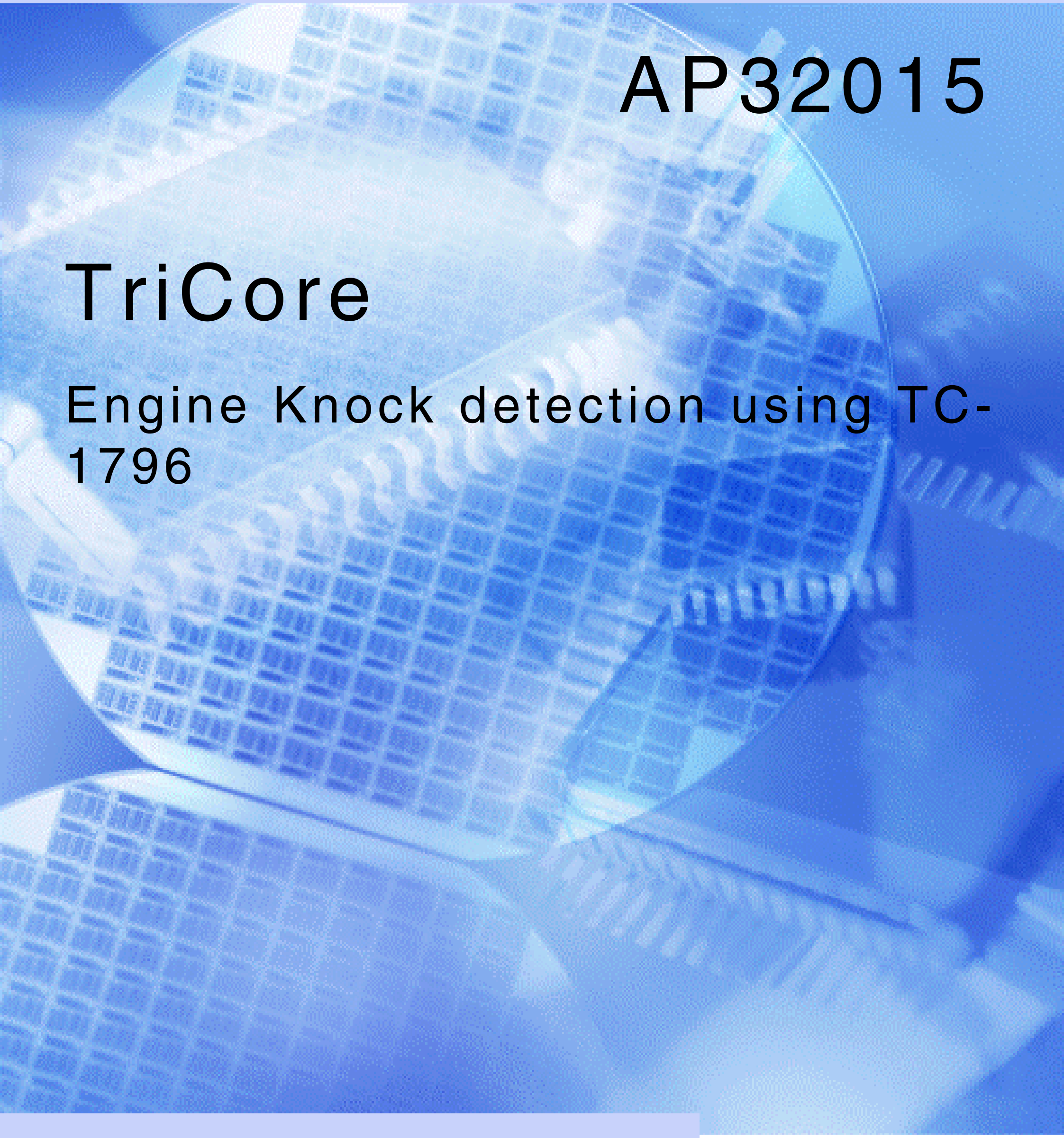
Application Note, V 1.0, Oct. 2004

AP32015

TriCore

Engine Knock detection using TC-1796



Microcontrollers

Never stop thinking.

Revision History:	2004-10	**V 1.0**
Previous Version:	-	
Page	Subjects (major changes since last revision)	

Edition 2004-10

Published by
Infineon Technologies AG
81726 München, Germany

Table of Contents

Page

1 Introduction 5

2 Engine Knock phenomena 6

3 State of the Art 7

4 Knock Detection Process 9
4.1 Generic Knock Detection Solution 9
4.2 Signal Acquisition TC1796 Implementation 10
4.2.1 Main Design Criteria 10
4.2.2 Anti-Aliasing Filter (AAF) Implementation Strategy 11
4.2.3 ANALOG External AAF 11
4.2.4 HW Internal AAF 12
4.2.5 Software AAF 14
4.2.6 Overall AAF Frequency Response 15
4.3 AMP and the Fast A/D Converter 16
4.4 DMA, DPRAM: Zero overhead data transport 17

5 Signal Processing (Knock-Detection), complete digital domain solution 18

6 Knock Detection Implementation Example 19
6.1 Additional Implementation Aspects 22
6.1.1 Knock-Detection - one of many tasks 22
6.1.2 The Relation of Efficient Algorithms AND Detection Quality 22

7 Conclusion 23

8 IN-DEPTH 24
8.1 DATA ACQUSITION 25
8.1.1 From Sensor to DMA, signal flow within FADC 25
8.1.2 FADC data format and the dynamic range 27
8.1.3 FADC Sampling Frequency setting (configuration) 30
8.1.4 Comb-Filter, the background 31
8.1.5 Comb-Filter hardware implementation on TC1796 35
8.1.6 Implementing an Anti-Aliasing Filter on TC1796 a Design Framework 36
8.1.6.1 Input Parameters 37
8.1.6.2 Output Results 38
8.1.6.3 Design Example 1 39
8.1.6.4 Design Example 2 41
8.1.6.5 Design Example 3 42
8.1.6.6 Matlab source code 44

8.1.6.7 Matlab script used as configuration input to the AntiAliasFilt() function 58
8.1.7 Implementing SW FIR +Decimator........ 61
8.1.7.1 TC1796 Source Code 61
8.2 Signal Processing (feature extraction)........ 68
8.2.1 Implementing FIR BPF on TC1796 68
8.2.1.1 TC1796 Source Code 68

9 References........ 75

10 Definitions, Acronyms, Abbreviations........ 76

1 Introduction

Knock detection capability is expected in all modern gasoline engine systems. In most systems in use taday a companion chip provides the necessary functionality.

Ever growing embedded processor capabilities allow integration of more and more functionalities; continuously eliminating external dedicated components and making the system more reliable, flexible and less expensive.

A representative knock detection solution will be explored using new Infineon TC1796 embedded processor and the necessary functionality mapped to the processor architecture. In the evaluated design example, the main functionality of the detection will be implemented using less then 4% of the CPU resources, and requiring only a simple external RC filter.

The pure digital domain implementation provides the designer with nearly unlimited flexibility. The complexity of the adopted solution is only limited by the available processor resources.

The powerful DSP capabilities of the TC1796 provide the implementation of efficient algorithms, allowing highly complex solutions to be realized for current and future knock detection designs.

It is not the objective of this Application Note to suggest any specific knock detection solution, but to demonstrate the relevance of the architecture to provide a cost effective high performance and flexible knock detection implementation platform.

The document is built of two parts:

- “Main” part describing the main design aspect and their mapping to the TC1796 architecture
- “In-Depth” part provides detailed descriptions of the architecture elements used, AAF design framework and the source code of optimal implemented DSP algorithms

The main part includes references to the “In-Depth” part allowing the desired level of details to be selected.

2 Engine Knock phenomena

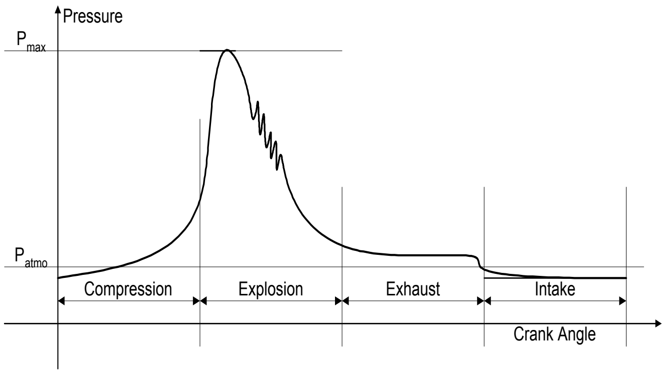


Figure 1 Knock Signal

To accomplish the proper combustion, it is necessary to work with high pressure in the combustion chamber. In this condition, high temperatures will be reached in the cylinder. The direct consequence is an abnormal combustion propagation that will quickly damage the chamber. The system must sense and correct this behavior.

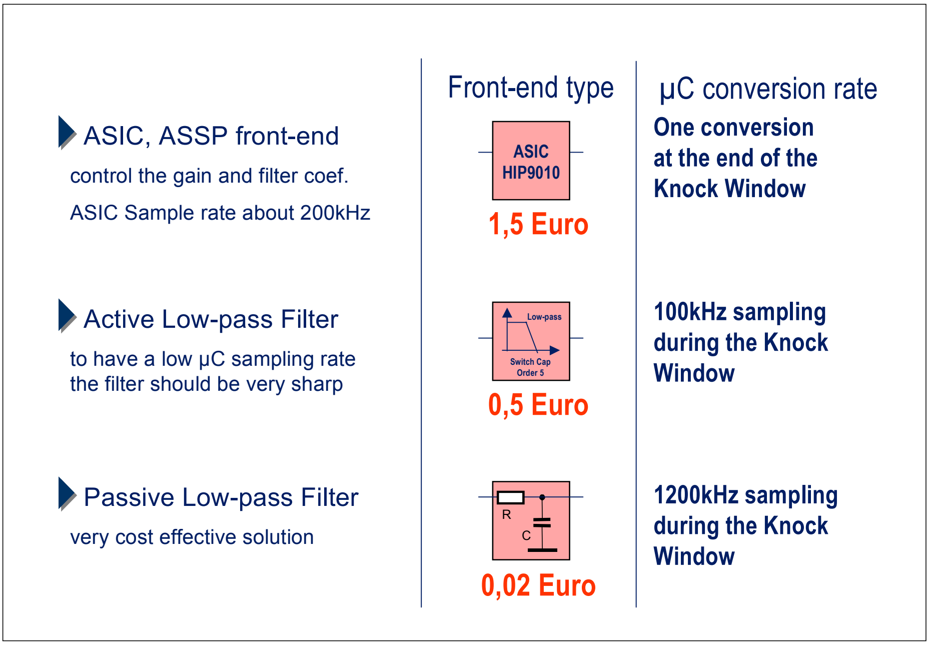
Since the knock is a random phenomena due to the dependencies of many parameters (fuel quality, manifold design, air temperature, compression ratio, air density), the system is obliged to observe each combustion cycle on a cylinder basis.

This analysis assumes that the knock detection monitoring is via a piezoelectric sensor mounted on the engine block. Many mechanical vibrations are recorded. The system has to be able to separate the vibrations caused by abnormal combustion from the normal mechanical vibration of the engine.

3 State of the Art

Knock detection is in most cases implemented using a companion chip to the main micro-controller. We can observe three types of companion chips:

- ASIC: Application Specific Integrated Circuit
- ASSP: Application Specific Standard Product
- DSP Standalone: Dedicated DSP implementation



	Front-end type	µC conversion rate
ASIC, ASSP front-end control the gain and filter coef. ASIC Sample rate about 200kHz	ASIC HIP9010 1,5 Euro	One conversion at the end of the Knock Window
Active Low-pass Filter to have a low µC sampling rate the filter should be very sharp	Low-pass Switch Cap Order 5 0,5 Euro	100kHz sampling during the Knock Window
Passive Low-pass Filter very cost effective solution	R C 0,02 Euro	1200kHz sampling during the Knock Window

Figure 2 Solution Clusters

For the cases of ASIC and ASSP the detection principle is fixed. The user can influence a few parameters but not the essential detection principle. Usually some external components are required. In the case of a dedicated DSP standalone the user has some freedom of implementation and configuration but needs lot of additional components which dramatically increase the cost. In addition this solution can very quickly runs into a communication bottleneck with the main micro-controller.

The engineers have very quickly recognized the potential to run the knock detection on the main micro-controller. Two major obstacles have been identified, the first is the availability of a micro-controller with enough data throughput and DSP performance, and the second is a cost-effective solution. Some first implementations were using active anti-aliasing filters in order to limit the sampling frequency in the 100 kHz range. This filter has been always seen as a compromise as it requires lots of components, so there is a clear objective: to reduce the anti-aliasing filter to the simplest form of the "RC-Filter".

The new Infineon TC1796 microcontroller extends the architecture to enable complete knock detection functionality with only a simple RC filter. If a differential input is required then a matching RC filter structure should be used.

4 Knock Detection Process

4.1 Generic Knock Detection Solution

In principal a knock-detection solution can be partitioned to three main blocks (as described on Figure 3). Its structure is derived from generic pattern recognition model.

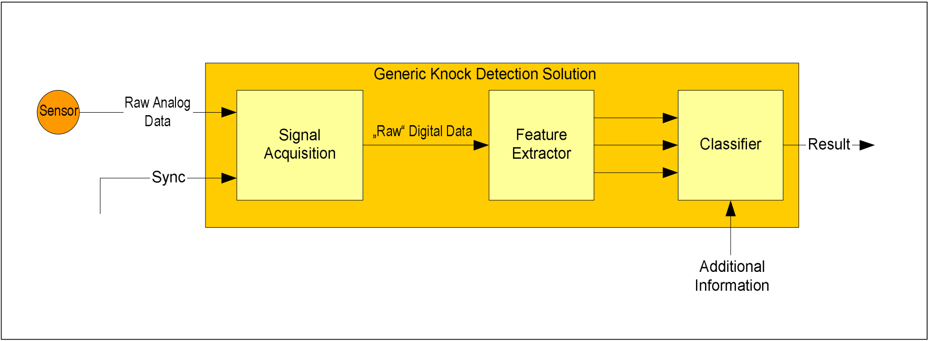


Figure 3 Generic Knock Detection Solution

In Figure 3 the following functional block can be identified:

- Signal Acquisition- primary function is to convert the physical vibration signal to an electrical signal for further processing.
  Typical elements:
  - Sensor to convert vibration to electrical signal
  - Low Pass Filter
  - Amplifier
  - A/D converter, in case further processing is made in the digital domain
- Feature Extraction (information reduction) - Extract the relevant information from the raw data which is necessary to correctly classify the signal (Knock Level). The most essential aspect is to identify and extract the signal characteristics which best represent the phenomena (Knock). In the best case the output is limited to only one feature, for example energy value in defined frequency range. In other cases, more features are required, for example energy in few frequency ranges.
- Classifier – Strives to provide the correct interpretation to the set of input features. In the simplest case, when one feature is available, the classification will be reduced to comparison of the feature value to some defined threshold.

In the following chapters describe the mapping of the functional blocks of Figure 3 to the TC1796 architecture. Initially the first two blocks will be in the focus for further evaluation

- Signal acquisition
- Signal processing

Both blocks require special architecture support beyond the simple CPU resources.

4.2 Signal Acquisition TC1796 Implementation

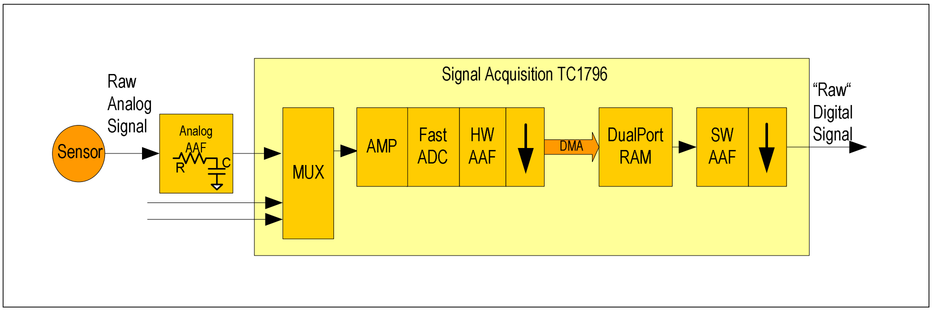


Figure 4 Signal Acquisition TC1796

The Signal Acquisition part receives the sensor signal, converts it to digital representation and stores it in memory. This is just a rough functional description. By a detailed study the following components can be recognized:

- AMP
- Fast A/D
- Three types of AAFs (Anti-Aliasing Filters)
- DMA
- Dual Port RAM (DPRAM)

4.2.1 Main Design Criteria

- Minimum of external components
- Relativly small constraint on the usable signal bandwidth
- Minimum constraints on the Knock Detection implementation details
- Reasonable usage of CPU resources
- Support of synchronic signal processing

4.2.2 Anti-Aliasing Filter (AAF) Implementation Strategy

Figure 4 shows signal acquisition part which includes three types of anti-aliasing filters (low-pass filter).

- Analog AAF: external analog low-pass filter
- HW AAF: hardware digital low-pass filter
- SW AAF: software digital low-pass filter

4.2.3 ANALOG External AAF

The desire to use minimal external components has an impact on the design, however the implementation must also fulfill the signal processing constraints.

Nyquist Theorem

A sampling frequency Fs will permit signals of interest to be reconstructed at one-half the sampling frequency. If sampling rate is 100 kHz, for example, then signals below 50 kHz can be reconstructed and reliably analyzed.

Usually the sensor signal spectrum cannot be predicted, so to prevent the aliasing effect into the band of interest (i.e. 0-35 kHz for knock detection), the signal has to be filtered by a low pass filter (anti-aliasing filter AAF) with sufficient damping before being analyzed. This analog filter could be specified i.e. with the following anti-aliasing conditions:

maximal -3dB at 35 kHz

minimal -25db at 50 kHz

Limiting the spectrum to 50 kHz will allow working with sampling frequency of 100 KHz without aliasing effects.

If the evaluation is made in opposite direction then the question is “which sampling frequency for a given minimal hardware of a simple RC filter is required?”

The First order filter in Figure 5 has 6dB/Octave attenuation relative to the 3 dB corner frequency.

Assuming corner frequency of 35KHz then 25 dB attenuation will be achieved after about 4 octaves, which means 16 times the corner frequency (or 16*35 = 560kHz).

To prevent aliasing effect, the sampling frequency should be 1.12 MHz. Therefor, if we would like the option to work with a simply external filter we would require that the A/D converter should be able to work in MHz range!

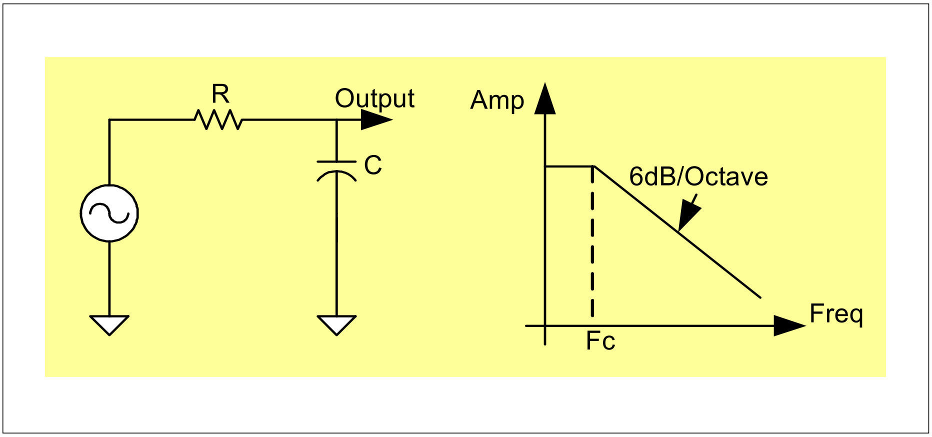


Figure 5 First Order Filter

Implementing the Knock Detection algorithms in software with Fs above the 1MHz will be very inefficient and a waste of CPU resources. The same algorithms working with 1/10 of Fs will require 1/10 of the CPU resources. It is obvious, that some additional AAF and sampling reduction (decimation filter) is necessary.

4.2.4 HW Internal AAF

Decimation or data reduction filtering of a source input of data samples is often done in hardware. This is advantageous because of the reduced load of the subsequent signal processing.

Normal digital filtering uses 3 operations; these are delay, multiplication of the samples with certain predefined weights, and the summation of the intermediate results to gain final results.

However the implementation of a multiplier structure on silicon for this application is not cost efficient due to the needed chip size area. It is therefore too expensive in terms of system costs. For this reason solutions are needed which only use of the delay and the summation operations.

There is one possible solution that uses the summation of equally weighted samples. Such a FIR filter with all N coefficients equal to one is called comb filter (see 8.1.2 and 8.1.5.) There is an efficient recursive form that makes the number of additions independent of filter size.

Two comb filters with programmable order and decimation factors have been implemented. Cascading both of filters provide further highly flexibile methods of implementing various filter characteristics.

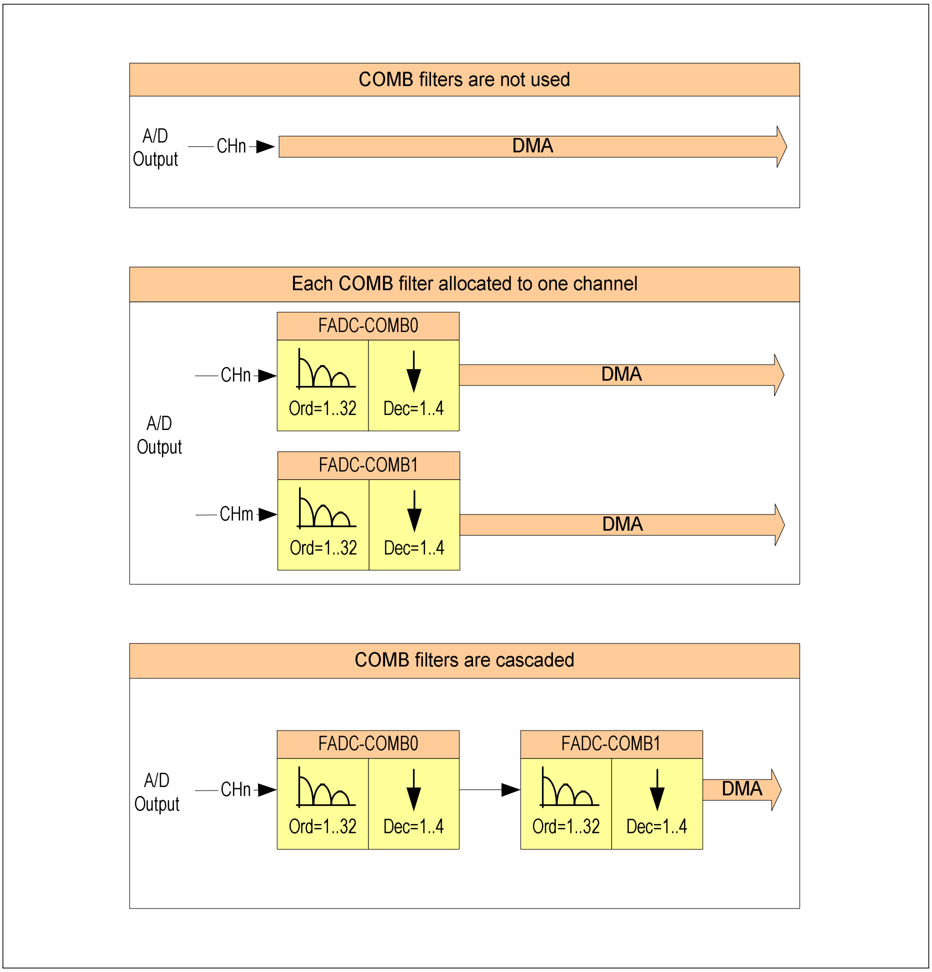


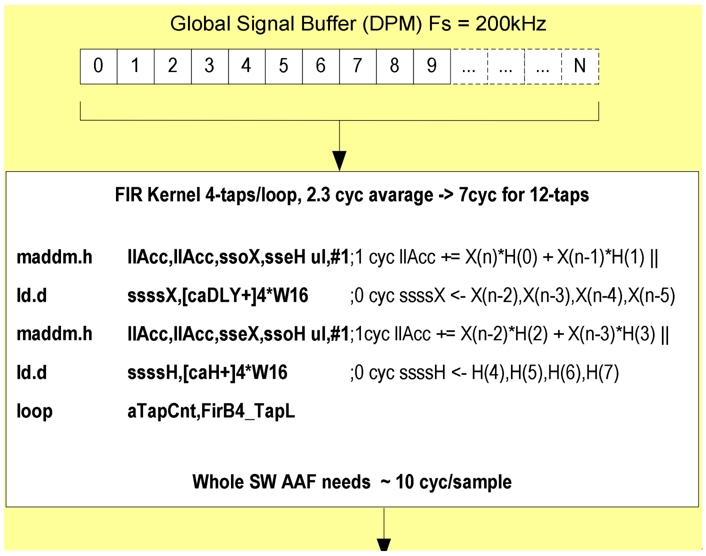
Figure 6 FADC comb filter various configuration

Figure 6 shows various possible configuration of the FADC signal flow. In the first case the comb filters are not used and the A/D output data are directly transferred by the DMA unit. In the second case one comb filter is used. If required, two channels can be processed in parallel, each one using one of comb filters. In the third case two cascaded filters are used.

4.2.5 Software AAF

The implemented FIR filter does not have its own delay-line but instead works directly on the Global Signal Buffer, making the processing more efficient (see 8.1.6 and 8.1.6.3). As only every second sample is required for further processing (down sampling by 2) each second output will be calculated. So effectively the filter is processing the signal in 100 kHz rate.

When working continuously the filter requires 12cyc/sample when evaluated on the output sampling rate (100 KHz). This means 12*100kHz = 1.2 Mcyc/sec of the CPU. Assuming TC1796 working with 150 MHz clock frequency -> (1.2/150) * 100 = 0.8% of the CPU resources will be used (see 8.1.7).



FIR Kernel 4-taps/loop, 2.3 cyc avarage -> 7cyc for 12-taps

```
maddm.h   llAcc,llAcc,ssoX,sseH ul,#1;1 cyc llAcc += X(n)*H(0) + X(n-1)*H(1) ||
ld.d      ssssX,[caDLY+]4*W16       ;0 cyc ssssX <- X(n-2),X(n-3),X(n-4),X(n-5)
maddm.h   llAcc,llAcc,sseX,ssoH ul,#1;1cyc llAcc += X(n-2)*H(2) + X(n-3)*H(3) ||
ld.d      ssssH,[caH+]4*W16         ;0 cyc ssssH <- H(4),H(5),H(6),H(7)
loop      aTapCnt,FirB4_TapL
```

Whole SW AAF needs ~ 10 cyc/sample

Figure 7 SW AAF implementation

4.2.6 Overall AAF Frequency Response

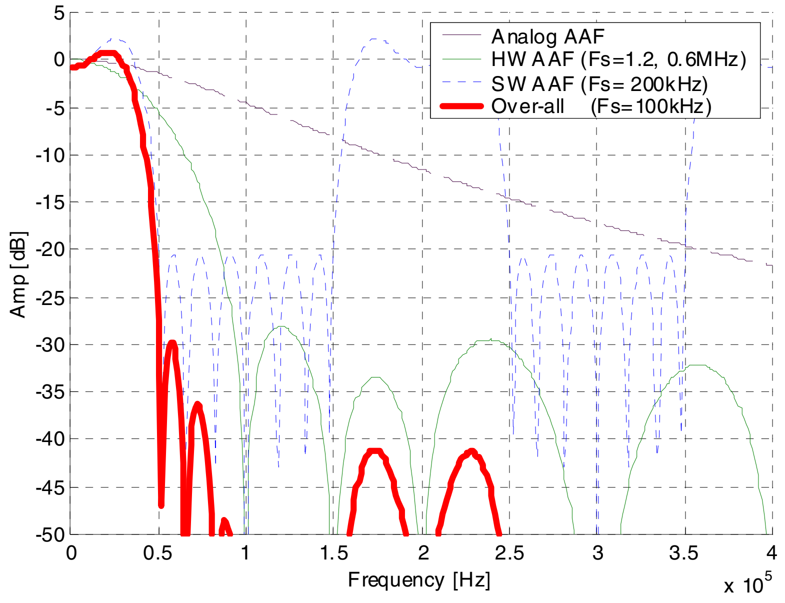


Figure 8 Overall AAF Frequency Response

The red curve on Figure 8 shows the overall AAF filter frequency response. In this implementation attenuation of stop-band range of about 30dB at 50kHz could be achieved. For further design cases see 8.1.6

4.3 AMP and the Fast A/D Converter

The amplifier (AMP) and the A/D converter have a direct impact on the characteristic of the conversion process from the analog domain to the digital domain. The amplifier also allows differential inputs for noise reduction.

The speed of the A/D converter together with the decimation filter either has an influence on the sampling rate or indirectly decides about available frequency range. In theory the available frequency bandwidth is ½ of the sampling rate but in a real application it is dependent on the anti-aliasing filter characteristic. High sampling rates guarantee a high frequency bandwidth which in some cases includes valuable information. Usually selecting the correct frequency is a very important factor which can have a crucial impact on the quality of the detector. It should be clear that a high sampling rate has a direct impact on the required resources. TC1796 has a flexible sampling rate setting which allows optimal sampling rates to be selected. The output of the decimation filter is a stream of digital data which is managed by the DMA unit.

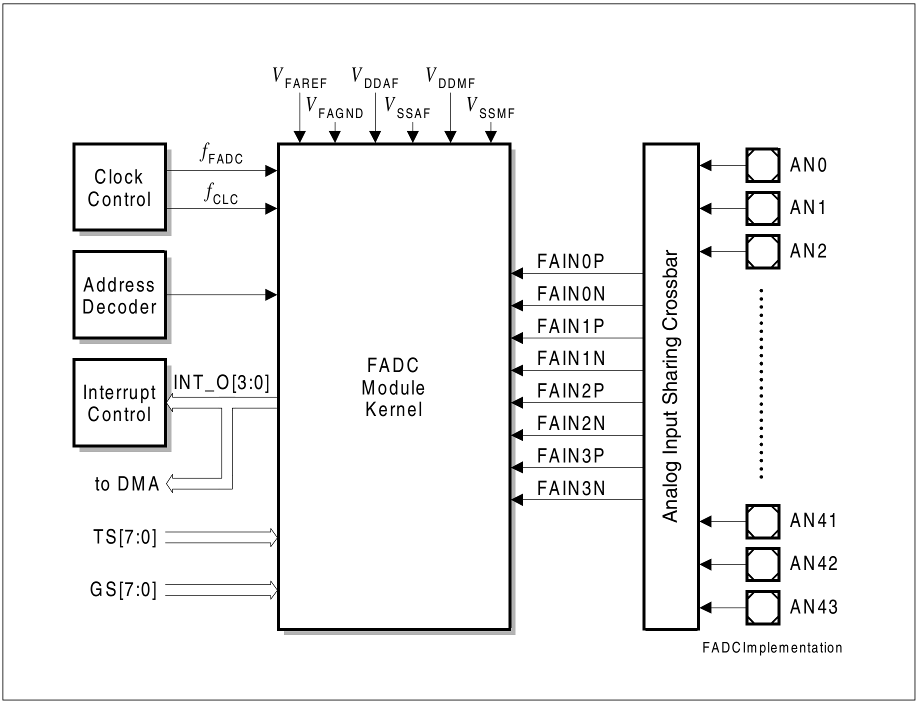


Figure 9 Fast A/D converter (FADC)

The A/D converter features

- 10 bit resolution
- Fast conversion of min. 280nsec (3.5 Msamples)
- 4 differential inputs channels
- Differential input amplifier with programmable gain of 1, 2, 4, 8 for each channel
- Free running (channel timer) or trigger conversion modes
- Trigger and gain control for external signals
- Built-in channel timers for internal triggering
- Channel timer request periods independently selectable for each channel
- Selectable, programmable anti-aliasing and data reduction filter block

4.4 DMA, DPRAM: Zero overhead data transport

Knock detection processing modules (software algorithms) working with typical sampling frequencies in the 100kHz range consume a significant amount of data which must be transported from the A/D converter to the internal data memory.

TC1796 will manage all the conversion results without consuming any CPU load. The DMA will transfer the samples into the Dual Ported RAM (DPRAM). An interrupt will be issued at the end of the block transfer. Using of DPRAM allows concurrent usage of fast internal memories by the CPU and the DMA unit without any performance impact.

5 Signal Processing (Knock-Detection), complete digital domain solution

Most resources and intelligence is included as part of the detector. All the implementation is made in software, so in principal an unlimited design flexibility is provided! To some degree this is also true in the real implementation. We can say unlimited flexibility but with limited complexity, as with each software implementation there are limited processor resources available. Some limit on complexity is expected as real-time performance should be assured. The blocks included in the diagram present the expected processing blocks which are used in the dedicated knock detection product. It is very probable that each solution will include energy measurement in selected frequency bands. FFT DFT FIR IIR or any other algorithms will be adequate for this [0]. Based on available products and research it is known that typical DSP algorithms will consume most of the resources. TriCore has a very powerful DSP capability which allows highly complex implementations to be realized for current and future knock detection designs.

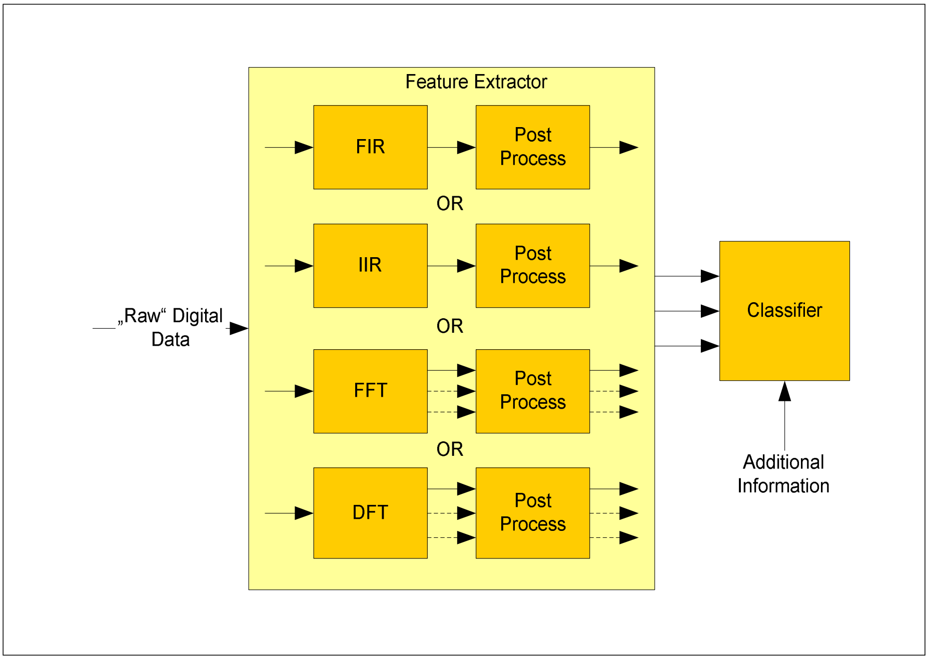


Figure 10 Examples of feature extraction (Signal Processing)

6 Knock Detection Implementation Example

To make some estimation of the necessary CPU resources, a state of the art Knock Detection principal will be evaluated [0]

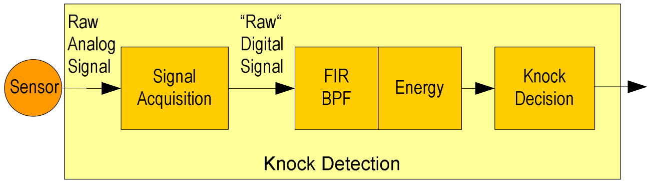


Figure 11 Knock Detection

The sensor signal that is converted by an A/D will be processed by a band-pass filter with frequency of interest. Finally the signal energy will be calculated and compared to a selected reference value. The filtering and energy evaluation will be synchronic to the knock-window that is managed by the signal acquisition unit and provided to the FIR/Energy unit .

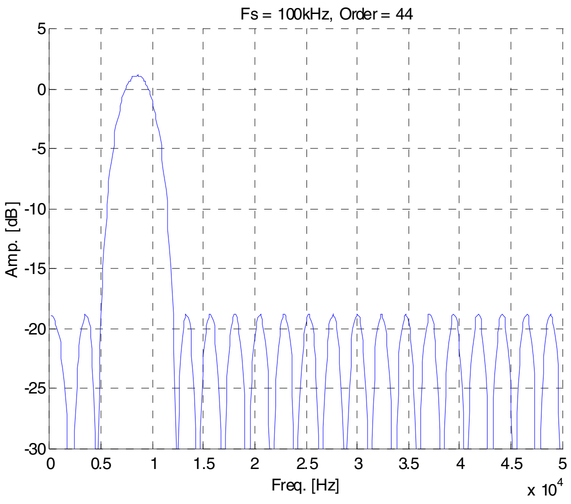


Figure 12 Frequency response of FIR BPF 44-taps

Filtering of the signal should be made synchronically to the “Knock Window”. Using FIR filter working directly on the “Global Signal Buffer” (without internal delay-line) allows fine synchronization without transient effects (same as SW AAF). Figure 13 shows the starting point of the filtering which is directly using the first 44 samples (0..43) providing stable a output [0]. Block processing is much more efficient than per sample processing, therefore the real processing will be not evaluated directly on the received signal but first buffered in “Global Signal Buffer” and then processed.

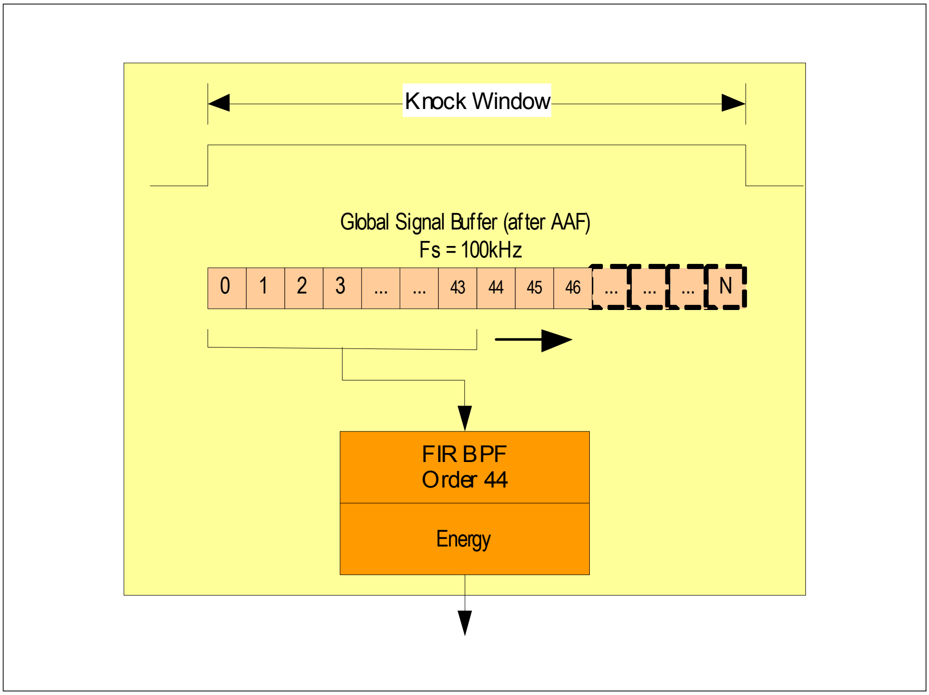


Figure 13 Synchronic Signal Processing

The kernel implementation is very similar to the SW AAF so the expected performance is directly proportional to the evaluation showed in Figure 7. Some additional processing is required for the energy measure but it is small relative to the filter implementation.

Overall the resources required for the filter (FIR, size = 44) including energy calculation are 38 cyc/sample. If the processing is continuous, the sampling rate = 100KHz and TC1796 working with 150 MHz clock frequency ->

38cyc/sec*100KHz = 3.8Mcyc/sec (of CPU total 150Mcyc/sec)
(3.8/150) * 100 = 2.6% of the CPU resources will be used.

CPU resources required to execute the AAF and the FIR filter -> 0.8% + 2.6% = 3.4%

Usually the “Knock Windows” will not occupy 100% of the time so even fewer resources should be expected.

The 3.4% are more adequate for the case when two types of evaluation windows are used, Knock-Windows and the Noise-Windows in which knock does not occur. From the resource point of view this matches the cases of continuous filter processing if both windows do not overlap,. The idea behind working with Knock-Windows and Noise-Windows would be to get additional information to adapt the threshold level used for Knock-Detection classification.

6.1 Additional Implementation Aspects

6.1.1 Knock-Detection - one of many tasks

Knock detection is one of many tasks that should be processed in real time and parallel to other tasks. Powerful multitasking support is crucial for properly functioning system

The following TC1796 features can be identified

- Flexible multi-master interrupt system (Interrupt serviced by CPU, PCP or DMA)
- Hardware controlled context switching
- Hardware Interrupt Priority arbitration with 255 priority levels
- Very fast interrupt response time: 160ns min (at 150MHz, more than 63 interrupt nodes used)

6.1.2 The Relation of Efficient Algorithms AND Detection Quality

The knock detection function requires DSP algorithms (like filtering and FFT calculations) to be implemented along with the other algorithms. The DSP algorithms are the most resource consuming, as they perform a lot of calculations on the input data. The strong architectural support for DSP algorithms combined with the creativity of the developer leads to efficient implementation. Such an efficient implementation leads to effective usage of resources. The resources gained by efficient implementation provide the user with the flexibility to extend the complexity of algorithm for better quality of the application. Therefore it can be concluded that the strong architectural support of the processor for DSP algorithms leads to a better quality of knock detection.

7 Conclusion

Ever growing embedded processor capabilities allow integration of more and more functionality; continuously eliminating external dedicated components and making the system more reliable, flexible and less expensive.

The new embedded processor has been demonstrated with extended capability that allows efficient implementation of knock detection functionality with only a simple external RC filter.

A representative knock detection solution has been explored and the necessary functionality has been mapped to the processor architecture. In the evaluated example the main functionality of the detection could be implemented using only 3.4% of the CPU resources,

Pure digital domain implementation provides the designer with nearly unlimited flexibility and allows complexity that is only limited by the available processor resources.

Powerful DSP capabilities providing efficient algorithms which allow highly complex solutions for current and future knock detection designs to be realized.

8 IN-DEPTH

This part of the Application Note extends the main part with additional implementation and functionality details of the TC1796 peripherals included in the knock-detection design. The block diagrams included in this part show the specific configurations used in the design. In addition to the peripheral functional blocks, configuration parameters and the appropriate register values are included. It should be emphasized that the configuration parameters as described on various block diagrams cover only the parameters which are directly related to the specific setting. Additional global parameter settings are not included.

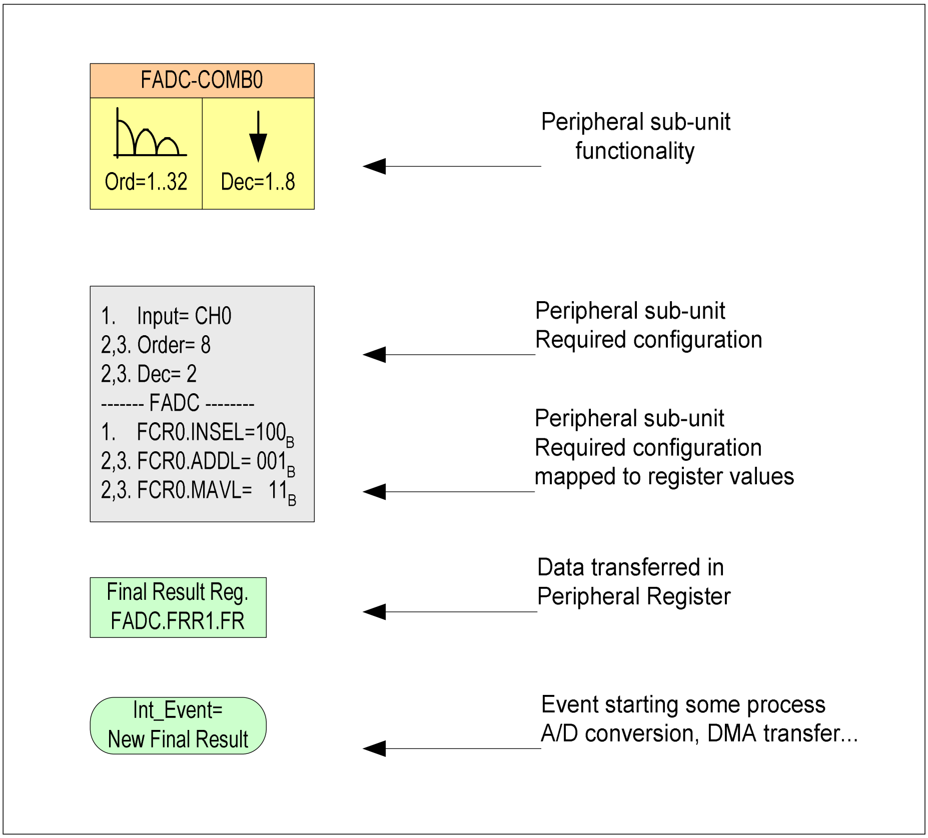

FADC-COMB0
Ord=1..32
Dec=1..8
Peripheral sub-unit functionality
1. Input= CH0
2,3. Order= 8
2,3. Dec= 2
------- FADC --------
1. FCR0.INSEL=100B
2,3. FCR0.ADDL= 001B
2,3. FCR0.MAVL= 11B
Peripheral sub-unit Required configuration
Peripheral sub-unit Required configuration mapped to register values
Final Result Reg. FADC.FRR1.FR
Data transferred in Peripheral Register
Int_Event= New Final Result
Event starting some process A/D conversion, DMA transfer...

Figure 14 Block Diagrams conventions

8.1 DATA ACQUSITION

8.1.1 From Sensor to DMA, signal flow within FADC

Figure 15 shows the main sub-modules of the FADC which process the sensor‘s analog signal.

Each of the four channels has its fixed allocated input pins. Each of the channels can be configured as single ended or differential input. If not required, both inputs lines can be disconnected. There is common amplifier and A/D converter which has a multiplexer on the input and output. The input channel will be automatically selected based on the convert request signal. At the same time the output multiplexer will be switched to the appropriate result channel. The described configuration uses two cascaded comb filters as described in 8.1.6.3 Design Example-1

The conversion will starts by the CH0 convert Req. event as depicted. When the conversion is completed the result value will be available in “Final Result Reg.”. Additionally, the “New Final Result” interrupt event will be generated which will start the DMA transfer. As in the previous block diagram, each module includes additional information of the configuration parameters and the suitable register settings.

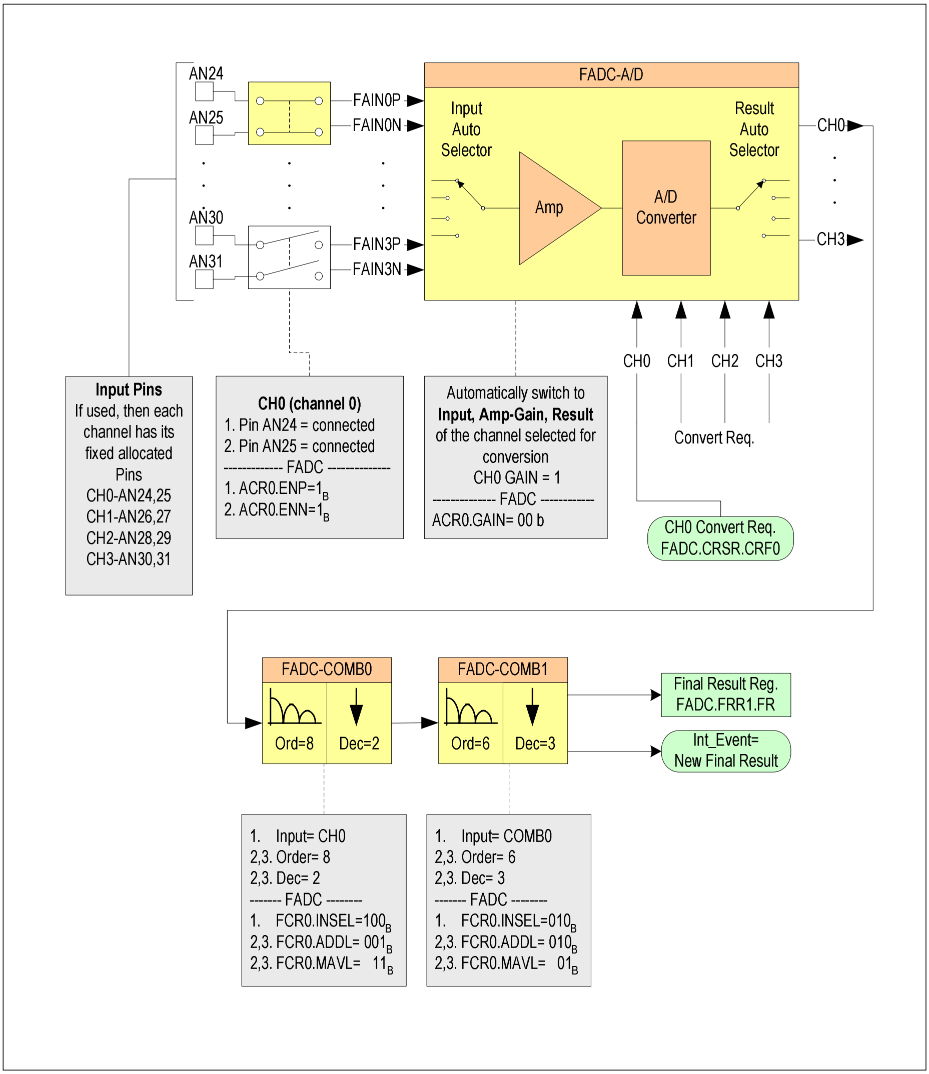


Figure 15 FADC Signal flow

8.1.2 FADC data format and the dynamic range

Figure 16 shows the relation of the sensor analog signal and the digital output format of the FADC A/D converter.

The sensor signal is represented as the digital equivalent to aid understanding. It can be observed that the analog signal can have positive as well negative values which in digital domain will be represented in signed number format. The A/D converter is of single supply type and can convert only positive input voltages. For this reason a level shifter is used and the converted input is directly mapped to the digital domain which is of unsigned type. It can be recognized that the A/D output voltage doesn't match the sensor signed format. Format conversion is required on the output of the A/D converter.

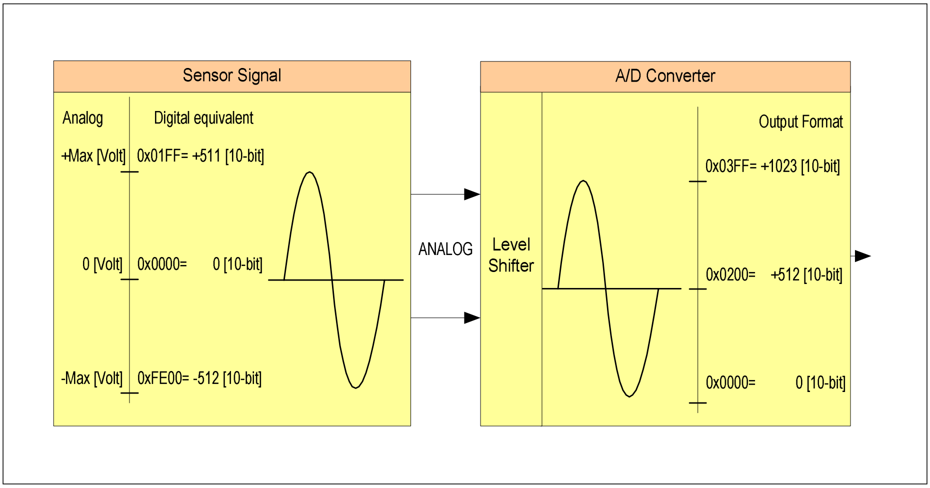


Figure 16 Format relation of the sensor signal and the A/D output

In most cases the direct output of the A/D converter will not be used but rather the output of one of the two comb filters.

Fig. 17 shows all three possible configurations to process the FADC converted signal. In all three cases the output has unsigned format but different dynamic ranges. These facts will influence:

- how the DMA transfer need to be configured
- which arithmetic should be used to make the format conversion 16 or 32-bit

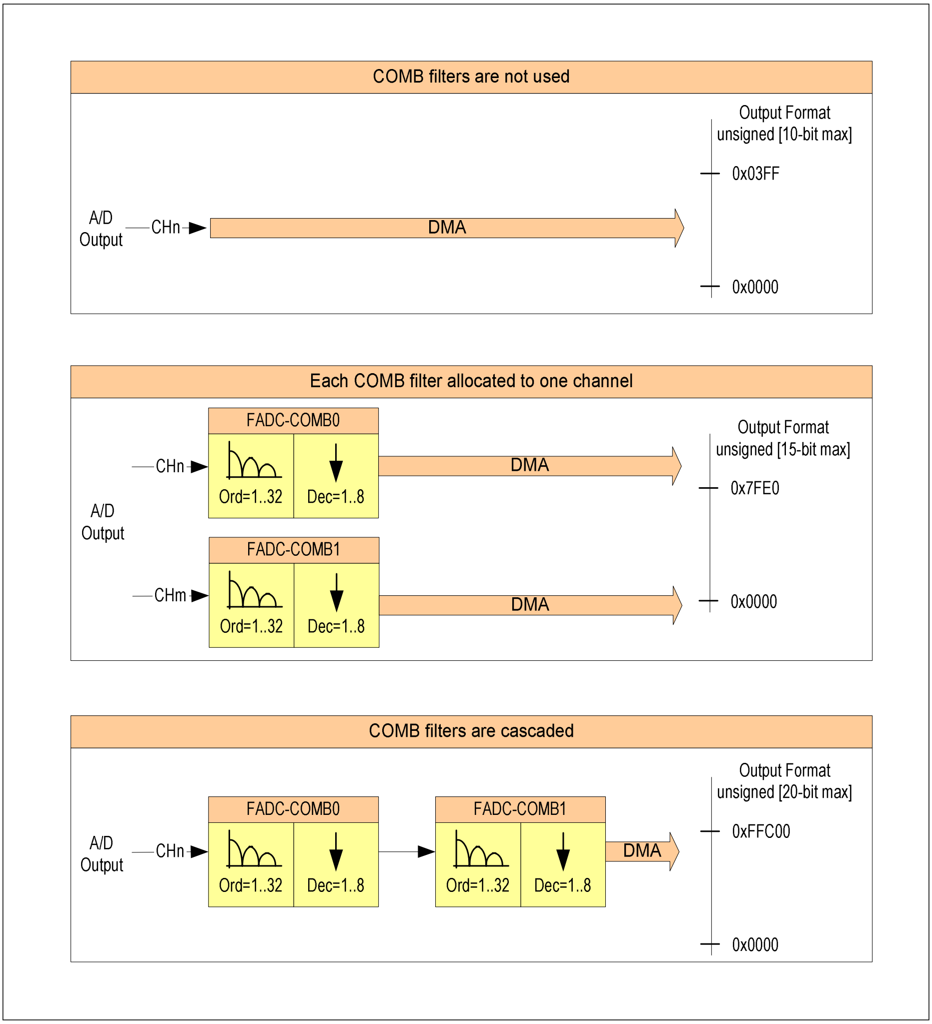


Figure 17 FADC Signal format and the dynamic range

All the cases are summarized in Table 1.

Table 1 Impact of FADC configuration on the DMA data-width and the Arithmetic for format conversion

	DMA		Arithmetic	
A/D	16-bit		16-bit	
A/D->COMB0	16-bit		16-bit	
A/D->COMB0->COMB1	16-bit	comb0.ord * comb1.ord < = 64	16-bit	comb0.ord * comb1.ord < = 32
	32-bit	comb0.ord * comb1.ord > 64	32-bit	comb0.ord * comb1.ord > 32

In the cases that the output is from A/D or that only one COMB filter is used, a 16-bit DMA and signed arithmetic conversion format can be used.

In the case where two cascaded COMB filters are used then the required DMA data-width and the required arithmetic is depended on the combined COMB filter order.

In Design Example-1 (which follow) COMB0.ord= 8, COMB1.ord= 6 then

COMB0.ord * COMB1.ord = 8*6 = 48 in this case DMA =16-bit, Arithmetic = 32-bit. It does not mean that all the arithmetic must be done with 32-bit, but the format conversion must be done with 32-bit signed arithmetic otherwise the positive values of 0x8000 and more will be interpreted as negative values. After format conversion and correct scaling, a 16-bit arithmetic can be used for further processing.

8.1.3 FADC Sampling Frequency setting (configuration)

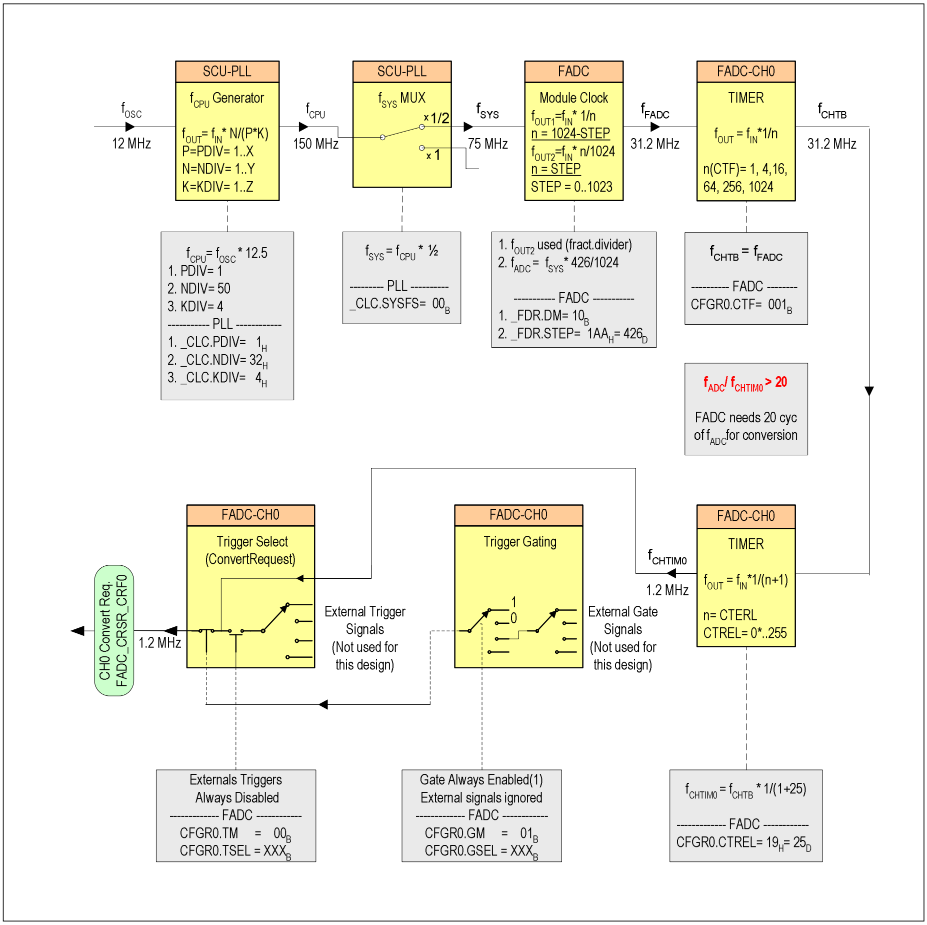


Figure 18 FADC Sampling Frequency setting

Figure 18 shows the TriCore modules involved to generate the “convert signal” of 1.2 MHz used by FADC A/D converter. In the above configuration the FADC timer is used. Each module includes additional information of the configuration parameters and the appropriate register settings.

8.1.4 Comb-Filter, the background

Decimation or data reduction filtering of a sampled data source input is often done in hardware. This is advantageous because of the reduced load of the subsequent signal processing.

Normal digital filtering uses 3 operations, these are delay, multiplication of the samples with certain predefined weights, and the summation of the intermediate results to gain final results.

However the implementation of a multiplier structure on silicon for this application is not cost efficient due to the required chip area. It is therefore too expensive in terms of system costs. For this reason solutions are needed which only use of the delay and the summation operations.

For this reason solutions are needed which only use the delay and the summation operation. There is one possible solution, which uses the summation of equally weighted samples. The corresponding formula (8.1) for the time domain is shown below.

$$y(n) = \mathop{\vphantom{\sum}}_{n=0}^{M-1} x(n-k) \tag{8.1}$$

where M is the order of the filter. All the filter coefficients are equal to one. Such a FIR filter with all its M coefficients equal to one is called a comb filter [0] of length M. The frequency response of the above-described filter plotted in Figure 19 has periodic notch frequencies that resemble the comb, hence the name comb filter.

The transfer function of the comb filter described by equation (8.1) is

$$H(z) = \sum_{n=0}^{M-1} z^{-k} = \frac{\left[1 - z^{-(M)}\right]}{\left(1 - z^{-1}\right)} = \frac{Y(z)}{X(z)} \tag{8.2}$$

The magnitude of frequency response of the transfer function in equation (8.2) can be expressed as follows (w is angular frequency).

$$H(w) = \frac{\sin \dfrac{wM}{2}}{\sin \dfrac{w}{2}} \tag{8.3}$$

It is known that

$$w = \frac{2\pi f}{f_s} \tag{8.4}$$

where f is a frequency, f_s is the sampling rate of filter input. Placing equation (8.4) in equation (8.3) we get

$$H(f) = \frac{\sin\left(\frac{\pi f M}{f_s}\right)}{\sin\left(\frac{\pi f}{f_s}\right)} \tag{8.5}$$

The equation (8.5) is used in Matlab AAF design framework for the calculation of comb filter frequency response.

From equation (8.5) can be seen that the frequency response has nulls at frequencies

$$\frac{\pi f M}{f_s} = k\pi \quad \text{where k = 1,2,3,} \tag{8.6}$$

$$\Rightarrow f = \frac{kf_s}{M} \tag{8.7}$$

The nulls of frequency response occur at integer multiples of $\frac{fs}{M}$ and are called notch frequencies.

The frequency response plot of moving average comb filter for M=8 is shown on Figure 19.

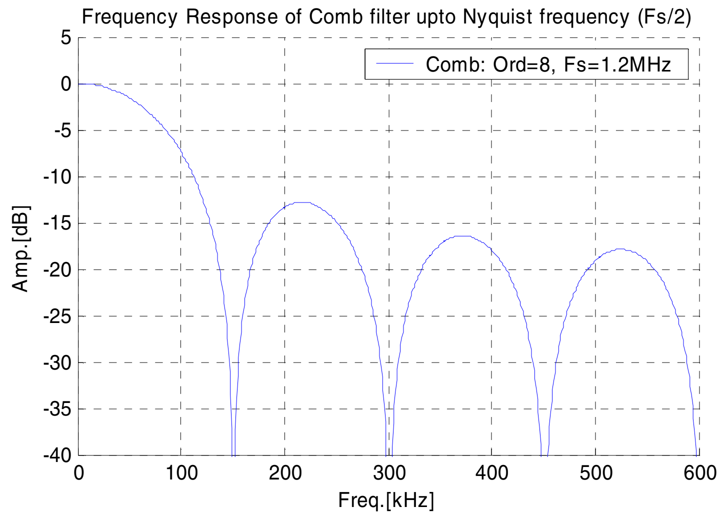


Figure 19 Comb filter frequency response

Even though each moving average comb filter has the advantage of easy hardware implementation, it has the following disadvantages for its use as anti aliasing filter.

- The periodic recurring maxima of the curve between 2 subsequent notches.
- The bad shape of the filter within the pass band region.
- The coupling between the first notch frequency and the decimation rate.

The disadvantages can be avoided mostly by cascading the two comb filters with appropriate filter characteristics.

It is obvious out of Figure 19 that if maximum of the second filter stage coincides with the minimum of the first filter stage then most of the above disadvantages can be suppressed.

First notch frequency of filter one = First maximum response point for filter two.
First notch frequency of filter one = 1.5 * First notch frequency of filter two.
I.e. fnotch1=1.5 fnotch2

Extending this relation with the different sampling frequencies for each filter

Fs1/M1 = Fs2/M2*3/2 ➔ Fs1/Fs2 = (M1/M2) * 3/2 ➔ D1 *M2 = M1* 3/2 where

Fs1 is input sampling rate of the first filter,
Fs2 is input sampling rate of the second filter or output-sampling rate of first filter,
M1 is the order of the first filter,
M2 is the order of the second filter and
D1 is the decimation factor (Fs1/Fs2) of thefirst filter.

Figure 20 demonstrates the advantage of cascading two comb filters. It depicts Hardware AAF filter characteristics of each comb filter working independently and when cascaded. (M1= 8, D1 =2, M2 =6, D2 =3) Input sampling rate =1.2 MHz

The improved characteristic is primarily in the stop band, where the notches of one filter smooth the peaks of the other.

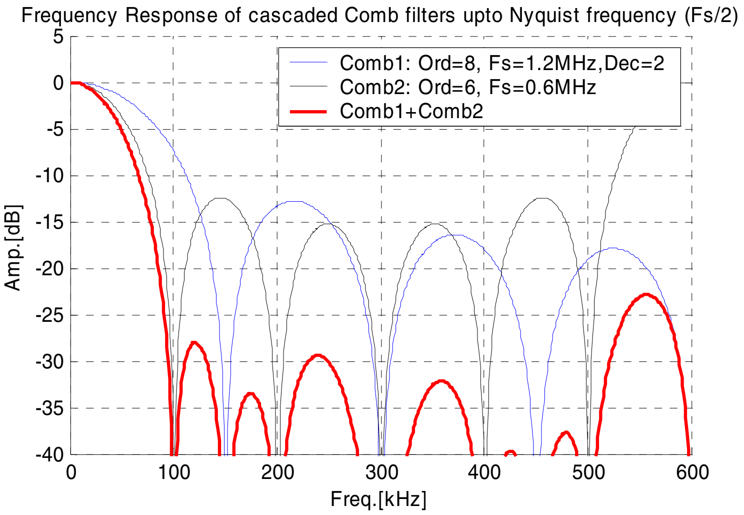
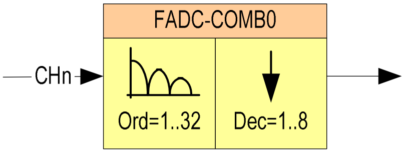


Figure 20 Cascading two Comb filters

8.1.5 Comb-Filter hardware implementation on TC1796

The TC1796 architecture supports two programmable anti-aliasing filters in hardware within the FADC kernel.



Ord = Comb filter Order
Dec = Decimation = Data Reduction = Sampling rate reduction

Figure 21 TC1796 comb filter functional representation

Figure 21 shows the functional representation of the FADC hardware filters that are implemented. The filters are implemented as low pass comb filters with programmable order of 1 to 32. Additionally the filter output allows sampling rate reduction (data reduction) from 1 to 8. As a results of specific hardware implementation there is some dependency between both configuration parameters Order (=Ord) and Decimation (Dec)

Internally in hardware (see Figure 22), each filter is implemented as a sum of intermediate results (Sx) which can be programmed from 1 to 8. The FR (Final Results) output is the moving average calculation on the Sx results.

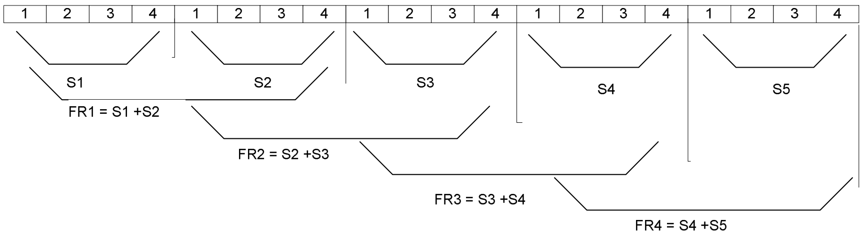


Figure 22 TC1796 Comp-Filter implementation

The implementation of data reduction filter with intermediate sum of 4 values and moving average length of 2 is shown on Figure 22. From this example we can recognize that the output sampling rate is reduced by 4 which is equal to the addition length of intermediate result Sx

The Final Result FR1 can be expressed as
FR1 = S1+S2 = x1+x2+x3+x4+x5+x6+x7+x8

where Xn are the input values to the comp filter.

The relationship between the Ord and Dec configuration parameters shown on Figure 21 and the internal hardware parameters can be expressed as

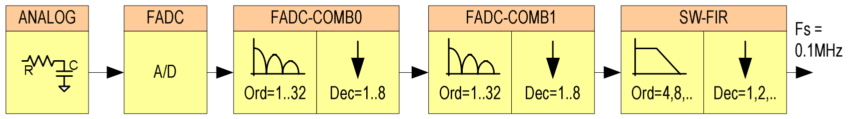
Dec = Length(Sx) = 4
Ord = Length(Sx)*Length(FRx) = 4*2 = 8

It can be identified that the Ord and Dec parameters are interrelated, so this has to be considered when defining these values.

8.1.6 Implementing an Anti-Aliasing Filter on TC1796 a Design Framework

This design framework provides configuration/tuning capability to achieve the desired overall AAF frequency response. It is implemented as a Matlab script with all the source code included in this document. Figure 23 shows the overall structure of the implemented AAF.

The basic concept (as already mentioned in the main text) is driven by a wish to use minimal external components, which in this design reduces the external analog filter to simply a first order RC structure



Input Parameters required for calculation of the overall frequency response				

Fc=60KHz
Fs=1.2MHz
		ord=8
		dec=2
		use=1
			ord=6
			dec=3
			use=1
				Fp =30KHz
				ord=11(12taps)
				dec=2
				use=1

Implicitly or fixed defined parameters				

ord=1 (fix)
use=1(fix)
	Fs=1.2MHz	Fs=1.2MHz	Fs=0.6MHz	Fs=0.2MHz

Implicitly defined, begin of stop-band				
Fst=0.6 MHz		Fst=0.3 MHz	Fst=0.1 MHz	Fst=0.05 MHz

Figure 23 Overall Anti-Aliasing-Filter (AAF) implementation structure (the input parameters are taken from design example 1)

The AAF structure as implemented on Figure 23 is the maximal variant which includes an analog RC-Filter, two COMB filters implemented in hardware and one FIR filter implemented as software function executed by TC1796 CPU. Setting the input parameter “use” of a particular module to zero will exclude it from the implementation. In the minimal version the RC-Filter alone can be used.

The design has few predefined configuration/methods which can not be changed through input parameters:

- Analog Filter (RC) has constant order ord=1 and is always used use=1
- SW-FIR software filter is of type FIR and uses Parks-McClellan optimal equal-ripple FIR filter design method for coefficient calculation

8.1.6.1 Input Parameters

It would be desirable to have the possibility to input the overall required AAF frequency response and have the program select the optimal configuration and parameters.

In the implemented program the user must enter directly the block parameters where their values are not usually directly related to the desired characteristic. It is recommended to use the included examples as a starting point for the new designs.

Table 2 Input Parameters

Parameter Name	Range	Remarks
Global		
AAF.Fp		Overall Pass-Band frequency
Analog Filter		
AAF.analog.fs	A/D Sampling freq [Hz]	only first order RC
AAF.analog.fc	3dB corner freq [Hz]	
HW Comb0 Filter		
AAF.comb0.use	1- used, 0- not used	
AAF.comb0.ord	order-1..32	consider the dependency of ord, dec
AAF.comb0.dec	Decimation – 1..8	consider the dependency of ord, dec
HW Comb1 Filter		
AAF.comb1.use	1- used, 0- not used	
AAF.comb1.ord	order-1..32	consider the dependency of ord, dec
AAF.comb1.dec	decimation – 1..8	consider the dependency of ord, dec
SW FIR Filter		
AAF.sw_fir.use	1- used, 0- not used	
AAF.sw_fir.ord	order- 4, 8, 12, …	for best SW implementation
AAF.sw_fir.dec	decimation – 1,2,3..	

8.1.6.2 Output Results

Based on the input configuration parameters the program generates two types of output

- Overall frequency response graph (see Figure 24)
- Filter coefficients of sw_fir (if used) stored in file

The generated frequency response graph should be used as a verification to see whether the calculated response fulfills the expectation. Eventually some tuning of parameters and the observation of the results will be required. The frequency response is drawn in two different resolutions; one to describe the overall response over all used processing frequency range, and the other to cover only the pass-band. The graphs include curves of partial results and the final response in red color.

8.1.6.3 Design Example 1

Table 3 Design Example-1: Input Parameters

Parameter Name	Parameter Value	Implicit Values
AAF.Fp	30 KHz	Overall Pass-Band frequency
AAF.analog.fs	1.2 MHz	A/D.fs = 1.2MHz
AAF.analog.fc	60 KHz	
AAF.comb0.use	1	fs_in= 1.2MHz; fs_out= 0.6MHz
AAF.comb0.ord	8	
AAF.comb0.dec	2	
AAF.comb1.use	1	fs_in= 0.6MHz; fs_out= 0.2MHz
AAF.comb1.ord	6	
AAF.comb1.dec	3	
AAF.sw_fir.use	1	fs_in= 0.2MHz; fs_out= 0.1MHz (100KHz)
AAF.sw_fir.ord	11 (12taps)	
AAF.sw_fir.dec	2	

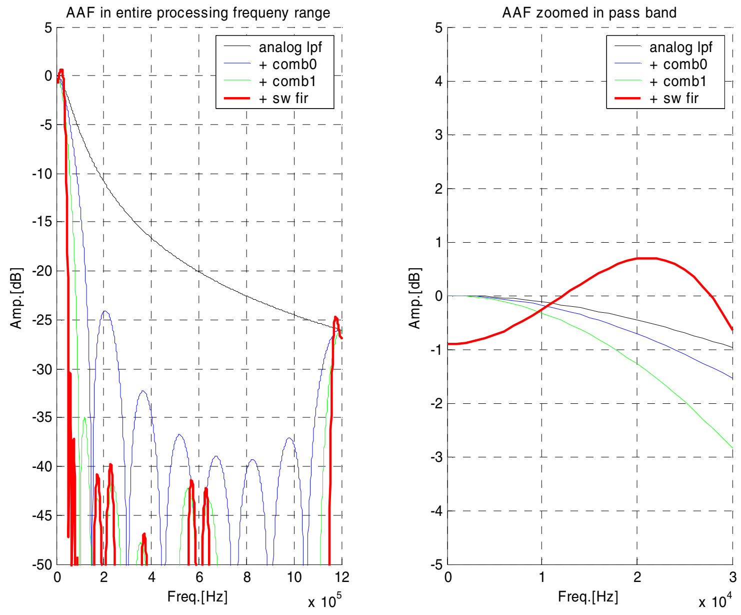


Figure 24 Design Example-1: Overall Frequency Response

Performance of this design:

- Pass-band ripple peak to peak of 1.8 dB
- Out of range frequency attenuation (aliasing frequencies) min 25dB by ~1.2 MHz

8.1.6.4 Design Example 2

Table 4 Design Example-2: Input Parameters (modules which are not in the table are excluded from design by setting its use=0)

Parameter Name	Parameter Value	Implicit Values
AAF.Fp	30 KHz	Overall Pass-Band freq. (used indirectly)
AAF.analog.fs	0.8 MHz	A/D.fs = 0.8 MHz
AAF.analog.fc	60KHz	
AAF.comb0.use	1	fs_in= 0.8 MHz; fs_out= 0.1MHz (100 KHz)
AAF.comb0.ord	8	
AAF.comb0.dec	8	

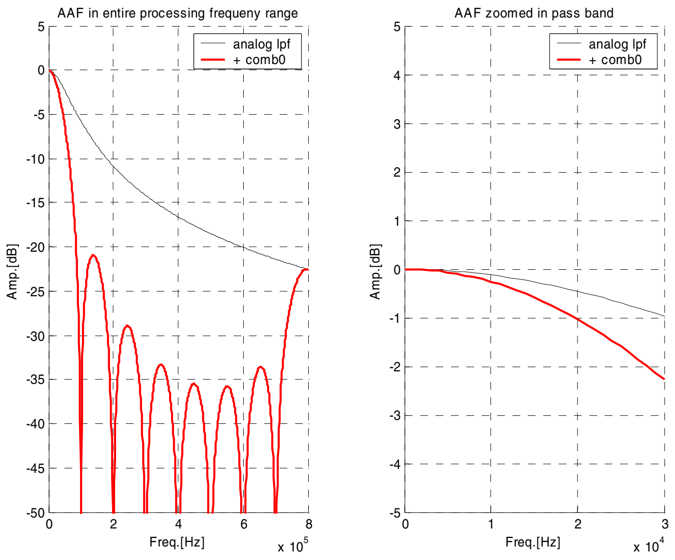


Figure 25 Design Example-2: Overall Frequency Response

Performance of this design:

- Pass-band ripple peak to peak of 2.3 dB
- Out of range frequency attenuation (aliasing frequencies) min 7dB by 50kHz. This is very poor rejection which will cause very strong aliasing of the frequency above 50kHz

8.1.6.5 Design Example 3

Table 5 Design Example-3

Parameter Name	Parameter Value	Implicit Values
AAF.Fp	30 KHz	Overall Pass-Band frequency
AAF.analog.fs	0.8 MHz	A/D.fs = 0.8 MHz
AAF.analog.fc	35 KHz	
AAF.comb0.use	1	fs_in= 0.8 MHz; fs_out= 0.2 MHz
AAF.comb0.ord	8	
AAF.comb0.dec	4	
AAF.sw_fir.use	1	fs_in= 0.2 MHz; fs_out= 100KHz
AAF.sw_fir.ord	15 (16taps)	
AAF.sw_fir.dec	2	

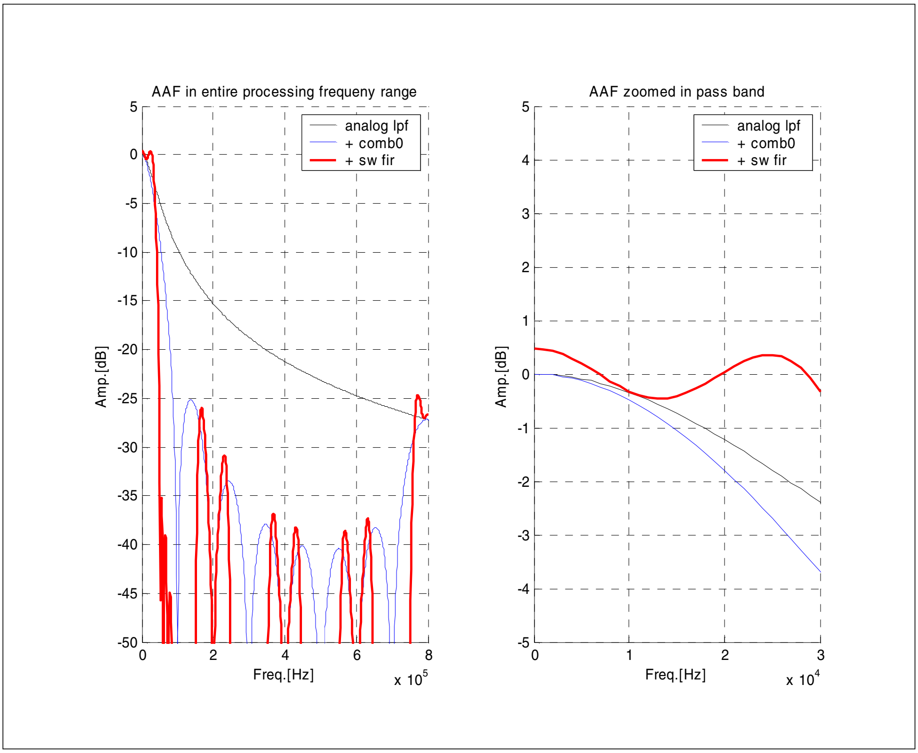


Figure 26 Design Example-3: Overall Frequency Response

Performance of this design:

- Pass-band ripple peak to peak of 1.0 dB (the sw fir has higher order then in example-1)
- Out of range frequency attenuation (aliasing frequencies) min 25dB by 780 KHz.
- The results in the stop-band are worse than in example-1 primarily in the range 100KHz to 300KHz

8.1.6.6 Matlab source code

Source code of Matlab function which calculate the overall frequency response and AAF.sw_fir coefficients as described in Design Examples-1 till 3 (see 8.1.6.3)

```
function   AntiAliasFilt(AAF)
% Antialiasing filter design
% -----------------------------------------------------------------------
% Design of the anti aliasing filter (AAF) from four cascaded Low-Pass
% filters
% analog_lpf: First order external analog low-pass filter
% comb0 : Internal, hardware implemented, programmable comb filter
% comb1 : Same as comb0, can be cascaded with comb0
% aaf_fir : Software implemented LPF
% The parameters for the design are passed by ConfigAAF() routine. This
% routine calculates and plots the anti-alias filter characteristics
% based on the parameters received.
% -----------------------------------------------------------------------
kHz = 1000;

% check for possible configuration of comb filter that are
% not realizable on hardware and generate the warning accordingly

if (AAF.comb0.dec >8)
  warning('Decimation factor of comb one filter  not realizable on
          hardware')
end

if (AAF.comb1.dec >8)
  warning('Decimation factor of Comb two filter not realizable on
          hardware')
end

ord = AAF.comb0.ord;
dec = AAF.comb0.dec;

if (~(ord == dec | ord == 2* dec| ord == 3*dec | ord == 4*dec) )
  warning('Order of comb one filter  not realizable on hardware')
end

ord = AAF.comb1.ord;
dec = AAF.comb1.dec;
```

```
if (~(ord == dec | ord == 2* dec| ord == 3*dec | ord == 4*dec))
  warning('Order of Comb two filter not realizable on hardware')
end

% frequency vector used to calculate and plot freq. response
f=[0*kHz:1*kHz:AAF.FSampIn];

% Setting of Plot parameters
figure;hold on;grid

% ---------------------------------------------------------------------
% analog_lpf: first order RC analog LPF
% ---------------------------------------------------------------------
% calculate frequency response [dB] on grid defined by f
analog_lpf.amp = -10*log10(1+(f/AAF.analog.fc).^2*(AAF.analog.ord));

% over-all frequency respose [dB] of external analog_lpf
overall.amp1 = analog_lpf.amp;

% ---------------------------------------------------------------------
% comb0
% ---------------------------------------------------------------------
if (AAF.comb0.use)
  % calculate frequency response on grid defined by f
  for i=1:AAF.FSampIn/kHz+1
    comb0.amp(i) = CombFreq((i-1)*kHz,AAF.FSampIn,AAF.comb0.ord);
  end

  % over-all frequency respose [dB] of two filters analog_lpf+comb0.amp
  overall.amp2 = analog_lpf.amp + comb0.amp;
end

% ---------------------------------------------------------------------
% comb1
% ---------------------------------------------------------------------
if (AAF.comb1.use)
  comb1.fs = AAF.FSampIn/ AAF.comb0.dec;  %Sampling Frequency after
             comb0

  % calculate frequency response [dB] on grid defined by f
  for i=1:AAF.FSampIn/kHz+1
    comb1.amp(i) = CombFreq((i-1)*kHz,comb1.fs,AAF.comb1.ord);
  end
  % over-all frequency response [dB] of three filters
  % analog_lpf+comb0.amp++comb1.amp
  overall.amp3 = analog_lpf.amp + comb0.amp + comb1.amp;
end

% ---------------------------------------------------------------------
% aaf_fir
% filter coefficients evaluated by REMEZ() function
% ---------------------------------------------------------------------
if (AAF.sw_fir.use)
  % The input sampling rate of the SW filter is decided by whether the
  % comb filters are in use or not.

  if (~AAF.comb0.use & ~AAF.comb1.use )
    aaf_fir.fs = AAF.FSampIn;
  elseif(AAF.comb0.use & ~AAF.comb1.use )
    aaf_fir.fs = AAF.FSampIn/AAF.comb0.dec;
  elseif(AAF.comb1.use)
    aaf_fir.fs = comb1.fs/AAF.comb1.dec;
  end

  % Nyquist frequency at the input of SW FIR filter
  aaf_fir.Nyqf = aaf_fir.fs/(2) ;

  % The stop band frequency of the fir filter is chosen as the half the
  % output sampling rate.
  AAF.Fst =aaf_fir.Nyqf/AAF.sw_fir.dec;

  % The frequency response of the FIR filter is expressed in terms of
  % fdef and adef.
  % The frequency points are defined in steps of one kHz till pass band
  % frequency. The number of fdef points should be even as they should
  % always be specified in pairs.
  % So if length of fdef is not even for particular pass band frequency,
  % the value just before the pass band frequency is dropped from the
  % fdef definition

  fdef =[0:1*kHz/aaf_fir.Nyqf:AAF.Fp/aaf_fir.Nyqf];
  adef =ones(1, length(fdef));

  index = length(fdef);
  if (bitand(index,1)) % checking whether number is even or odd
    % if odd, reduce the index by one and remove the fdef value just
    % before pass band frequency
```

```
 index = length(fdef)-1;

 fdef(index)= AAF.Fp/aaf_fir.Nyqf;
 adef(index) =1
end

% Stop band frequency and the desired frequency response
fdef(index+1)= AAF.Fst/aaf_fir.Nyqf;
adef(index+1) =0;

% Nyquist frequency and the desired frequency response
fdef(index+2)= 1;
adef(index+2) =0;

adef2 = adef;
% inverse calculation
% adef2 includes the deviation of till now calculated filter
% so the aaf_fir should compansate this
% desired freqency response combined from "adef" with the inversed
% frequency response of analog_lpf+comb0+comb1
% This case provides a better frquency response in pass band

if (AAF.comb1.use)
  for j=1:length(fdef)
    jj=round(fdef(j)*(aaf_fir.fs/2)/(1*kHz))+1;
    adef1(j)=...
    10^(-(comb0.amp(jj)+comb1.amp(jj)+analog_lpf.amp(jj))/20);
  end
  adef2 = adef.*adef1;

elseif(AAF.comb0.use)
  % desired freqency response combined from "adef" with the inversed
  % frequency response of analog_lpf+comb0
  % This case provides a better frequency response in pass band
  for j=1:length(fdef)
    jj=round(fdef(j)*(aaf_fir.fs/2)/(1*kHz))+1;
    adef1(j)=...
    10^(-(comb0.amp(jj)+analog_lpf.amp(jj))/20);
  end
  adef2 = adef.*adef1;

end

% Find Filter coefficients for desired filter response
[B,ERR,RES]=REMEZ(AAF.sw_fir.ord,fdef,adef2);
```

```
  % calculate frequency response [dB] on grid defined by f
  H=FREQZ(B,1,f,aaf_fir.fs);
  aaf_fir.amp = 20*log10(abs(H));

  % Save coefficients
  SaveCoefToFile(B,AAF.sw_fir.filename);

  % ---------------------------------------------------------------------
  % over-all frequency response [dB] of the 4 cascaded filters
  % analog_lpf+comb0.amp+comb1.amp+aaf_fir.amp. depending on whether
  %certain filters are used or not, the overall response varies.
  % ---------------------------------------------------------------------

  if (AAF.comb1.use)
    overall.amp4 = analog_lpf.amp+comb0.amp+comb1.amp+ aaf_fir.amp;
  elseif (AAF.comb0.use & ~AAF.comb1.use)
    overall.amp4 = analog_lpf.amp+comb0.amp+ aaf_fir.amp;
  elseif (~AAF.comb0.use &  ~AAF.comb1.use)
    overall.amp4 = analog_lpf.amp+ aaf_fir.amp;
  end

end


% The code generates the output for the configuration shown below
% -----------------------------------------------------------------------
%
% -------------   ----------   ----------   ----------   ----------
% | analog_lpf|-->|   A/D    |-->|comb0    |-->|comb1    |-->| aaf_fir|->
% |    LPF    |   |          |   |    LPF  |   | LPF     |   |   LPF  |
% -------------   ----------   ----------   ----------   ----------
%
% -----------------------------------------------------------------------

if (AAF.comb0.use & AAF.comb1.use & AAF.sw_fir.use)

  switch(AAF.graph.type)
    case 'Overall'
      subplot(121); hold on;grid;
      title(['AAF in entire processing frequeny range'] )

      plot(f,overall.amp1,'k','LineWidth',1);
      plot(f,overall.amp2,'b','LineWidth',1);
      plot(f,overall.amp3,'g','LineWidth',1);
      plot(f,overall.amp4,'r','LineWidth',2);
      legend('analog lpf','+ comb0','+ comb1','+ sw fir');

      xlabel('Freq.[Hz]')
      ylabel('Amp.[dB]')

      xlim ([0 AAF.FSampIn])
      ylim([-50 5])

      subplot(122); hold on;grid;
      title(['AAF zoomed in pass band'] )

      plot(f(1:AAF.Fp/kHz+1),overall.amp1(1:AAF.Fp/kHz+1),'k',
           'LineWidth',1);
      plot(f(1:AAF.Fp/kHz+1),overall.amp2(1:AAF.Fp/kHz+1),'b',
           'LineWidth',1);
      plot(f(1:AAF.Fp/kHz+1),overall.amp3(1:AAF.Fp/kHz+1),'g',
           'LineWidth',1);
      plot(f(1:AAF.Fp/kHz+1),overall.amp4(1:AAF.Fp/kHz+1),'r',
           'LineWidth',2);

      legend('analog lpf','+ comb0','+ comb1','+ sw fir');

      xlim ([0 AAF.Fp])
      ylim([-5 5])

    case 'Individual'
      subplot(121); hold on;grid;
      title(['AAF in entire processing frequeny range'] )

      plot(f,analog_lpf.amp,'k');
      plot(f,comb0.amp,'b');
      plot(f,comb1.amp,'g');
      plot(f,aaf_fir.amp,'r');

      legend('analog lpf','comb0','comb1','sw fir');

      xlim ([0 AAF.FSampIn])
      ylim([-50 5])

      subplot(122); hold on;grid;
      title(['AAF zoomed in pass band'] )

      plot(f(1:AAF.Fp/kHz+1),analog_lpf.amp(1:AAF.Fp/kHz+1),'k');
      plot(f(1:AAF.Fp/kHz+1),comb0.amp(1:AAF.Fp/kHz+1),'b');
      plot(f(1:AAF.Fp/kHz+1),comb1.amp(1:AAF.Fp/kHz+1),'g');
      plot(f(1:AAF.Fp/kHz+1),aaf_fir.amp(1:AAF.Fp/kHz+1),'r');
```

```
      xlim ([0 AAF.Fp])
      ylim([-5 5])

  end;
end


% The code generates the output for the configuration shown below
% -----------------------------------------------------------------------
%
%    -------------   ----------   ----------   ----------
% -> | analog_lpf|-->|  A/D    |-->|comb0    |-->|comb1    |-->
%    |    LPF    |   |         |   |   LPF   |   |  LPF    |
%    -------------   ----------   ----------   ----------
%
% -----------------------------------------------------------------------


if ( AAF.comb0.use &  AAF.comb1.use &  ~AAF.sw_fir.use)

  switch(AAF.graph.type)
    case 'Overall'
      subplot(121); hold on;grid;
      title(['AAF in entire processing frequeny range'] )

      plot(f,overall.amp1,'k','LineWidth',1);
      plot(f,overall.amp2,'b','LineWidth',1);
      plot(f,overall.amp3,'r','LineWidth',2);

      legend('analog lpf','+ comb0','+ comb1');

      xlabel('Freq.[Hz]')
      ylabel('Amp.[dB]')

      xlim ([0 AAF.FSampIn])
      ylim([-50 5])

      subplot(122); hold on;grid;
      title(['AAF zoomed in pass band'] )

      plot(f(1:AAF.Fp/kHz+1),overall.amp1(1:AAF.Fp/kHz+1),'k',
           'LineWidth',1);
      plot(f(1:AAF.Fp/kHz+1),overall.amp2(1:AAF.Fp/kHz+1),'b',
           'LineWidth',1);
      plot(f(1:AAF.Fp/kHz+1),overall.amp3(1:AAF.Fp/kHz+1),'r',
           'LineWidth',2);
      legend('analog lpf','+ comb0','+ comb1');

      xlim ([0 AAF.Fp])
      ylim([-5 5])

    case 'Individual'
      subplot(121); hold on;grid;
      title(['AAF in entire processing frequeny range'] )

      plot(f,analog_lpf.amp,'k');
      plot(f,comb0.amp,'b');
      plot(f,comb1.amp,'r');

      legend('analog lpf','comb0','comb1');

      xlim ([0 AAF.FSampIn])
      ylim([-50 5])

      subplot(122); hold on;grid;
      title(['AAF zoomed in pass band'] )

      plot(f(1:AAF.Fp/kHz+1),analog_lpf.amp(1:AAF.Fp/kHz+1),'k',
           'LineWidth',1);
      plot(f(1:AAF.Fp/kHz+1),comb0.amp(1:AAF.Fp/kHz+1),'b',
           'LineWidth',1);
      plot(f(1:AAF.Fp/kHz+1),comb1.amp(1:AAF.Fp/kHz+1),'g',
           'LineWidth',1);

      legend('analog lpf','comb0','comb1');

      xlim ([0 AAF.Fp])
      ylim([-5 5])

  end;
end


% The code generates the output for the configuration shown below
% -----------------------------------------------------------------------
%
%    -------------   ----------   ----------
% -> | analog_lpf|-->|  A/D    |-->|comb0    |-->
%    |    LPF    |   |         |   |   LPF   |
%    -------------   ----------   ----------
%
% -----------------------------------------------------------------------
```

```
if (AAF.comb0.use & ~AAF.comb1.use & ~AAF.sw_fir.use )

  switch(AAF.graph.type)
    case 'Overall'
      subplot(121); hold on;grid;
      title(['AAF in entire processing frequeny range'] )

      plot(f,overall.amp1,'k','LineWidth',1);
      plot(f,overall.amp2,'r','LineWidth',2);

      legend('analog lpf','+ comb0');

      xlabel('Freq.[Hz]')
      ylabel('Amp.[dB]')

      xlim ([0 AAF.FSampIn])
      ylim([-50 5])

      subplot(122); hold on;grid;
      title(['AAF zoomed in pass band'] )

      plot(f(1:AAF.Fp/kHz+1),overall.amp1(1:AAF.Fp/kHz+1),'k',
           'LineWidth',1);
      plot(f(1:AAF.Fp/kHz+1),overall.amp2(1:AAF.Fp/kHz+1),'r',
           'LineWidth',2);

      legend('analog lpf','+ comb0');

      xlim ([0 AAF.Fp])
      ylim([-5 5])

    case 'Individual'
      subplot(121); hold on;grid;
      title(['AAF in entire processing frequeny range'] )

      plot(f,analog_lpf.amp,'k');
      plot(f,comb0.amp,'r');

      legend('analog lpf','comb0');

      xlim ([0 AAF.FSampIn])
      ylim([-50 5])

      subplot(122); hold on;grid;
      title(['AAF zoomed in pass band'] )
      plot(f(1:AAF.Fp/kHz+1),analog_lpf.amp(1:AAF.Fp/kHz+1),'k',
           'LineWidth',1);
      plot(f(1:AAF.Fp/kHz+1),comb0.amp(1:AAF.Fp/kHz+1),'b',
           'LineWidth',1);

      legend('analog lpf','comb0');

      xlim ([0 AAF.Fp])
      ylim([-5 5])

  end;
end


% The code generates the output for the configuration shown below
% -----------------------------------------------------------------------
%
%    -------------   ----------   ----------   ----------
% -> | analog_lpf|-->|   A/D    |-->|comb0    |-->| aaf_fir|->
%    |     LPF    |   |         |   |    LPF  |   |   LPF  |
%    -------------   ----------   ----------   ----------
%
% -----------------------------------------------------------------------


if (AAF.comb0.use & ~AAF.comb1.use & AAF.sw_fir.use)
  switch(AAF.graph.type)
    case 'Overall'
      subplot(121); hold on;grid;
      title(['AAF in entire processing frequeny range'] )

      plot(f,overall.amp1,'k','LineWidth',1);
      plot(f,overall.amp2,'b','LineWidth',1);
      plot(f,overall.amp4,'r','LineWidth',2);

      legend('analog lpf','+ comb0','+ sw fir');

      xlabel('Freq.[Hz]')
      ylabel('Amp.[dB]')

      xlim ([0 AAF.FSampIn])
      ylim([-50 5])

      subplot(122); hold on;grid;
      title(['AAF zoomed in pass band'] )
      plot(f(1:AAF.Fp/kHz+1),overall.amp1(1:AAF.Fp/kHz+1),'k',
           'LineWidth',1);
      plot(f(1:AAF.Fp/kHz+1),overall.amp2(1:AAF.Fp/kHz+1),'b',
           'LineWidth',1);
      plot(f(1:AAF.Fp/kHz+1),overall.amp4(1:AAF.Fp/kHz+1),'r',
           'LineWidth',2);

      legend('analog lpf','+ comb0','+ sw fir');

      xlim ([0 AAF.Fp])
      ylim([-5 5])

    case 'Individual'
      subplot(121); hold on;grid;
      title(['AAF in entire processing frequeny range'] )

      plot(f,analog_lpf.amp,'k');
      plot(f,comb0.amp,'b');
      plot(f,aaf_fir.amp,'r');

      legend('analog lpf','comb0','sw fir');

      xlim ([0 AAF.FSampIn])
      ylim([-50 5])

      subplot(122); hold on;grid;
      title(['AAF zoomed in pass band'] )

      plot(f(1:AAF.Fp/kHz+1),analog_lpf.amp(1:AAF.Fp/kHz+1),'k',
           'LineWidth',1);
      plot(f(1:AAF.Fp/kHz+1),comb0.amp(1:AAF.Fp/kHz+1),'b',
           'LineWidth',1);
      plot(f(1:AAF.Fp/kHz+1),comb1.amp(1:AAF.Fp/kHz+1),'r',
           'LineWidth',2);

      legend('analog lpf','comb0','sw fir');

      xlim ([0 AAF.Fp])
      ylim([-5 5])

  end;
end
```

```
% The code generates the output for the configuration shown below
% ----------------------------------------------------------------------
%
%    -------------   ----------    ----------
% -> | analog_lpf|-->|  A/D    |-->| aaf_fir|-->
%    |    LPF     |   |         |   |  LPF   |
%    -------------   ----------    ----------
%
% ----------------------------------------------------------------------


if (~AAF.comb0.use & ~AAF.comb1.use & AAF.sw_fir.use)
  switch(AAF.graph.type)
    case 'Overall'
      subplot(121); hold on;grid;
      title(['AAF in entire processing frequeny range'] )

      plot(f,overall.amp1,'k','LineWidth',1);
      plot(f,overall.amp4,'r','LineWidth',2);

      legend('analog lpf','+ sw fir');

      xlabel('Freq.[Hz]')
      ylabel('Amp.[dB]')

      xlim ([0 AAF.FSampIn])
      ylim([-50 5])

      subplot(122); hold on;grid;
      title(['AAF zoomed in pass band'] )

      plot(f(1:AAF.Fp/kHz+1),overall.amp1(1:AAF.Fp/kHz+1),'k',
           'LineWidth',1);
      plot(f(1:AAF.Fp/kHz+1),overall.amp4(1:AAF.Fp/kHz+1),'r',
           'LineWidth',2);

      legend('analog lpf','+ sw fir');

      xlim ([0 AAF.Fp])
      ylim([-5 5])

    case 'Individual'
      subplot(121); hold on;grid;
      title(['AAF in entire processing frequeny range'] )
      plot(f,analog_lpf.amp,'k');
      plot(f,aaf_fir.amp,'r');

      legend('analog lpf','sw fir');

      xlim ([0 AAF.FSampIn])
      ylim([-50 5])

      subplot(122); hold on;grid;
      title(['AAF zoomed in pass band'] )

      plot(f(1:AAF.Fp/kHz+1),analog_lpf.amp(1:AAF.Fp/kHz+1),'k',
           'LineWidth',1);
      plot(f(1:AAF.Fp/kHz+1),aaf_fir.amp(1:AAF.Fp/kHz+1),'r',
           'LineWidth',2);

     legend('analog lpf','sw fir');

     xlim ([0 AAF.Fp])
     ylim([-5 5])

  end;
end

xlabel('Freq.[Hz]')
ylabel('Amp.[dB]')

%--------------------------------------------------------------------
function y = CombFreq(f,fs,M)
% Implementation of Comb filter
% please refer to the comb filter section in the Appendix of application
% note for details regarding the comb filter equation.
% fs is the sampling rate
% M is the order of the comb filter

%Comb Filter Frequency Response
if f==0
    x=1;
else
    x=sin((pi*f*M)/fs)/(M*sin((pi*f)/fs));
end

y=20*log10(abs(x));

%--------------------------------------------------------------------
function SaveCoefToFile(aCoef, filename)
% Save the aaf_fir coefficients to file
```

```
nH = length(aCoef);

aCoef_fix16 = FixSat16Bits(aCoef)

OutFileName = filename;
%Open the Out file
fid = fopen(OutFileName,'w');

%Save Input nX Vector
fprintf(fid,'//nH = %6.f \n',nH);

%Save the coefficients in sfract format
fprintf(fid,'//Coefficients in 1Q15 sfract format\n');
for k = 1:nH
    fprintf(fid,'%10.6f, \n',aCoef(k));
end;

%Save the coefficients in short format
fprintf(fid,'\n\n//Coefficients in 1Q15 short format\n');
for k = 1:nH
    fprintf(fid,'%6.f, \n',aCoef_fix16(k));
end;

status = fclose(fid);
```

8.1.6.7 Matlab script used as configuration input to the AntiAliasFilt() function

This script is used in Design Example-1, the parameters are described in Table 3

```
% Anti Alias filter design
% _---------------------------------------------------------------------
% Design of the anti aliasing filter (AAF) from four cascaded Low-Pass
% filters
% analog_lpf: First order external analog low-pass filter
% comb0 : Internal, hardware implemented, programmable comb filter
% comb1 : Same as comb1, can be cascaded with comb0
% aaf_fir : Software implemented LPF
% ----------------------------------------------------------------------

% The purpose of the program is to separate the implementation details
% from the parameter definition. All the parameters required in AAF
% design are defined here. The program in turn calls the Filter design
% program which plots the over all characteristics of AAF. The user can
% vary these parameters and verify from the plot whether his
% requirements are met.The user can fine tune the parameters till the
% requirements are satisfied. A default configuration is provided for
% anti-aliasing which can be fine tuned as per the requirements.

% A specific design to fulfill following over-all characteristic is done
% with requirements shown below
% Fpass. = 0-30 KHz
% Fstop  = 50 KHz
% Apass  = 2dB
% Astop  = 20dB
% The configuration and parameters for the design are shown below.
% ----------------------------------------------------------------------
%
% -------------   ----------   ----------   ----------   ----------
% | analog_lpf|-->|  A/D     |-->|comb0    |-->|comb1    |-->| aaf_fir|->
% |    LPF    |   |          |   |   LPF   |   |  LPF    |   |   LPF  |
% -------------   ----------   ----------   ----------   ----------
% Fs=1.2MHz       Fs=1.2MHz    Fs=0.6MHz    Fs=0.2MHz
%                              ord=8        ord=6        ord=11(12taps)
%                              dec=2        dec=3        dec=2
% Fst=0.6MHz      Fst=0.3MHz   Fst=0.1MHz   Fst=0.05MHz
%
%
% Fp  - Pass frequency is 30kHz for all filters
% Fs  - sampling frequency used for processing (fs_in)
% dec - decimation factor dec = fs_in/fs_out
% Fst - Start of stop-band derived based on Nyquist freq
%       (Fst=Fs/(2*dec))
%
%       Astop = 20dB (20dB attenuation of signal above Nyquist freq)
% ----------------------------------------------------------------------

% Initialization of general variables
clear;hold off;close all
kHz=1000;

% Over all desired filter characteristics
% Pass band = AAF.Fp =30 * kHz;
AAF.Fp =30 * kHz;

% Stop band is decided based on overall decimation
% Input sampling rate. Sampling rate of ADC converter
% The output sampling rate is Input sampling rate divide by overall
decimation factors.
AAF.FSampIn =1200 * kHz;

% Analog filter Parameters
AAF.analog.fc   =  60 * kHz;
AAF.analog.ord  = 1;  % always fix it to 1

% First Comb filter parameters
AAF.comb0.use =1;
AAF.comb0.ord =8;
AAF.comb0.dec =2;
```

```
% Second Comb filter parameters
AAF.comb1.use =1;
AAF.comb1.ord =6;
AAF.comb1.dec =3;

% SW AAF parameters
AAF.sw_fir.use  =1;
AAF.sw_fir.ord  =11;
AAF.sw_fir.dec = 2;

% File name to store coefficients
AAF.sw_fir.filename = 'aaf_fir.txt'

% Plotting of the anti Alias filter response.
% if AAF.graph.type ='overall' the combined filter response at each
% filter stage i.e analog, analog+comb0, anlog+comb0+comb1 and
% analog+comb0+comb1+sw_fir is plotted

% if AAF.graph.type ='Individual' the individual filter response
% i.e analog, comb0, comb1 and sw_fir is plotted.
AAF.graph.type = 'Overall'  %'Individual', 'Overall'

% Routine which calculates and plots the filter response
AntiAliasFilt(AAF);
```

8.1.7 Implementing SW FIR +Decimator

The following TriCore source code is implemented in assembly using Tasking compiler syntax. It is the implementation of the AAF FIR filter (called “AAF.sw_fir” in the input parameter tables). This implementation has predefined size of 12-taps stored in FIRTAPS and Decimation of 2 stored in DECFAC. It is a special implementation of FIR filter which does not have its own delay-line and works directly on the input data buffer.

8.1.7.1 TC1796 Source Code

```
;***********************************************************************
; void AafFirUn_12T(
;           cptrDataS   *X,
;           cptrDataS   *H,
;           DataS       *R,
;           int         nX_)
;
; INPUTS:
;       FIRTAPS Define the number of filter coefficients (set during
;               compilation)
;       DECFAC  Decimation
;       X       Circ-Ptr to Input-Buffer
;       H       Circ-Ptr of Coeff-Buffer of size nH, nH = 12
;       R       Ptr to Output-Buffer = Filter output values in 1Q15
;       nX      Number of Input values to process ->
;               If nX=4 FIRTAPS=12 it means that filter will use
;               nX+FIRTAPS-1
;               input values 4+12-1 = 15!
;
;
; OUTPUTS:
;       R(nX)   Output-Buffer = Filter output values in 1Q15
;       *X      Keep the updated pointer to the next input values
;               Useful in case the processing is split to few calls
;
; RETURN:
;       -
;
```

```
; DESCRIPTION:
;       FIR filter transversal structure (direct form),
;       Fixed size of 12-taps
;       Fixed decimation of 2
;       Input block processing,
;       16-bit fractional input, coefficients and output data format,
;       Optimal implementation requires filter order to be multiple of 4
;       ----------------------------------------------------------------
;       Implementation of FIR filter which working directly on the
;       input-buffer.
;       There is no separate Delay-Line!
;       Defining input-data as circular buffer allow flexible concurrent
;       management
;       of dataacquisition and processing.
;       Usually the size of the buffer should be bigger then the filter
;       size. The circular buffer can be also used as linear buffer by
;       selecting adaquate size. Before each call to this function the
;       input-;buffer pointer should point to the start of the buffer.
;       Other possibility it to modify the implementation so that linear
;       input-buffer will be directly supported
;       The best implementation is achieved with constant filter size.
;       Because small filter size, full unrolling of the kernel has been
;       implemented.
;       The decimation of two (down sampling) will be implemented by
;       skipping one input sample at the start of input data blocks
;
; ALGORITHM:
;       Case the filter has 12 taps
;       R(0)  = X(0)*H(0)+X(1)*H(1)+X(2)*H(1)+ ..+ X(11)*H(11)
;       R(1)  = X(2)*H(0)+X(3)*H(1)+X(4)*H(1)+ ..+ X(13)*H(11) start=X(2)
;       R(2)  = X(4)*H(0)+X(5)*H(1)+X(6)*H(1)+ ..+ X(15)*H(11) start=X(4)
;       ....
;
;       where,
;       R(n)                  : Output sample of the filter at index n
;       X(n)                  : Input sample of the filter at index n
;       H(0),H(1),H(2),..     : Filter coefficients
;
; TECHNIQUES:
;       1)  Loop unrolling, 4 taps/loop.
;       2)  Use of packed data load/store
;       3)  Coefficient buffer implemented as circular buffer
;       4)  Use of dual MAC instructions.
;       5)  Intermediate results stored in 64 bit register
;           (16 guard bits).
;       6)  Instruction ordering for zero overhead Load/Store
;
```

```
; ASSUMPTIONS:
;       1)  Filter order should be multiple of 4 and minimum filter
;           order is 8
;       2)  Inputs, outputs, coefficients should be in 1Q15 format.
;
; MEMORY NOTE:
;
; _______________________________________________________________________
;
;  Pointer      Pointer     Variable                  Alignment
;               Type                           IntMem or       ExtMem
;                                              ExtMem+Cache    NoCache
;
; _______________________________________________________________________
;
;   H           Circ        H(0), H(1), H(2),  8-bytes         8-bytes
;                           ..., H(FIRTAPS-1)
;
; _______________________________________________________________________
;
;   X           Circ        X(n), X(n+1), X(n+2),.8-bytes      8-bytes
;
; _______________________________________________________________________
;
;   R           Lin         R(n), R(n+1), R(n+2),.2-bytes      2-bytes
;
; _______________________________________________________________________
;
;
; REGISTER USAGE:
;       a2, a3, a4, a5, a6, a7, a12, a13, a14, a15.
;       d4, d5, d6, d8, d9, d10, d11, d12, d13, d14
;
; CYCLE COUNTS:
;       12 cyc/sample  use nX=200
; CODE SIZE:
;       92 bytes
;
;***********************************************************************
```

```
;------------------- Register Allocation ------------------------------
.define     FIRTAPS      "12"         ;
.define     DECFAC       "2"          ;Decimation (Fs_out/Fs_in)

.define     capX         "adArg1"     ;a4 Ptr to Circ-Ptr of Input-Buffer
.define     capH         "adArg2"     ;a5 Ptr to Circ-Ptr of Coef
.define     aR           "adArg3"     ;a6 Ptr to Output-Buffer

.define     nX           "wArg1"      ;d4 Number of samples to process

.define     caH          "a2/a3"      ;Circ-Ptr of Coeff-Buffer
.define     caeH         "a2"         ;Even-Reg of Circ-Ptr
.define     caoH         "a3"         ;Odd-Reg of Circ-Ptr

.define     caX          "a12/a13"    ;Circ-Ptr of Input-Buffer
.define     caeX         "a12"        ;Even-Reg of Circ-Ptr
.define     caoX         "a13"        ;Odd-Reg of Circ-Ptr

.define     aSampleCnt   "a14"        ;Loop-Cnt-Reg (loop ext)
.define     aTapCnt      "a15"        ;Loop-Cnt-Reg (loop int)

.define     ssssX        "e10"        ;Filter internal state
.define     sseX         "d10"        ;Even-Reg
.define     ssoX         "d11"        ;Odd-Reg

.define     ssssH        "e8"         ;Filter Coeff.
.define     sseH         "d8"         ;Even-Reg
.define     ssoH         "d9"         ;Odd-Reg

.define     llAcc        "d12/d13"    ;Filter result
.define     leAcc        "d12"        ;Even-Reg
.define     loAcc        "d13"        ;Odd-Reg

.define     dTmp         "d14"        ;Generic temporary Data-Reg
.define     dTapLoops    "d15"        ;

;=================== Executable Code ===================================
.align  32

AafFirUn_12T:

  FEnter      0
  ;Align
  nop                               ;TC-1796
  nop                               ;TC-1796
```

```
;-------------------------------------------------------------------------
;Init the loop count for input of nX samples
;Cnt = ((int)(nX)/DECFAC) -1    (-1 for loop adjust)

  sh          nX,#-1              ;Correct only for DECFAC=2!!!
  add         nX,nX,#-1           ;-1 for loop adjust

  mov.a       aSampleCnt,nX

  ld.da       caX,[capX]          ;Load the CircPtr of Input-Buffer to
                                  ;Reg-Pair caX
  ld.da       caH,[capH]          ;Load the Circ-Ptr of Coef-Buffer to
                                  ;Reg-Pair caX

;------------------- Loop for Input Buffer Starts ----------------------
;X(n) "n" indexes in remarks are correct for first sample

FirB4_InDataLSU1:

    mov         leAcc,#0            ;Clear the Even-Reg of llAcc ||
    ld.d        ssssH,[caH+c]4*W16  ;ssoH <- H(0),H(1), H(2),H(3)

    mov         loAcc,#0            ;Clear the Odd-Reg of llAcc ||
    ld.w        ssoX,[caX+c]2*W16   ;ssoX <- X(0),X(1)

;------------------- Kernel --------------------------------------------

    ; 4 taps
    maddm.h     llAcc,llAcc,ssoX,sseH ul,#1
                                    ;llAcc += X(0)*H(0) + X(1)*H(1) ||
    ld.d        ssssX,[caX+c]4*W16  ;ssssX <- X(2),X(3),X(4),X(5)

    maddm.h     llAcc,llAcc,sseX,ssoH ul,#1
                                    ;llAcc += X(2)*H(2) + X(3)*H(3) ||
    ld.d        ssssH,[caH+c]4*W16  ;ssssH <- H(4),H(5),H(6),H(7)

    ; 4 taps
    maddm.h     llAcc,llAcc,ssoX,sseH ul,#1
                                    ;llAcc += X(4)*H(4) + X(5)*H(5) ||
    ld.d        ssssX,[caX+c]4*W16  ;ssssX <- X(6),X(7),X(8),X(9)

    maddm.h     llAcc,llAcc,sseX,ssoH ul,#1
                                    ;llAcc += X(6)*H(6) + X(7)*H(7) ||
    ld.d        ssssH,[caH+c]4*W16  ;ssssH <- H(8),H(9),H(10),H(11)
    ; 4 taps
    maddm.h      llAcc,llAcc,ssoX,sseH ul,#1
                                     ;llAcc += X(8)*H(8) + X(9)*H(9) ||
    ld.w         sseX,[caX+c]-(FIRTAPS-2-DECFAC)*W16;ssssX < X(10), X(11)
                                     ;Update caX to the next input-sample
                                     ;which means X(2) (not X(1))
    maddm.h      llAcc,llAcc,sseX,ssoH ul,#1
                                     ;llAcc += X(10)*H(10) + X(11)*H(11)

;------------------- Kernel Ends ------------------------------------

;------------------- PostProcess, Store the filter output---------------

    ; Save to aR Output Buffer
    shas         dTmp,loAcc,#16      ;Format the filter output to 16-bit
                                     ;saturated value
    st.q         [aR+]W16,dTmp       ;Store result in the Output-Buffer
                                     ;Repeat the loop (int)(nX)/DECFAC)
    loop         aSampleCnt,FirB4_InDataLSU1

    FReturn

;------------------- Undefine the Registers -----------------------------

.undef      DECFAC
.undef      FIRTAPS
.undef      capX
.undef      capH
.undef      aR

.undef      nX

.undef      caH
.undef      caeH
.undef      caoH

.undef      caX
.undef      caeX
.undef      caoX

.undef      aSampleCnt
.undef      aTapCnt
```

```
.undef       ssssX
.undef       sseX
.undef       ssoX

.undef       ssssH
.undef       sseH
.undef       ssoH

.undef       llAcc
.undef       leAcc
.undef       loAcc

.undef       dTmp
.undef       dTapLoops
```

8.2 Signal Processing (feature extraction)

8.2.1 Implementing FIR BPF on TC1796

The following TriCore source code is implemented in assembly using Tasking compiler syntax. It is the implementation of the band-pass FIR filter which calculates the energy within defined frequency range and time window. This implementation has a predefined size of 44-taps stored in FIRTAPS. It is a special implementation of FIR filter which does not have its own delay-line and works directly on the input data buffer.

8.2.1.1 TC1796 Source Code

```
;************************************************************************
; DataL KnockBpfFir_44T(
;           cptrDataS    *X,
;           cptrDataS    *H,
;           DataS        *R,
;           int          nX_)
;
; INPUTS:
;       FIRTAPS Define the number of filter coefficients(set during
;               compilation)
;       X       Circ-Ptr to Input-Buffer
;       H       Circ-Ptr of Coeff-Buffer of size nH=44
;       R       Ptr to Output-Buffer = Filter output values in 1Q15
;       nX      Number of Input values to process ->
;               If nX=4 FIRTAPS=44 it means that filter will use
;               nX+FIRTAPS-1
;               input values 4+44-1 = 47!
;
;
; OUTPUTS:
;       R(nX)   Output-Buffer = Filter output values in 1Q15 (Optioal
;               for Test!)
;       *X      Keep the updated pointer to the next input values
;               Useful in case the processing is split to few calls
;
; RETURN:
;       sum(abs(R[i]))
;
```

```
; DESCRIPTION:
;       FIR filter transversal structure(direct form),
;       Fixed size of 44-taps
;       Input block processing,
;       16-bit fractional input, coefficients and output data format,
;       Optimal implementation requires filter order to be multiple of 4
;       ----------------------------------------------------------------
;       Implementation of FIR filter which working directly on the
;       input-buffer.
;       There is no separate Delay-Line!
;       Defining input-data as circular buffer allow flexible concurrent
;       management of data acquisition and processing.
;       Usually the size of the buffer should be bigger then the filter
;       size.
;       The circular buffer can be also used as linear buffer by ;
;       selecting adequate size. Before each call to this function the
;       input-;buffer pointer should point to the start of the buffer.
;       Other possibility it to modify the implementation so that linear
;       input-buffer will be directly supported
;       The best implementation is achieved with constant filter size.
;
; ALGORITHM:
;       An FIR filter with filter order nH can be represented by
;       following mathematical equation
;
;       R(n) = X(n) * H(0) + X(n-1) * H(1) + ..+ X(n-nH+2) * H(nH-2) +
;             X(n-nH+1) * H(nH-1)
;
;       where,
;       R(n)                : Output sample of the filter at index n
;       X(n)                : Input sample of the filter at index n
;       H(0),H(1),H(2),..   : Filter coefficients
;       nH                  : Filter order (number of coefficients)
;
; TECHNIQUES:
;       1)  Loop unrolling, 4 taps/loop.
;       2)  Use of packed data load/store
;       3)  Delay line implemented as circular buffer.
;       4)  Coefficient buffer implemented as circular buffer
;       5)  Use of dual MAC instructions.
;       6)  Intermediate results stored in 64 bit register (16 guard
;           bits).
;       7)  Instruction ordering for zero overhead Load/Store
;
```

```
; ASSUMPTIONS:
;       1)  Filter order should be multiple of 4 and minimum filter
;           order is 8
;       2)  Inputs, outputs, coefficients should be in 1Q15 format.
;
; MEMORY NOTE:
;
; _______________________________________________________________________
;
;  Pointer      Pointer      Variable                    Alignment
;               Type                               IntMem or      ExtMem-
;                                                  ExtMem+Cache   NoCache
; _______________________________________________________________________
;
;   H           Circ         H(0), H(1), H(2),       8-bytes      8-bytes
;                            ..., H(FIRTAPS-1)
; _______________________________________________________________________
;
;   X           Circ         X(n), X(n+1), X(n+2),..  2-bytes     2-bytes
; _______________________________________________________________________
;
;   R           Lin          R(n), R(n+1), R(n+2),..  2-bytes     2-bytes
; _______________________________________________________________________
;
;
; REGISTER USAGE:
;       a2, a3, a4, a5, a6, a7, a12, a13, a14, a15.
;       d4, d5, d6, d8, d9, d10, d11, d12, d13, d14
;
; CYCLE COUNTS:
;       38 cyc/sample when nX=100 (Save to aR Output Buffer not used)
; CODE SIZE:
;       86 bytes;
;***********************************************************************
```

```
;-------------------- Register Allocation -------------------------------
.define      FIRTAPS      "44"         ;

.define      capX         "adArg1"     ;a4 Ptr to Circ-Ptr of Input-Buffer
.define      capH         "adArg2"     ;a5 Ptr to Circ-Ptr of Coef
.define      aR           "adArg3"     ;a6 Ptr to Output-Buffer

.define      nX           "wArg1"      ;d4 Number of samples to process

.define      caH          "a2/a3"      ;Circ-Ptr of Coeff-Buffer
.define      caeH         "a2"         ;Even-Reg of Circ-Ptr
.define      caoH         "a3"         ;Odd-Reg of Circ-Ptr

.define      caX          "a12/a13"    ;Circ-Ptr of Input-Buffer
.define      caeX         "a12"        ;Even-Reg of Circ-Ptr
.define      caoX         "a13"        ;Odd-Reg of Circ-Ptr

.define      aSampleCnt   "a14"        ;Loop-Cnt-Reg (loop ext)
.define      aTapCnt      "a15"        ;Loop-Cnt-Reg (loop int)

.define      ssssX        "e10"        ;Filter internal state
.define      sseX         "d10"        ;Even-Reg
.define      ssoX         "d11"        ;Odd-Reg

.define      ssssH        "e8"         ;Filter Coeff.
.define      sseH         "d8"         ;Even-Reg
.define      ssoH         "d9"         ;Odd-Reg

.define      llAcc        "d12/d13"    ;Filter result
.define      leAcc        "d12"        ;Even-Reg
.define      loAcc        "d13"        ;Odd-Reg

.define      dTmp         "d14"        ;Generic temporary Data-Reg
.define      dTapLoops    "d15"        ;
.define      dSum         "d2"         ;Return value


;=================== Executable Code ==================================
    .align   32

KnockBpfFir_44T:

    FEnter       0
    ;Align

    nop
```

```
;----------------------------------------------------------------------
    mov         dSum,#0

    add         nX,#-1              ;-1 loop adjust for FirB4_InDataL ||

    mov.a       aSampleCnt,nX

    ld.da       caX,[capX]          ;Load the Circ-Ptr of Input-Buffer
                                    ;to Reg-Pair caX
    ld.da       caH,[capH]          ;Load the Circ-Ptr of Coef-Buffer to
                                    ;Reg-Pair caX

    ;>>2 4Taps/loop,-1 loop adjust -1 additional iteration after Kernel
    mov         dTapLoops,#(FIRTAPS/4-2);
    mov.a       aTapCnt,dTapLoops   ;Initialize the aTapCnt

;------------------- Loop for Input Buffer Starts ---------------------

FirB4_InDataLS:

    mov         leAcc,#0            ;Clear the Even-Reg of llAcc ||
    ld.d        ssssH,[caH+c]4*W16  ;ssoH <- H(0),H(1), H(2),H(3)

    mov         loAcc,#0            ;Clear the Odd-Reg of llAcc ||
    ld.w        ssoX,[caX+c]2*W16   ;ssoX <- X(n),X(n-1)

;------------------- Kernel -------------------------------------------

;The index i,j of X(i),H(j) (in the comments)are valid for first loop
;iteration
;For each next loop i,j should be decremented and incremented by 4 resp.
;'n' refers to current instant

FirB4_TapLS:   ;OPT ALIGN = xxx00,xxx20, xxx40..., xxx80

    maddm.h     llAcc,llAcc,ssoX,sseH ul,#1
                                    ;llAcc += X(n)*H(0) + X(n-1)*H(1) ||
    ld.d        ssssX,[caX+c]4*W16  ;ssssX<- X(n-2),X(n-3),X(n-4),X(n-5)

    maddm.h     llAcc,llAcc,sseX,ssoH ul,#1
                                    ;llAcc += X(n-2)*H(2) + X(n-3)*H(3)
    ld.d        ssssH,[caH+c]4*W16  ;ssssH <- H(4),H(5),H(6),H(7)

    loop        aTapCnt,FirB4_TapLS

;------------------- Kernel Ends --------------------------------------
```

```
    maddm.h      llAcc,llAcc,ssoX,sseH ul,#1
                                    ;llAcc += X(n-nH+4)*H(nH-4) +
                                    ;         X(n-nH+3)*H(nH-3) ||
    ld.w         sseX,[caX+c]-(FIRTAPS-3)*W16 ;Set the caX for next samp
                                    ;
    maddm.h      llAcc,llAcc,sseX,ssoH ul,#1
                                    ;llAcc += X(n-nH+2)*H(nH-2) +
                                    ;         X(n-nH+1)*H(nH-1) ||

    mov.a        aTapCnt,dTapLoops  ;Initialize the aTapCnt

;------------------- PostProcess, Store the filter output--------------

    ; Optional: Save to aR Output Buffer
    shas         dTmp,loAcc,#16     ;Format the filter output to 16-bit
                                    ;saturated value
    st.q         [aR+]W16,dTmp      ;Store result in the Output-Buffer

    ; Return Sum of the 16-bit filters abs value outputs in 32-bit reg
    abs          loAcc,loAcc

    adds.u       dSum,dSum,loAcc

    loop         aSampleCnt,FirB4_InDataLS  ;Loop = length of data

    FReturn

;------------------- Undefine the Registers ----------------------------

.undef       FIRTAPS
.undef       capX
.undef       capH
.undef       aR

.undef       nX

.undef       caH
.undef       caeH
.undef       caoH

.undef       caX
.undef       caeX
.undef       caoX
```

```
.undef      aSampleCnt
.undef      aTapCnt

.undef      ssssX
.undef      sseX
.undef      ssoX

.undef      ssssH
.undef      sseH
.undef      ssoH

.undef      llAcc
.undef      leAcc
.undef      loAcc

.undef      dTmp
.undef      dTapLoops
.undef      dSum
```

9 References

E. B. Hogenauer, “An Economical Class of Digital Filters for Decimation and Interpolation.” IEEE Trans. On Acoustic, Speech, and Signal Processing, Vol. ASSP-29, April 1981, pp. 155-162

Intersil FN4371.1 November 1998, Data Sheet ,”HIP9011 Engine Knock Signal Processor”

Application Report- “Engine Knock Detection Using Spectral Analysis Techniques With a TMS320 DSP” Texas Instrument, 1995

A.V. Oppenheim and R.W. Schafer, "Discrete-Time Signal Processing" Prentice Hall, Englewood Cliffs, New Jersey 07632, 1989. ISBN 0-13-216292-X

10 Definitions, Acronyms, Abbreviations

DMA:	Direct Memory Access
DSP:	Digital Signal Processing
FIR :	Finite Impulse Response
Fs:	Sampling Frequency
AAF:	Anti-aliasing filter
SW:	Software
HW:	Hardware
ASSP:	Application Specific Standard Product
ECU:	Electronic Control Unit
RC filter:	Resistance & Capacitance Filter
A/D:	Analogue to Digital Converter
DPRAM:	Dual Ported Random Access Memory
ASIC:	Application Specific Integrated Circuit