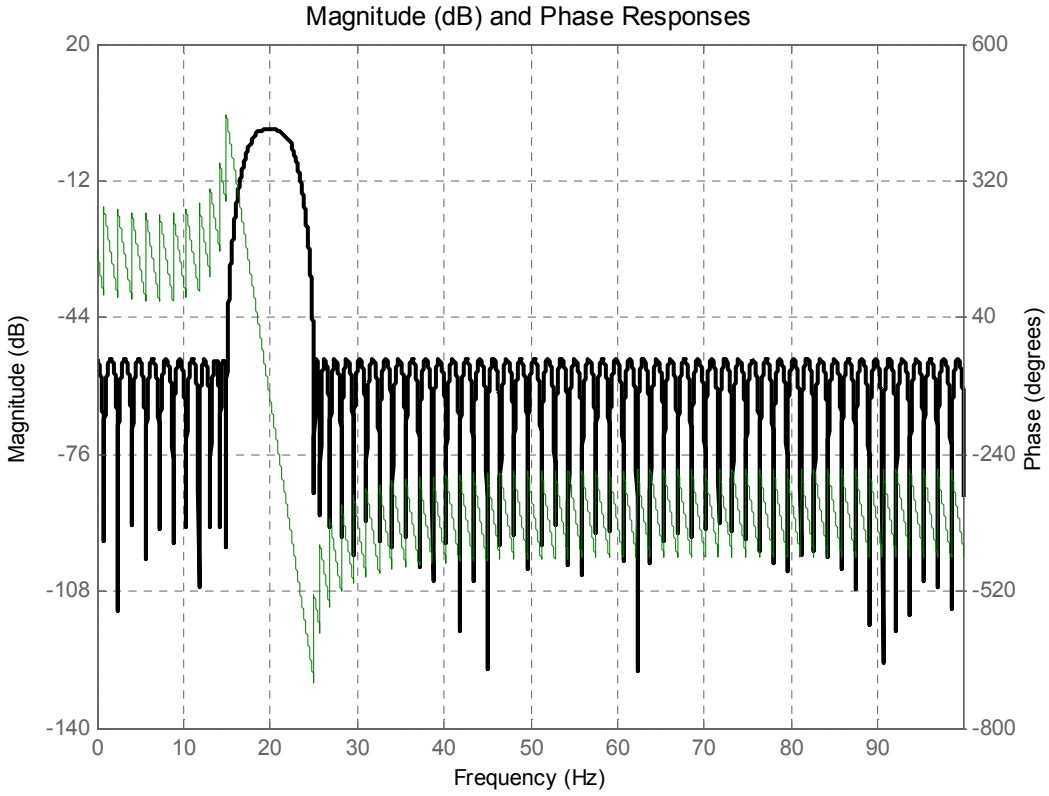


Figure LM6-4 FIR filter response, magnitude (bold) and phase (fine)

To understand why this is an advantage, have a look at the *group delay*. The group delay is defined as the gradient of phase with respect to frequency. A linear phase response therefore corresponds to a constant group delay (Figure LM6-5).

Constant group delays are important when working with multi-frequency signals such as audio signals or multi-tone transmission systems. The group delay of a system specifies how long it takes a signal group of a particular frequency to travel from the system's input to its output. When the group delay is *not* constant we will experience signal distortion. A voice signal including the vowel sequence 'u-i' (i. e. a low frequency followed by a higher frequency) might for example be inverted and end up being 'i-u' because the higher frequent/faster 'i' *overtakes* the lower frequent/slower 'u'.

The constant group delay of an FIR filter tells us by how many samples a signal is delayed when passing through the filter. Produce the impulse response of the filter (Figure LM6-6) and estimate the time at which the response has reached its maximum (t_{peak}). Can you relate this time to the group delay? How many sample times elapse before the impulse has passed the filter? How would you describe the filter's effect on the shape of the sharp input impulse?

The impulse response of a Finite Impulse Response (FIR) filter ends after N sample times, where N is the filter order. How does the group delay relate to the filter order?

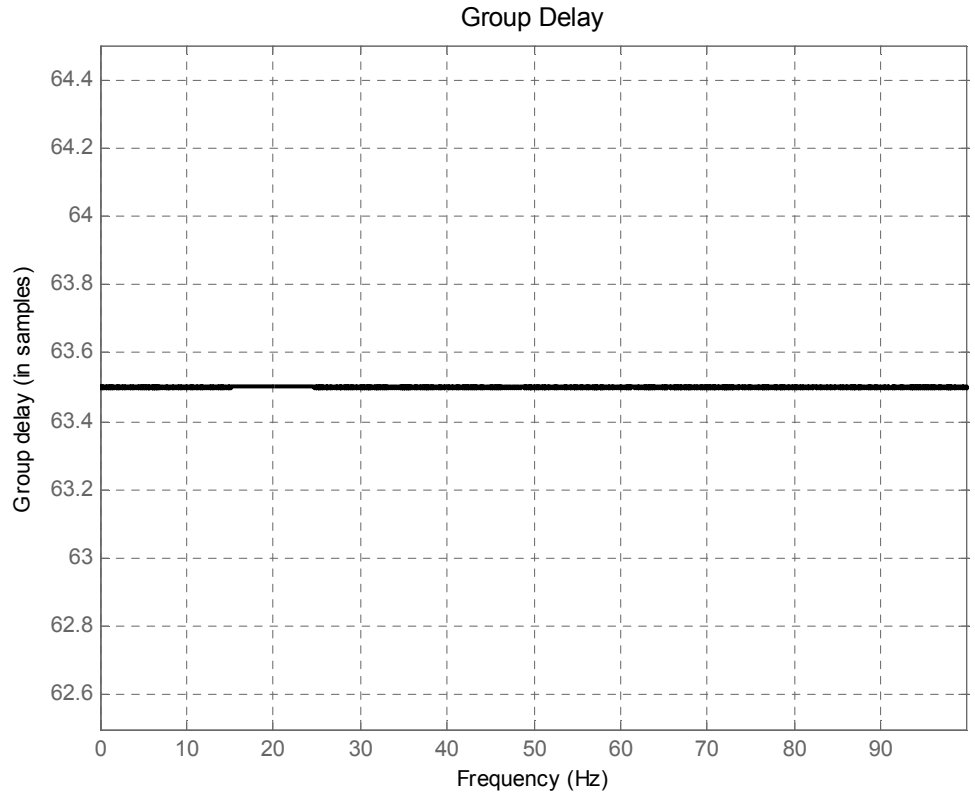


Figure LM6-5 FIR filter produce constant group delay times

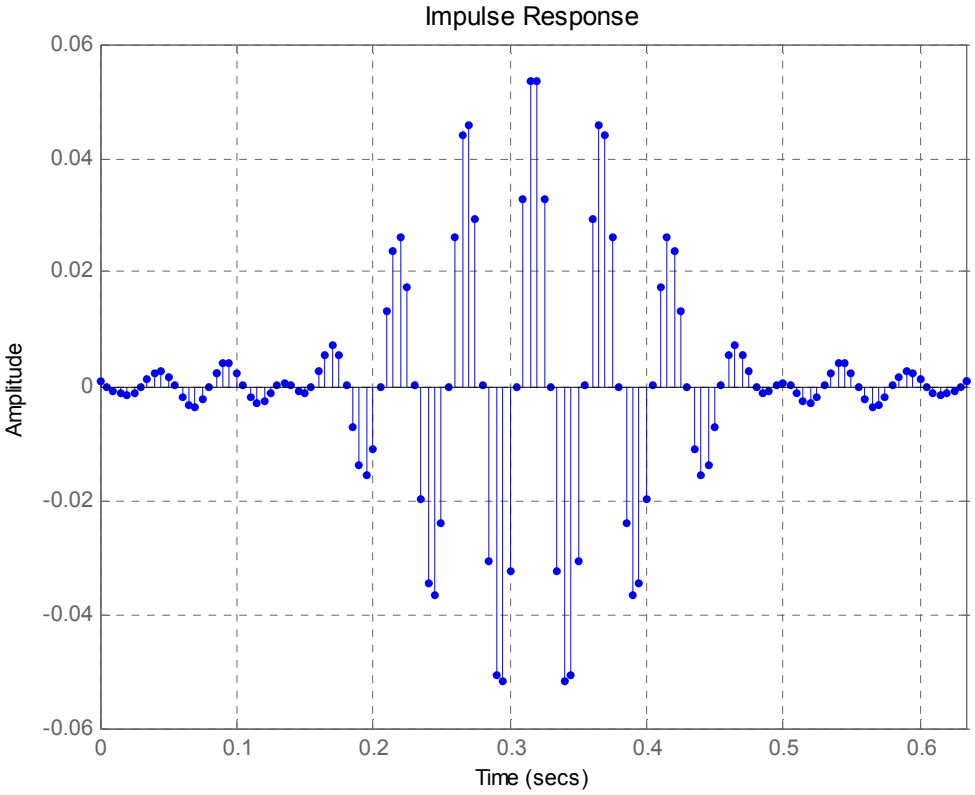


Figure LM6-6 Impulse response of an FIR filter

Implementation

We are now going to implement our FIR filter on the MC9S12DP256B/C. Inspect the calculated filter coefficients b_k. Compare the first few coefficient (b_1, b_2, b_3, …) with the last ones (…, b_{125}, b_{126}, b_{127}). What do you observe? Can this relationship be used in any way to optimize our filter implementation with regard to the required computational effort?

Select menu File → Export and export your coefficients to the MATLAB workspace (Figure LM6-7).

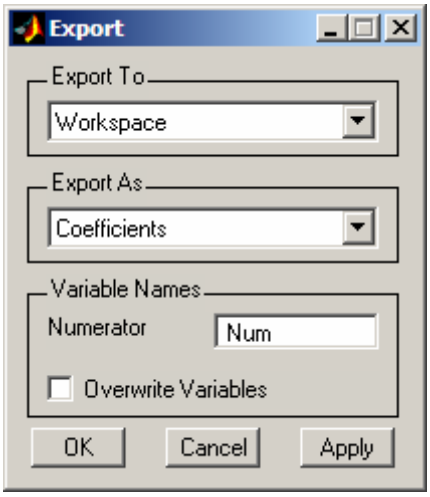


Figure LM6-7 Exporting the filter coefficients to the workspace

You should end up with a row vector variable *Num* containing the 127 filter coefficients. Convert this vector to a column vector and save it to a file using the following command line

```
>> Num = Num';
>> save myFIRcoefficients.txt Num -ascii -double
```

This command writes the filter coefficients into text file *myFIRcoefficients.txt* using a double precision format (16 digits). We will now modify this file to turn it into a C-language variable definition. Load *myFIRcoefficients.txt* into *notepad* (don't use the MATLAB editor for this) and append a comma (',') at the end of every line but the last one. Surround all coefficients by a pair of curly brackets ('{ }'). Precede the opening bracket by the C-language variable definition

```
const float b[] =
```

and terminate the closing bracket by a semicolon (';'). Save the modified file. Altogether you should end up with a variable definition similar to the following:

```
const float b[] = {   1.0041285782540202e-003,
 -2.7015465793699882e-004,
 -7.5440253582149529e-004,
…

-2.7015465793699882e-004,
  1.0041285782540202e-003 };
```

By including this file at the top of your filter program you have access to all coefficients of the filter through the one-dimensional array b[k], e.g.:

```
#include "myFIRcoefficients.txt"

      …
      y = y + b[k]*x[k]
      …
```

The next step is to turn the FIR filter transfer function (equation LM6.1) into a corresponding *difference equation:*

$$T_{FIR}(z) = \frac{Y(z)}{X(z)} = \sum_{k=1}^{N} b_k z^{-k} \qquad \text{(LM6.4)}$$

$$\Leftrightarrow \quad Y(z) = \left(\sum_{k=1}^{N} b_k z^{-k} \right) \cdot X(z) \qquad \text{(LM6.5)}$$

$$\Rightarrow \quad y[n] = \sum_{k=0}^{N} \left(b_k \cdot x[n-k] \right) \qquad \text{(LM6.6)}$$

The difference equation (LM6.6) corresponds to the filter structure shown in Figure LM6-8. In this diagram the filter coefficients have been labelled by h(k) rather than b_k.

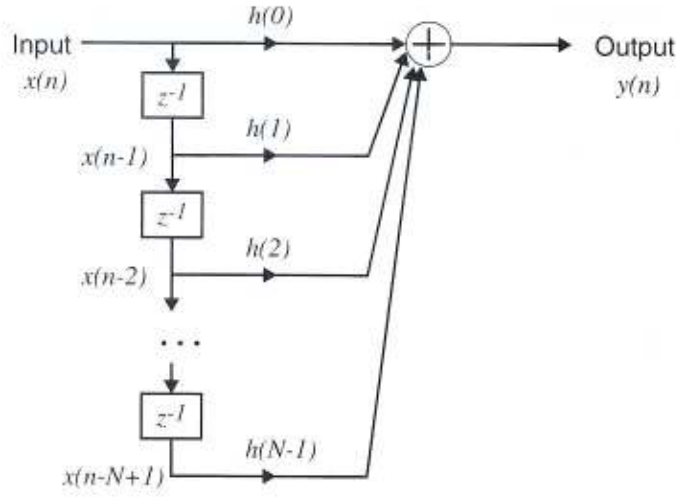


Figure LM6-8 FIR filter structure (direct form)

The above structure can easily be implemented in form of a simple for(;;) loop:

```
out = 0;
for(k=0; k<N; k++) {

  /* contribution of input value x(n-k) */
  out = out + b[k] * inp[N-1-k];

}
```

This assumes that the N most recent input values have been stored in the one-dimensional array variable *inp* in ascending order from the oldest to the most recent input value, i. e. inp[0] = x(n-N+1), inp[1] = x(n-N+2), …, inp[N-1] = x(n). The current output value of the filter is calculated and stored in *out*. Figure LM6-9 illustrates the use of array *inp[ ]*.

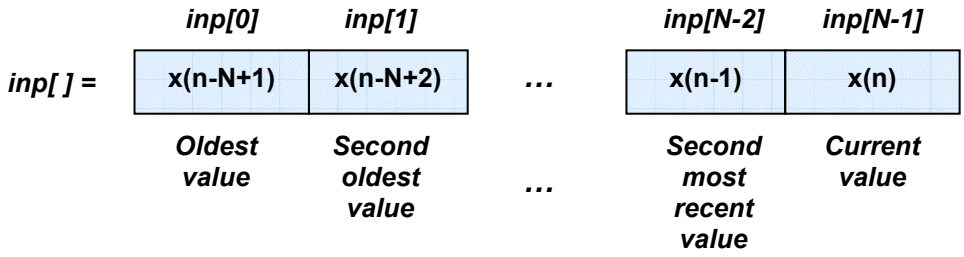


Figure LM6-9 Storing previous input values in an array variable

Once the output has been updated (i. e. output value *out* has been sent to the DAC) the filter program becomes dormant until the next sample instance occurs and a new value is read from the ADC. At this point we need to ensure that the input value array is updated. The new input value is to be stored in the element containing the current value *(inp[N-1])*. However, the previous content of this element can not just be overwritten but has to be shifted to the element of the 'second most recent value' *(inp[N-2])*. This process continues until the entire array has been shifted (Figure LM6-10).

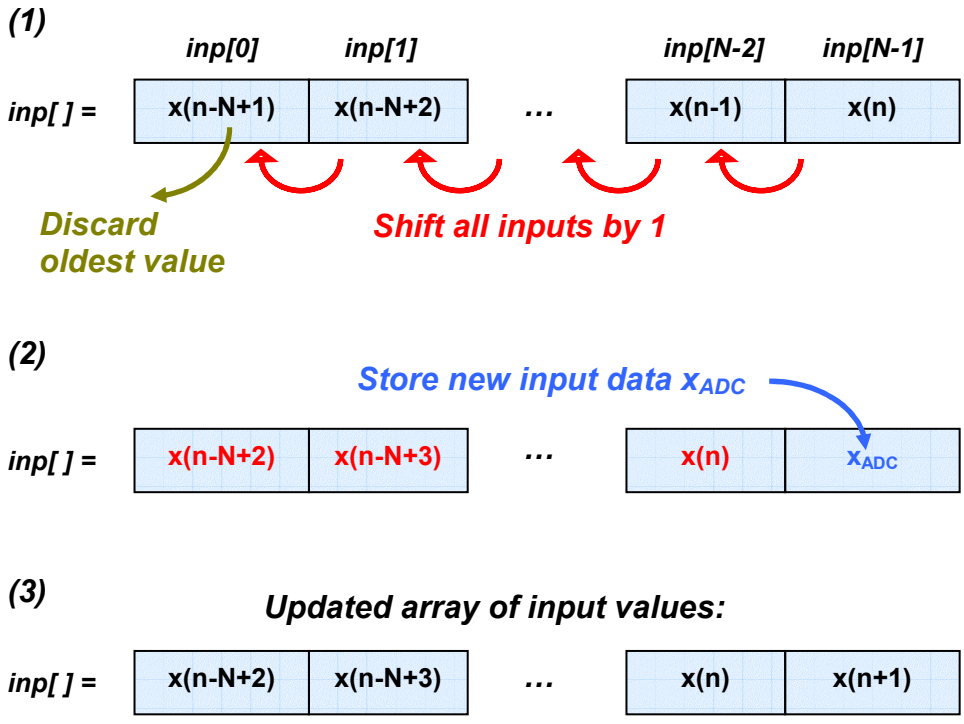


Figure LM6-10 Updating the input array inp[ ]

A C-code fragment that implements the above scheme is shown below:

```
/* inp[0] = inp[1], inp[1] = inp[2] … inp[N-2] = inp[N-1]*/
for(k=0; k<N-1; k++) {

  inp[k] = inp[k+1];

}

/* set last element to new input value */
xn[N-1] = ADC_Read();
```

This is not a very elegant solution as a lot of the already limited computing power of the microcontroller is wasted on unnecessary data movements in memory. A better approach would be to implement the tap delay line of the filter as a *ring buffer*.

Ring buffers can be thought of as circular memories, i. e. they do not have a fixed beginning and/or end. A ring buffer can be accessed using freely programmable index pointers. In our case we need to keep track of where the current input value x(n) is to be stored. A simple way of doing this is to define and update an index variable *current_input*. This concept is illustrated in Figure LM6-11 for N = 16 memory cells.

The advantage of this model is that we can simply shift all input values by decreasing the index variable *current_input* by one. The *contents* of the individual cells never have to be moved!

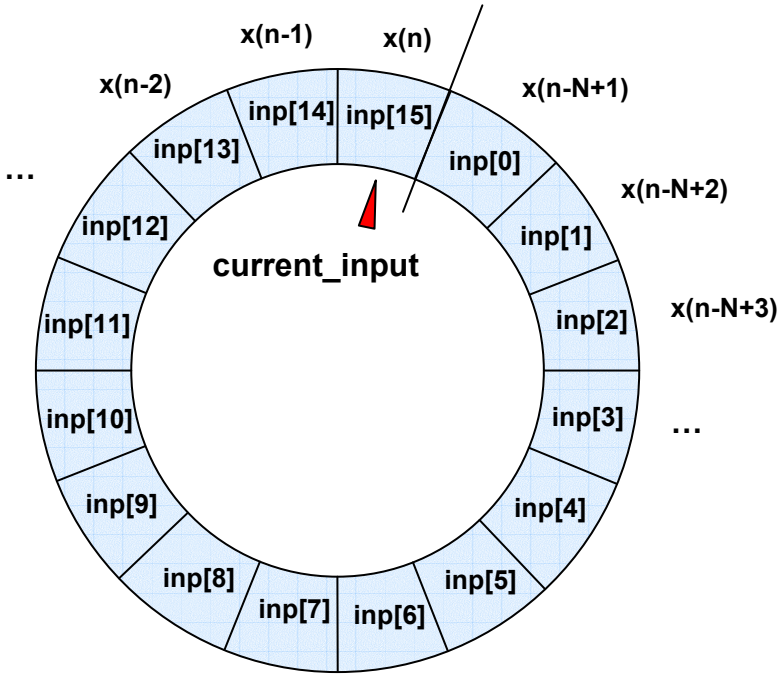


Figure LM6-11 Ring buffer structure for a 16-elements filter array

Implementing such a ring buffer is easy. All we need to do is to define a linear array and make sure that all corresponding index variables are wrapped around at both ends.

In case of a recursively updated filter such as our FIR filter, this implies the following: When a new input value is received from the ADC we increase the buffer index *'write_idx'* by one, taking care of possibly having to wrap around at the end of the buffer (Figure LM6-12). The new value is then stored in *inp[write_idx]*, thus overwriting the oldest previously stored value x(n-127).

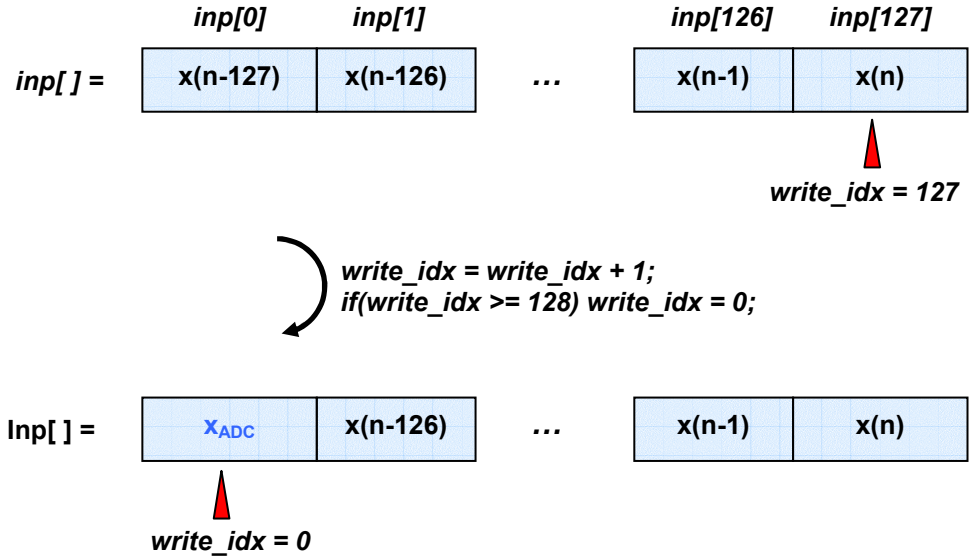


Figure LM6-12 Wrapping around at the end of the buffer

Evaluation of our filter then starts at index *write_idx* counting downwards.

To keep things neat and tidy it would be a good idea to place all buffer-related code in a separate source file *(ringBuffer.c)*, using the corresponding header file *(ringBuffer.h)* to export global variables and public functions to other source files. The following functions might come in handy:

```
void          ringBuffer_Init(unsigned int size);
unsigned int  ringIndex_Inc(int index);
unsigned int  ringIndex_Dec(int index);
```

Function *ringBuffer_Init* sets all elements of the globally defined buffer variable *inp[ ]* to zero. Function *ringIndex_Inc* increments the supplied index variable by one; the index variable is reset to zero if the buffer size is exceeded (buffer wraps around). Function *ringIndex_Dec* does the opposite thing: It decreases the supplied index variable by one; should this lead to a buffer underrun, the index variable is reset to 'buffer length minus one'.

The following code fragments should assist you in finding a solution using the above described approach. You can download an incomplete CodeWarrior project from myUni:

Mechatronics IIIM → Course Documents → Tutorials → 9S12 → FIR

Complete this project and verify that your filter complies with the given specification. Bear in mind that discrete-time transfer functions depend on the sample time. Make sure that your filter algorithm is evaluated every $1/f_S$ seconds.

ringBuffer.c

```
/* ********************* ringBuffer.c ************************
 * FW-07-04
 * Ring buffer functions
 * ********************************************************** */

#include  "fir.h"                   /* filter length (FIR_taps) */



/* global variables */
#define       rBufSize   FIR_taps    // define size
float         inp[rBufSize];         // define the ring buffer...


/********************* ringBuffer_Init ************************/
// place a value on the ring buffer
// call-up parameters: write index, data value
void           ringBuffer_Init(unsigned int size) {

int   i;

  // initialise ring buffer variable
  for(i=0; i<size; i++) inp[i] = 0.0;

}


/********************* ringIndex_Dec ************************/
// decrement ring index by one, wrap around if required
// call-up parameter: current index
// return value: decremented index
unsigned int  ringIndex_Dec(int index) {

  /* decrement index */
  index--;

  /* wrap around - if required */
  if(index < 0) index = rBufSize-1;

  /* return modified index */
  return index;

}



/********************* ringIndex_Inc ************************/
// increment ring index by one, wrap around if required
// call-up parameter: current index
// return value: incremented index
unsigned int  ringIndex_Inc(int index) {

  /* increment index */
  index++;

  /* wrap around - if required */
  if(index >= rBufSize) index = 0;

  /* return modified index */
  return index;

}
```

fir.c

```
/* ************************ fir.c ****************************
 * FW-07-04
 * FIR filter
 * ********************************************************** */

#include "fir.h"                  /* FIR_taps, b */
#include "ringBuffer.h"           /* ringIndex_Inc(), inp[] */

#include "myFIRcoefficients.txt"  /* filter coefficients, fpass = 20 Hz, filter order:
127 */



//******** FIR_filter ***************
// filter signal in 'inp' using coefficients b
// input: read index
```

```
// output: filter output (16-bit, signed)
signed short FIR_filter(int read_idx) {

int     k;
float   y;

  /* reset result variable */
  y = 0;

  /* evaluate filter */
  for(k=0; k<FIR_taps; k++) {

    /* contribution of kth filter coefficient */
    y += b[k]*inp[read_idx];

    /* adjust read index (backwards -> decrement) */
    read_idx = ringIndex_Dec(read_idx);

  }

  /* return filter output value */
  return (signed short)y;

}


//******** FIR_Init ***************
// initialise filter ring buffer
void FIR_Init(void) {

  ringBuffer_Init(FIR_taps);

}
```

timer 7 – interrupt handler

```
/* interrupt handler, timer 7 (vector 15) */
__interrupt void TOC7_ISR(void) {

signed short  filtout;

  // timing - IN
  PORTB ^= 0x80;          // toggle port B bit 7


  /* payload - start */
  {

    short Data_scaled;

    /* rescale input data to balanced range: -0x200 ... 0x1FF */
    Data_scaled = (signed)ADC_Data - 0x200;

    /* store scaled input value in the ring buffer */
    inp[current_input_idx] = (float)Data_scaled;

    /* increment current_input index */
    current_input_idx = ringIndex_Inc(current_input_idx);

    /* apply filter */
    filtout = FIR_filter(current_input_idx);

    /* rescale output data to DAC range: 0 ... 0x3FF */
    Data_scaled = (unsigned int)(filtout + 0x200);

    /* copy value to DAC */
    DAC_Write(Data_scaled);

  }
  /* payload - end */


  /* acknowledge interrupt */
  TFLG1 = 0x80;          // clears C7F

  // timing - OUT
  PORTB ^= 0x80;          // toggle port B bit 7

}
```

Testing the filter

To measure the timing of a real-time system you need an oscilloscope. At the beginning of the timer ISR, one of the digital I/O lines is set high; the line is reset to low at the end of the ISR. This produces a pulse train whose period and pulse width depend on the timer value and the duration of the control algorithm, respectively. Measure this pulse train. You should observe a signal similar to the one shown in Figure LM6-13. Does this result surprise you? What does this say about the efficiency of our filter?

Can you think of a way of improving this situation? *(Hint: It would be good if we were able to cut down on the number of multiplications within the filter loop. There is a property of the filter coefficients which we have ignored so far – try to exploit this property)*.

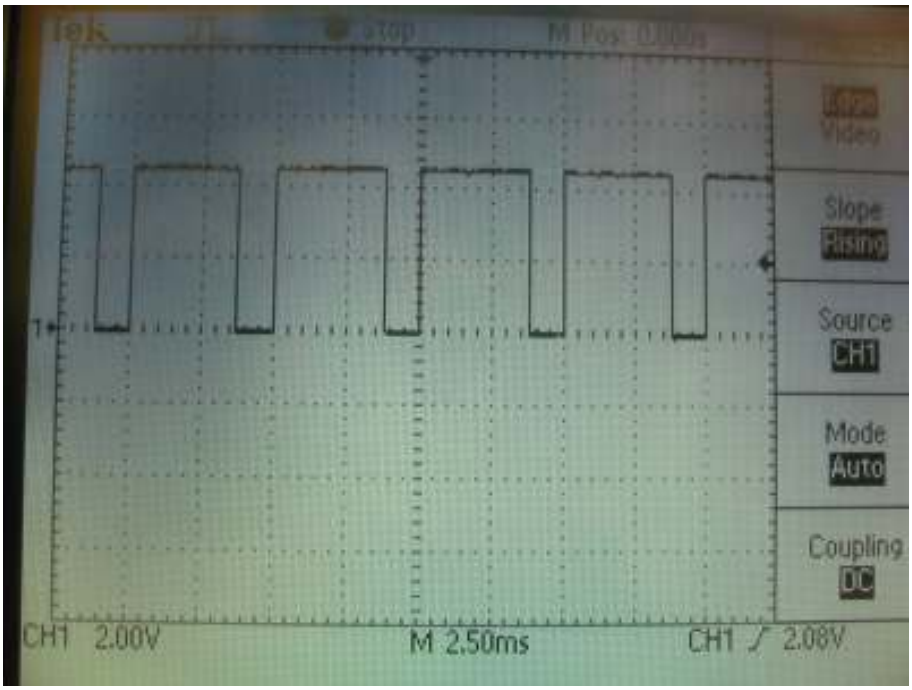


Figure LM6-13 Measuring the execution time of the filter algorithm

Multiplications are time-intensive operations. This is particularly true when working on hardware platforms which do not support *floating-point arithmetic*. Unlike digital signal processors (DSP), most microcontrollers do not include a dedicated floating-point unit (FPU) and are therefore limited to fixed-point operations using integer numbers or the software based emulation of a FPU. *IEEE standard 754* defines single precision (32-bit) and double precision (64-bit) floating-point arithmetic. This standard is the basis of the software based floating-point support provided in ANSI-C (data types *float* and *double*, implementation of arithmetic functions, complex numbers).

Inspect the filter coefficients (file: myFIRcoefficients.txt). Can you think of a way of turning these fractional coefficients into integer numbers? What dynamic range is covered by our coefficients (cf. DSP lecture notes, p. 94)? What dynamic range can be achieved on a 16-bit fixed-point architecture? Notice that these numbers carry a sign bit. Can we realise our filter as fixed-point algorithm without loss of information / the introduction of additional noise? *(Hint: Load your coefficients into the MATLAB workspace and use the functions min, max and log10)*.

The dynamic range covered by the filter coefficients can be visualised using the MATLAB commands *hist*:

```
>> b_pos = b(find(b>0));
>> b_scaled = b_pos/min(b_pos);
>> hist(b_scaled, 128)
>> xlabel('FP coefficient')
>> ylabel('frequency')
>> title('Number of occurrences of filter coefficients')
>> set(gca, 'XScale', 'log')
```

This produces the following plot (Figure LM6-14):

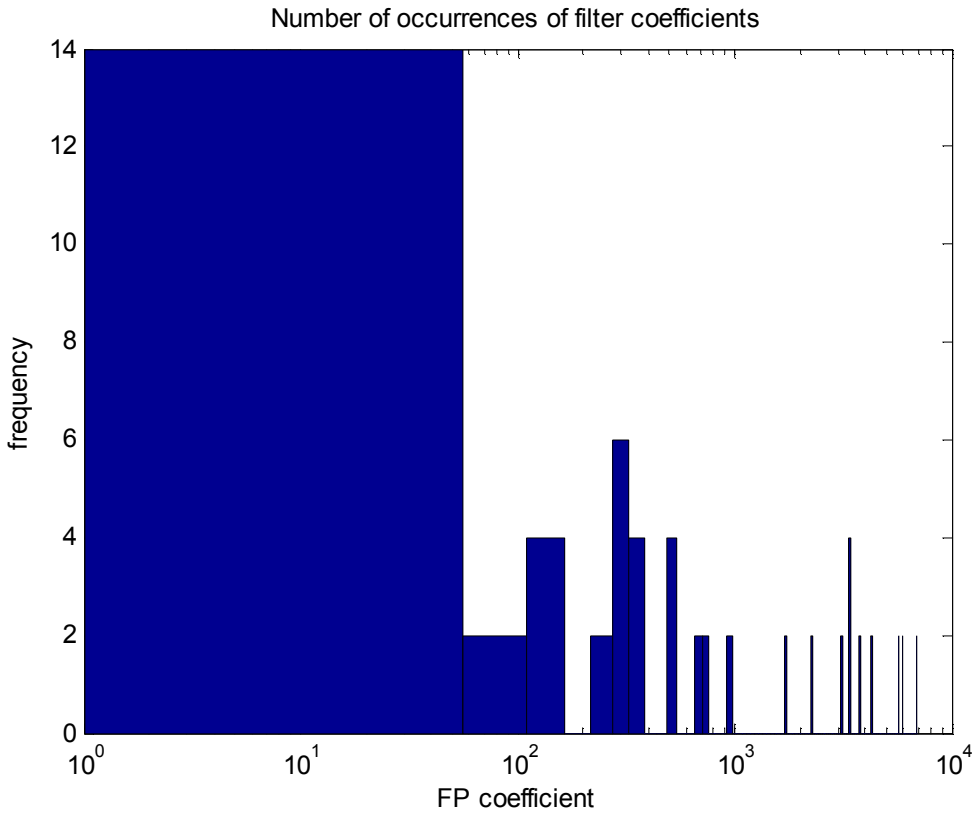


Figure LM6-14 Frequency of individual filter coefficients

It seems that most of the coefficients fall within a narrow band from 1 to 100. This can be used to further reduce the number of filter coefficients by combining neighbouring coefficients to their average value. However, be aware that such a strategy results in loss of information / the introduction of noise.

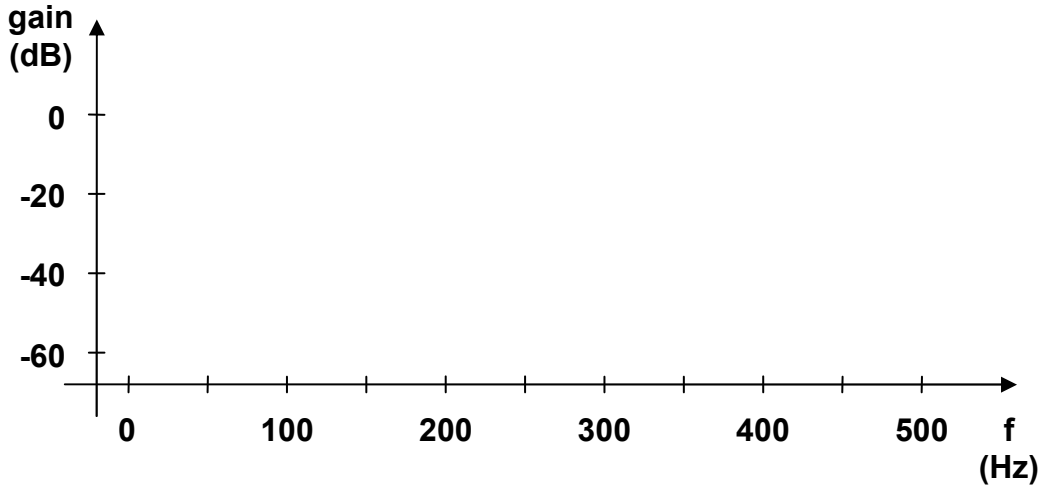
If you are keen, you could have a go at implementing the filter as fixed-point program: Re-scale all coefficients to turn them into integer numbers. Modify the structure of the ring buffer (float → integer) as well as source file *fir.c*. Measure the time it takes to update this filter. Compare the file sizes of the downloadable S19 file (/bin/Monitor.sx) of your floating-point filter program to that of the fixed-point realisation.

Slowly increase the input frequency from DC to the sample frequency (200 Hz). You should observe a narrow pass band at f_0 = 20 Hz. If your filter works properly the

output signal should be attenuated by at least 60 dB (factor 1/1000) when the frequency is below 15 Hz or above 25 Hz.

What happens when the input frequency reaches 180 Hz? Can you explain this phenomenon? What other input frequencies would lead to the same effect? Verify your answer by tuning the input signal into the suggested frequencies.

Sketch the frequency spectrum of your digital filter in the range from 0 to 500 Hz.



What would be the right thing to do in order to avoid the above phenomenon?

Concluding remarks

FIR filter have a number of attractive properties, e. g. their linear phase within the pass-band as well as their unconditioned stability and robustness. The filter is essentially an open loop system which computes a weighted average of the current and past input values.

This investigation should have made you aware of the fact that the number of filter coefficients can become quite substantial when the filter is made very selective. The evaluation of an FIR filter with many coefficients places a considerable load on the CPU of a small microcontroller. Ultimately, the bandwidth of a digital control system is limited by the number of processor cycles between any two sample instances. The evaluation of our FIR band-pass filter with 127 coefficients took around ¾ of the available period (Figure LM6-13).

A way to remedy this situation is to use a more suitable hardware platform such as a DSP processor or a microcontroller with a MAC unit. Alternatively, the FIR filter could be replaced by an *Infinite Impulse Response (IIR)* filter. An IIR filter algorithm combines the measured input signal with the computed filter output signal. This *feedback structure* can be modelled with fewer coefficients than would be needed to realise a corresponding FIR filter.

Use the MATLAB command *fdatool* to design an *Elliptic IIR* filter with the same band-pass specification as used for the FIR filter above. What is the required filter order? Export the filter coefficients to the MATLAB workspace. The default IIR filter structure of *fdatool* is sometimes referred to *'Direct Form II, Second Order Sections'*. The use of second-order sections (SOS) leads to a filter realisation with increased robustness. Display matrix *SOS* and count the number of coefficients – this should be a number much smaller than 127.

Download the following IIR filter implementation from myUni:

Mechatronics IIIM → Course Documents → Tutorials → 9S12 → IIR

Run the filter program and verify its correct operation (cut-off frequencies, pass-band attenuation, stop-band attenuation, etc.).

Determine the time it takes to evaluate the IIR filter. This can be done by measuring the ISR execution time signal on port B, pin 7 (PB7). You should observe a scope trace similar to the one shown in Figure LM6-15.

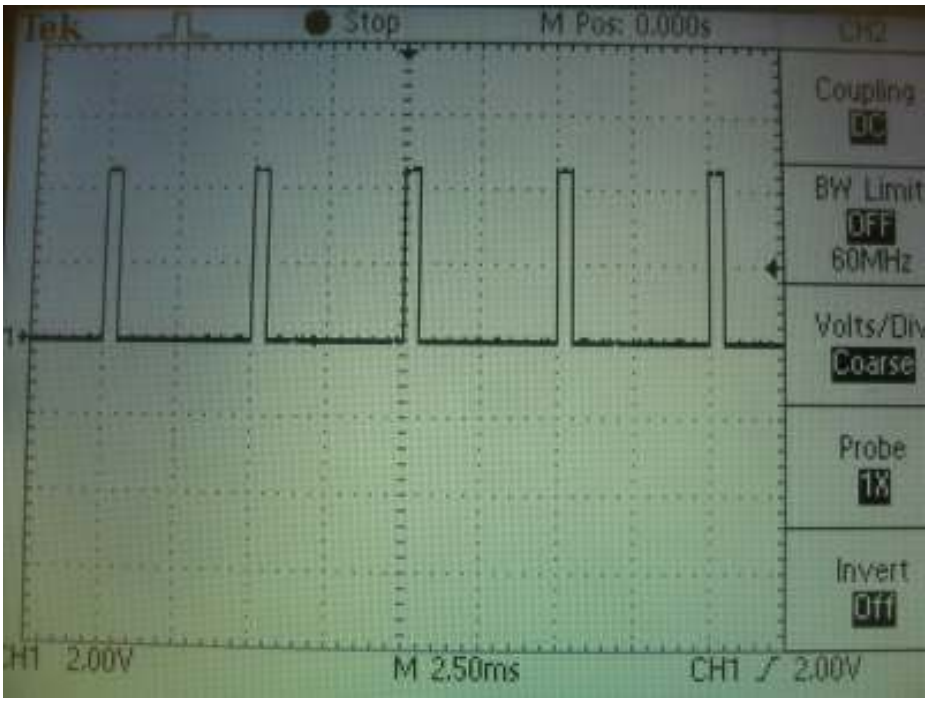


Figure LM6-15 Evaluating the execution time of the IIR filter

The same filter performance is now achieved much more efficiently.

However, it should be pointed out that the use of feedback in the IIR filter often leads to a system with a relatively slowly transient response: It takes some time until the internally circulating signals have settled onto a steady state. You can observe this by adjusting the frequency of the input signal to the centre of the pass-band (20 Hz) and then quickly changing it to a frequency within the stop-band (e. g. f > 25 Hz). With the IIR filter a settling time of around 4 seconds is observed. By comparison, the frequency step response of the FIR filter settles in approximately $\tau_s \approx 1$ second.