Technical Report

Pattern Recognition and Image Processing Group
Institute of Computer Aided Automation
Vienna University of Technology
Favoritenstr. 9/1832
A-1040 Vienna AUSTRIA
Phone: +43 (1) 58801-18351
Fax: +43 (1) 58801-18392
E-mail: irfan.adilovic@tuwien.ac.at
URL: http://www.prip.tuwien.ac.at/

PRIP-TR-104 February 2, 2006

An audio filter framework for JOP

Irfan Adilovic[1]

Abstract

The aim of this project is to develop a conceptual implementation of a filter framework for the Java Optimized Processor (JOP). The target system is a board with (among other things) a Cyclone FPGA and an AC'97 audio codec. The filter framework and the target system are not intended for industrial use, but rather for educational purposes about programming embedded systems in pure Java without foregoing any cross-platform capabilities of Java. The filter framework needs to run without changes on desktop systems (in a free JVM, like SUN's or IBM's) as well as on the targeted embedded system.

[1]Thanks to Martin Schöberl for his support, without which this work would not be possible.

1 Introduction

Embedded systems are mostly programmed in languages like C, and it would be interesting to see a pure Java program running without any modifications on both a desktop computer and an embedded system. The project is offered as part of the *Digital Signal Processors* course at the Vienna Technical University.

1.1 Hardware

The target system consists of a Cyclone FPGA board with I/O extensions offering a USB interface, SD-Card connectors, bluetooth communication, a serial interface and an AC'97 audio codec. See [11] and [10] for further info.

1.2 Theoretical background

An audio filter is a type of filter used for processing sound signals. An audio filter is typically designed to pass some frequency regions through unattenuated while significantly attenuating others. Analog filters achieve these goals by letting the source signal (directly) through analog circuitry whereas digital filters achieve the goals by passing the sampled source signal through a mathematical function defined by the filter. However the signal first needs to be sampled by an analog to digital converter.

A finite impulse response filter outputs a weighted sum of past and present input values[1]. It is a simple digital filter whose response to an impulse ultimately settles to zero regardless of the filter's configuration. This is in contrast to infinite impulse response filters which may have an infinite response to a single impulse.

The finite impulse response filter calculates the response to an impulse by multiplying past and present input samples with some fixed coefficients and summing up the result to yield a response (processed sample), there is no feedback (feeding the output back as input). If the input samples are i_0, i_1, i_2, i_3, etc. and the FIR coefficients are c_0, c_1 and c_2, then the output of the filter will be:

$o_0 = c_0 i_0 + c_1 0 + c_2 0$
$o_1 = c_0 i_1 + c_1 i_0 + c_2 0$
$o_2 = c_0 i_2 + c_1 i_1 + c_2 i_0$
$o_3 = c_0 i_3 + c_1 i_2 + c_2 i_1$

and so on.

A trivial example of a FIR filter is the moving average filter: it outputs the average of the last n input values. This filter, when averaging over n values, implemented as FIR filter would have n identical coefficients $\frac{1}{n}$.

Further details (such as coefficient generation) is outside the scope of this paper.

[1] One could think of *weighted past, present and future input values* for non-real-time filtering where data is available.

1.3 Goals

The goal of the project is to develop a finite impulse response filter in Java and apply it on a sound signal that is fed into the filter independently of the filter implementation. In addition to that, a small framework needs to be developed, which allows for chaining of different filters and implementation of further filters which can be used together. The implementation needs to run on a desktop computer with a free JVM and on the target embedded system.

1.4 Previous Work and Current State

As ever, the persistent increase in processor speeds and the persistent drop in costs of such, has made it feasible to have embedded systems programmed in Java. In the beginnings, Java was unpopular because it has a further layer of abstraction (virtual machine) as compared to languages like C, and because it was simply slower (see section 4.2 for one of the reasons). To illustrate, until May 2000, SUN's KVM was ported to more than 25 devices that are in the class of 16/32-bit microprocessors with a few hundred kilobytes total memory [8].

The idea of embedded Java existed within months of the initial launch of the Java technology in 1995 [7].

As of this writing, there are several commercial embedded systems for Java, having either a hardware or software virtual machine. Examples include EJC [14] running atop a RTOS on a 32-bit ARM720T processor, JStamp [3] on the aJ-80 processor with a hardware VM, leJOS [1] on a 16-bit Hitachi H8300 with a VM that fits in 32kb, commercial microprocessors based on picoJava [9], etc.

However the DSP market is relatively silent on the issue, there are no Java-DSP breakthroughs as of now. Capable boards exist, but they are not really mainstream DSP boards.

2 Design and Implementation

The imposed goals allow for a clear design of the framework in Java. We will have

- an interface common to all filters, we name it `Filter`
- an abstract class which the filters may extend and which encapsulates common functionality, we name it `AbstractFilter`
- a meta-filter which chains other filters and acts transparently as a filter itself, we name it `FilterChain`
- a finite impulse response filter, with the name of `FIR`

New filters may be implemented as needed, they just need to implement the `Filter` interface, and possibly use the functionality provided in `AbstractFilter`.

2.1 Filter

The only function of the `Filter` interface is to define a function to process a buffer of audio samples. This function needs to accept an array of source integer samples (regardless of the actual resolution, assuming it is not greater than 32 bits[2]), the offset at which to start processing in the source array, an array where to store the result, the offset at which to start storing the result and finally, how many samples to process. The signature of the only declared function is:

```
public int processBuffer(int[] src, int srcOffset, int[] dest, int destOffset, int length);
```

Originally, the `Filter` interface included the declaration of a function called `processSample` - why it was removed will be discussed later.

2.2 AbstractFilter

The abstract class `AbstractFilter` is meant for factorization of any common functionality of filters, and originally implemented `processBuffer` so that each sample is processed by the `processSample` function of the concrete filter. Therefore the concrete filter does not have to implement the `processBuffer` method of the `Filter` interface, if it can rely on the obvious functionality provided by `AbstractFilter` (which is usually the case). This was left out because of inlining restrictions of Java, which will be discussed later. The class was not removed after the removal because it *may* prove useful for any future common functionality. It should be noted that the abstract class does not have a performance impact, since polymorphism induced through the usage of the `Filter` interface necessitates dynamic binding of `processBuffer` calls.

`AbstractFilter` contains a constant named `PRECISION` which defines the precision of fixed point numbers. This constant could have been moved to the `Filter` interface itself, but for more flexibility it was kept in the abstract class because here, it can be manipulated in run-time when needed (imagine switching from 16-bit to 20-bit input precision at runtime, requiring more space for the integer part in the fixed-point representation).

2.3 FilterChain

With the meta filter `FilterChain`, one can append any filter implementing the `Filter` interface (including an instance of `FilterChain` itself) to a list of filters through which the input signal is passed and returned. This can be used to transparently create different kinds of complex filters and then reuse and combine them as needed.

`FilterChain`s would also be very useful in dynamic configuration of filters, for example when implementing an equalizer - where the user's actions can lead to creation, chaining and deletion of several filters.

[2]The upper bound may be filter-dependant. Current implementation of the FIR filter guarantees to work for samples of 24 bits in size, see section 2.4.1.

2.4 `FIR`

The finite impulse response filter is initialized with an array of integer coefficients which are converted to fixed point numbers. These are then normalized to yield a sum of 1, so that no overall amplification or attenuation is induced through the coefficients (some amplification or attenuation is necessarily present, this is explained in the following subsection). Other than that, the filter implements the `processBuffer` function of the `Filter` interface and applies the proper finite impulse response logic on the input. Past samples are stored in a cyclic buffer and subsequently used in calculating the processed sample during the run time of `processBuffer`.

2.4.1 Fixed-point numbers and the quantization error

With a precision of p, a fixed-point number, contained in an integer has 2^p quantization levels between -1 and $1 - 2^{-(p-1)}$. Each input value that falls between two adjacent quantization levels is interpreted as one of the two, depending on the rounding. In the FIR filter implementation the rounding mode is truncation, so always the smaller of the two adjacent levels is chosen.

The maximum quantization error thus approaches the distance between two adjacent quantization levels $\bar{e} \to 2^{-(p-1)}$. In other words, the quantization error is always in the interval $[0, 2^{-(p-1)})$.

It is important to note that as a consequence of the rounding mode, the mean error is not zero, it is $\bar{e}_{mean} = -2^{-p}$ (assuming uniform distribution of input) because the interpreted values are on the average smaller than actual values.

There is also a quantization error induced by the ADC that captures the sound signal. The same argumentation about the quantization error holds[3].

In the current implementation of the FIR filter, the chosen fixed-point format is s23.8 and ADC is configured for 16-bit precision, so the maximum quantization error induced in combination is

$$\begin{aligned} \bar{e}_{max} &\to 2^{-(8-1)} + 2^{-(16-1)} \\ &\to 2^{-7} + 2^{-15} \\ &\to 0.007843017578125 \end{aligned}$$

In the initial phase, the framework was developed and tested on a desktop computer with SUN's JVM. The Java Sound Subsystem provides sound data in a different format: a byte buffer instead of an integer buffer, with both channels intermixed. A small adapter utility was developed which converts such a buffer into two separate buffers of integers.

The targeted embedded system provides data also in a not directly usable format - two 16-bit samples in one 32-bit integer. This decomposition is trivial and was implemented in-line in the source code which uses the filter functionality.

[3] Assuming the ADC also truncates the input to the lower quantization level. Were that not the case, the maximum error would be $\bar{e} \to 2^{-p}$

3 Optimization

After the framework was implemented and tested in a desktop computer environment, it has proven to perform quite badly on the targeted embedded system. Three issues were identified and optimized: *inlining*, *wrapping of cyclic indices* and *prefetching of member variables*.

3.1 Inlining of `processSample`

The aforementioned `processSample` method declared in the `Filter` interface induced a severe performance hit because of the added function call overhead per sample.

Function call overhead per sample, with the `processSample` utilized, is at least k where k is the number of cycles needed for a function call. It can be more, if `processSample` is used only indirectly through `processBuffer` (i.e. because of `processBuffer` call overhead - if utilized).

With `processSample` inlined, the overhead per sample is equal to $\frac{k}{n}$ where n is the amount of samples processed per `processBuffer` call, considerably less.

As discussed in section 2.10.3. of the Java Virtual Machine Specification [5] and section 8.4.3.3. of the Java Language Specification [2], safe inlining is possible only for `final` methods[4]. This is a consequence of the design that is at the very heart of the language: dynamic binding. Not to mention, even for final methods, inlining cannot explicitly be controlled by the programmer: if the method being called is final, the compiler *can* inline the function, it is not obliged to do so.

The problem with dynamic binding is that at compile time, the compiler does not always know the exact type of the object on which `processBuffer` is being called, because `processBuffer` is in the `abstract` class `AbstractFilter` and the `processSample` method is at that point only an abstract method. Assuming we have more filters than just `FIR` implemented, it could be any of the `processSample` implementations that would eventually get called.

The compiler does know the exact type of the object at compile time for declarations like `FIR fir = new FIR(...);`, and in such cases, the compiler can inline the method calls. However, using *all* filters in such a way is very limiting, because absolutely no dynamic reconfiguration of filters would be possible: we would not be able to combine or replace filters during run-time. The whole point of a filter *framework* would be ruined. The `Filter` interface and the `AbstractFilter` abstract class would be useless, because `processBuffer` calls on variables of these types would always be dynamically bound (for example `Filter f = new FIR(...)`). The meta filter `FilterChain` would need to be implemented for each filter separately, because the usage of the `Filter` type within `FilterChain` would disable inlining.

Hence, in order to gain control over inlining, manual inlining was undertaken. The `processSample` method was removed from the `Filter` interface, the `processBuffer`

[4]In his article about the Java HotSpot Virtual Machine's performance [6], Steven Meloan discusses the ability of the HotSpot VM to inline non-final methods. However, the discussed methods are subject to the same restrictions as the original ones: if more than one implementation can lie behind the method call, one cannot inline that call.

method was removed from `AbstractFilter` and both combined in the `FIR` filter (and expected to be thus combined in future filters).

3.2 Wrapping of cyclic indices

The buffer of past samples in the `FIR` filter was of cyclic nature. This meant that when processing the samples, an index had to be run manually through the buffer and wrapped around when it came to the buffer's end (forward iteration) or beginning (backward iteration).

Using `if` statements in such situations introduces jumps in the bytecode which are expensive (see cycle lengths of `goto` and various `if` instructions in [13]) for an inner loop. These wraps were optimized by forcing the buffer length to be a power of two, so that a simple bitwise *and* operation with a mask will do. To illustrate, the following code:

```
if (−−someIndex < 0)  someIndex += bufferLength;
```

was replaced with

```
someIndex = (someIndex - 1) & l_indexWrappingOptimizer;
```

where `l_indexWrappingOptimizer` is defined as `bufferLength - 1` and introduced to avoid repeated evaluation of the constant `bufferLength - 1` expression. An obvious equivalent of this code replacement was also made for the forward iteration variant.

3.3 Prefetching of member variables

Accessing member variables is expensive: accessing a static class variable costs 17 cycles and accessing an object variable costs 25 cycles, compared to 1 to 2 cycles for loading local variables on the JOP [13] [12]. Fetching them once just after entry into `processBuffer` improves the speed of their access within the processing loops of `processBuffer`. This improvement is trivial - local variables were introduced, which reduced member variable access to *one* per member variable per `processBuffer` call as compared to $O(bufferLength)$ and $O(bufferLength \cdot filterLength)$ (depending on whether the access happens in the outer, buffer-processing loop, or the inner, filtering loop).

4 Problems and performance issues

Several problems were encountered during the development of the framework, two major of them being inherent to Java.

4.1 Inlining

C allows for explicit declaration of functions that are to be inlined (See section 6.7.4 of the C99 standard [4]), but in Java, as discussed, there are no guarantees about inlining. The programmer can only fulfill the preconditions for inlining and hope that the compiler

will inline the function. In our case, it is impossible to avoid dynamic binding, having the goals of the project in mind.

Inlining can be done directly in code (and we have partly done it in our implementation as mentioned above), but this reduces the reusability of the code itself. We could have avoided function calls even per buffer by manually inlining all the code of the filter(s) directly in the code which fetches sound data. But then, our filters would be utterly unusable in different environments. Apart from that, filter chaining, dynamic filter placement and configuration would not be feasible at all.

4.2 Array access in Java and C

Java guarantees, in utter contrast to C, that an incorrect array access shall be trivially programmatically detectable. In C, there is no guarantee as to the access-correctness whatsoever [4]. However, Java throws an IndexOutOfBoundsException if we are accessing an array outside of its valid index range (section 2.15.4 in [5]). This means that in a trivial and **successful** array access, there is at least:

1. *A comparison of the index with 0*,
2. *A fetch of the upper bound value (member variable access)*,
3. *A comparison of the index with the upper bound*,
4. Memory address calculation based on array beginning and index[5], and
5. Read or write of data.

The emphasized part of the above list is missing in C, which makes C far less reliable, but much more performant. Such behavior of Java makes iteration over arrays generally slow, and especially those inner loops which are sometimes even programmed in assembler for better performance.

Such a problem is inherent to Java[6] and cannot just be solved by refactorization, further code optimization etc. One would have to have some kind of a special compiler *and* virtual machine which allow for stripping of such checks in order to make the code more performant, but the programmer would be faced with all the problems a C programmer is faced with in working with arrays and debugging them. This would, of course, beat the point of using Java in the first place, and is not a viable option.

[5] It should also be noted that getting the information about the array beginning in memory is different in C when compared to Java: in C, the array itself is a pointer and the information is readily available. In Java, the information is hidden somewhere within the array object, most probably as a member variable, whose fetch requires more cycles than reading the pointer value in C - another performance gain in C.

[6] It is the JVM which provides the guarantee about array access correctness, not the compiled code. The compiled code contains an instruction like `iaload` or `iastore` which is interpreted by the JVM. There is simply no facility for direct memory access, everything is provided through such abstract JVM instructions [5].

4.3 Other problems

Other problems that were encountered were mainly of interpretative nature, or problems external to the project.

Among these was the choice of `Filter` interface so that data is easily converted to required form on different systems. To illustrate: the targeted embedded system provides two 16-bit samples in a 32-bit integer - separately for each channel, and the Java Sound Subsystem in a desktop computer environment provides us with byte data where the channels of the input are interleaved.

Another problem was about the signedness of the samples - interpreting them as signed or unsigned makes a big difference on the output. This problem was identified by testing with a signal generator and an oscilloscope - spikes in the output signal were visible at the zero region of the signal.

A reproducible problem with the JOP's JVM was also identified: polymorphism and dynamic binding caused exceptions in run-time. This problem was easy to avoid, and is of nature external to the project. Specifically, method invocations on a variable declared and initialized like `Filter f = new FIR(...)` would cause exceptions. Replacing that part with `FIR f = new FIR(...)` temporarily resolved the problem.

Another problem emerged while debugging high-pass filters. The normalization routine functional for filters with only positive coefficients (typical low-pass filters) was not meaningful for filters with negative coefficients (typical high-pass filters) and was thus turned off for such cases. This required manual feed of fixed-point numbers to the filter.

Because of the problems that affected performance, the longest filter that was able to run without distorting the signal was of length 4. In order to visualize the kind of distortion, in figure 1, an output of a filter with more coefficients is provided.

In order to provide more execution time to the filter, we tried to reduce the sampling rate of the AC'97 chip so that filtering with larger FIR filters is feasible. We tried to reduce the sampling rate from the default and maximum of 48kHz to a lower one. However, problems with either the Wishbone interface [15] to the AC'97 chip or the chip itself did not allow for any frequency change. A simple sinusoid signal would pass properly, with no change through the JOP (without a filter) and the AC'97 chip with 48kHz, but very distorted with *any* other frequency setting (still no filter). The oscilloscope output looked rather like many sinusoidal signals overlayed, in principle similar to the output of the 8-coefficient filter without reduction in sampling rate (Figure 1).

These problems ultimately lead to the maximum filter length of only 4 coefficients[7].

[7]In order to gain hard data for the Results section, some tests were conducted during the revision of this document. In these tests, the AC'97 seemed to function properly with reduced sampling rate, software errors being absolutely impossible. There is a high probability that this testing was conducted on a board different from the three former boards, the former ones being probably faulty. However, because of time constraints, no further development or testing was performed.

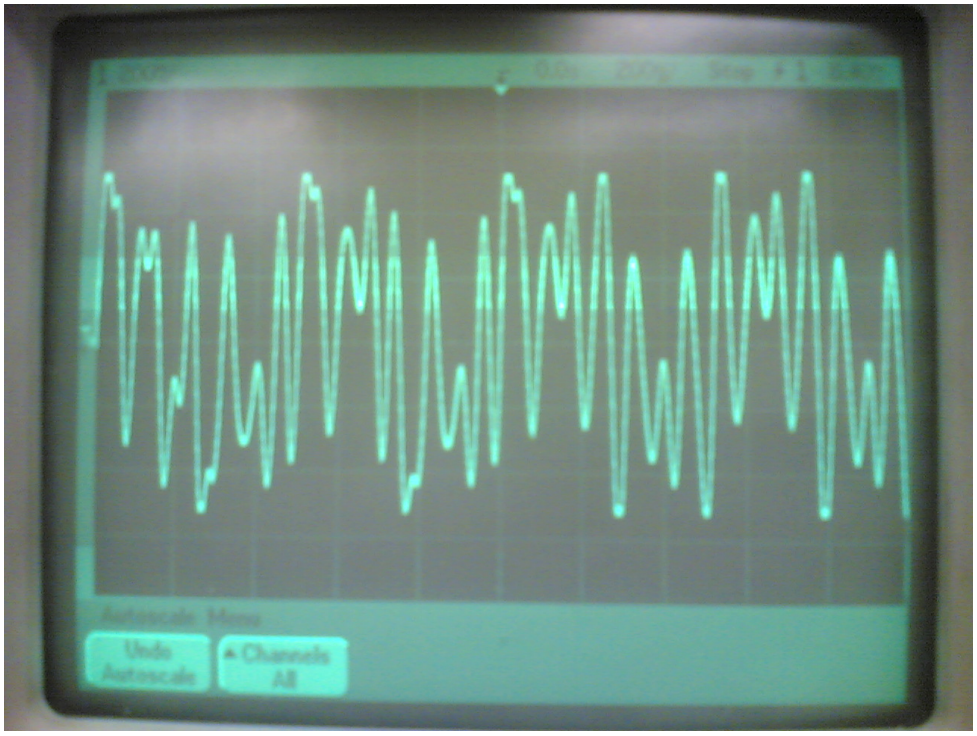

Figure 1: A sine wave passed through a `FIR` filter with 8 trivial coefficients $1, 0, 0, 0, 0, 0, 0, 0$. This filter performs no modification of the input whatsoever.

5 Results

The implementation of the filter was successful, the output of the sound chip was tested with a signal generator and an oscilloscope, and we could show that our filter did the work it was supposed to do.

5.1 A low-pass filter

A low-pass filter is a filter which passes low frequencies of an input signal unattenuated, while significantly attenuating higher ones. With 4 coefficients, one cannot hope for a first-class filter, far from that. However, one can generate a mediocre filter and test it, even when it is of little practical use.

Figure 2 shows the magnitude response for a low-pass filter with 4 coefficients, as generated by Matlab, being a reference for the actual state of affairs. Figure 3 which we extracted from tests with the signal generator and the oscilloscope, shows the actual magnitude response of the `FIR` filter at stake. The coefficients generated by Matlab were $0.027099, 0.472656, 0.472656, 0.027099$[8]. The magnitude response can be seen also in Table 1.

As can be seen, the actual magnitude response resembles *very well* the reference magnitude response.

[8]The filters were, of course, properly converted to the fixed-point format before being used by the filter framework.

low-pass (dB)	Freq.(kHz)	high-pass (dB)
-0.157770244	1	-19.1721463
-0.211003647	2	-14.2439654
-0.390842154	3	-11.24498874
-0.555943232	4	-8.873949985
-0.800103233	5	-7.210270215
-1.090628297	6	-5.882725754
-1.411621486	7	-4.791550332
-1.851772785	8	-3.87640052
-2.270185497	9	-3.122891548
-2.757372414	10	-2.498774732
-3.298877966	11	-1.851772785
-3.903586426	12	-1.432082955
-4.671743058	13	-1.031740684
-5.481767354	14	-0.724243453
-6.375175252	15	-0.500560114
-7.37112462	16	-0.336498559
-8.404328068	17	-0.211003647
-9.735647999	18	-0.104861108
-11.05683937	19	-0.052272312
-13.15154638	20	-0.140098031
-15.91760035	21	-0.500560114
-19.65933321	22	-1.350524706
-25.03623946	23	-3.0980392
-31.70053304	24	-6.339059235

Table 1: Data of magnitude response for the low- and high-pass filter.

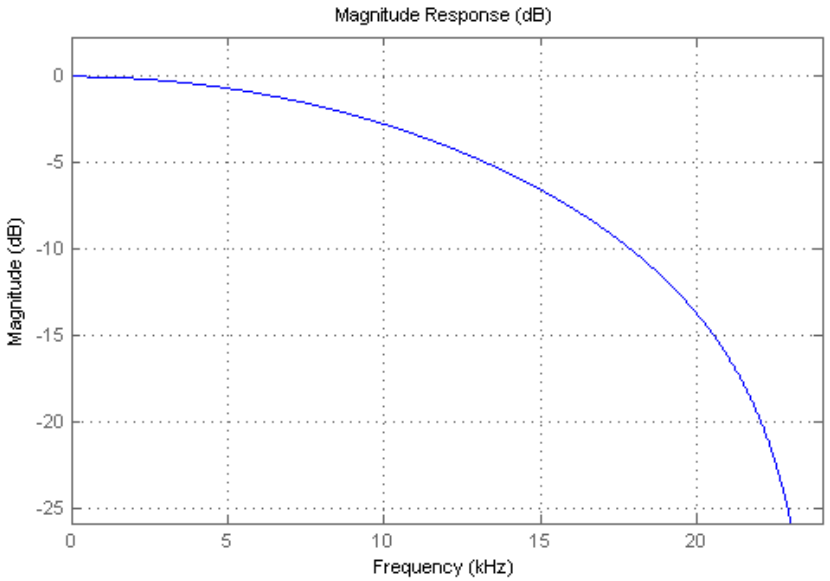

Figure 2: Reference magnitude response for a low-pass filter.

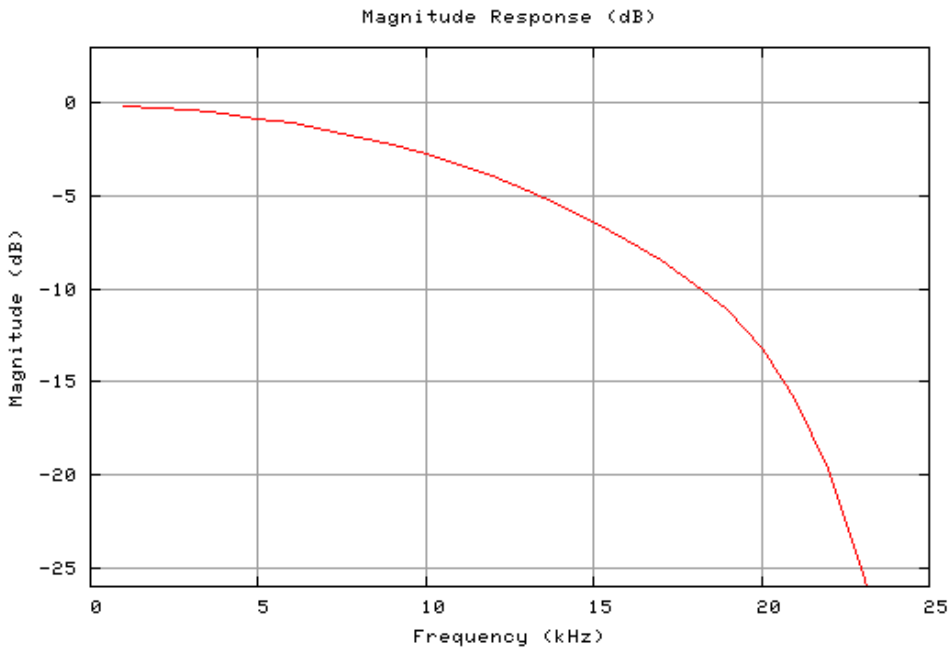

Figure 3: Actual magnitude response for a low-pass filter.

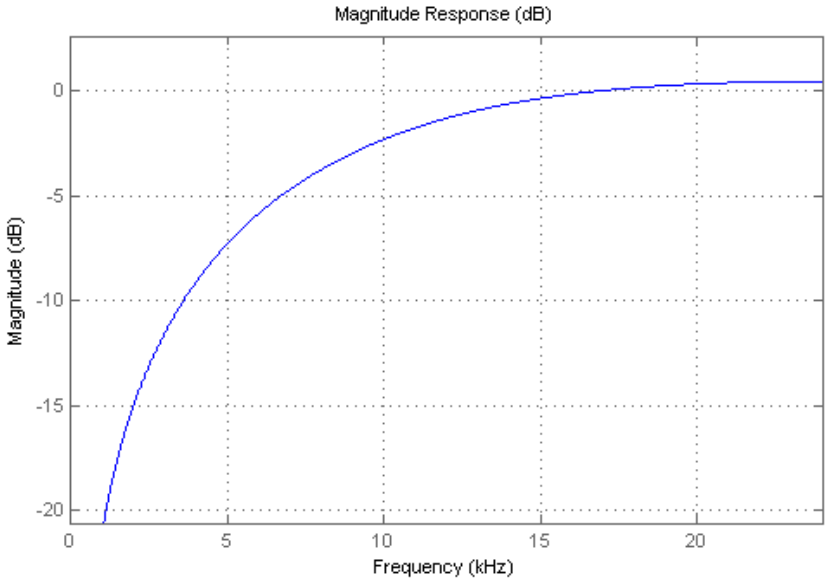

Figure 4: Reference magnitude response for a high-pass filter.

5.2 A high-pass filter

A high-pass filter is the exact opposite of the low-pass filter – it filters out low frequencies, and passes high ones.

Again generated by Matlab, figure 4 shows the reference magnitude response for a low-pass filter with 4 coefficients $-0.040685, -0.566304, 0.566304, 0.040685$. Figure 5, on the other hand shows how the filter performs in reality.

This filter too, resembles the reference very well, albeit not at frequencies higher than 20 kHz. This attenuating behavior for higher frequencies has been observed just as well without filtering, and it had the same factor of attenuation as with the high-pass filter. Hence the filter must be judged as correct at these frequencies too, since it is to be concluded that this is the property of the underlying hardware.

Apart from these two filters, a band-pass filter (which passes a fixed frequency band, and attenuates frequencies below and above the band) was tested, but its quality was very poor, as such a filter is by its nature more complex than the two filters discussed above.

Aural testing was also performed, and while the results were not very impressing, they were certainly detectable. The low effectiveness of the filters during aural testing lies within the fact that – due to the limitation on the number of coefficients – the filter cutoff is too much distributed, and the frequencies that should "suddenly" be cut off, are instead gradually cut off, making the effect harder to perceive. A real low- or high-pass filter would have a much shorter cutoff, and the effect would be that the desired frequencies, instead of being gradually attenuated, suddenly and (almost) completely disappear for the human ear.

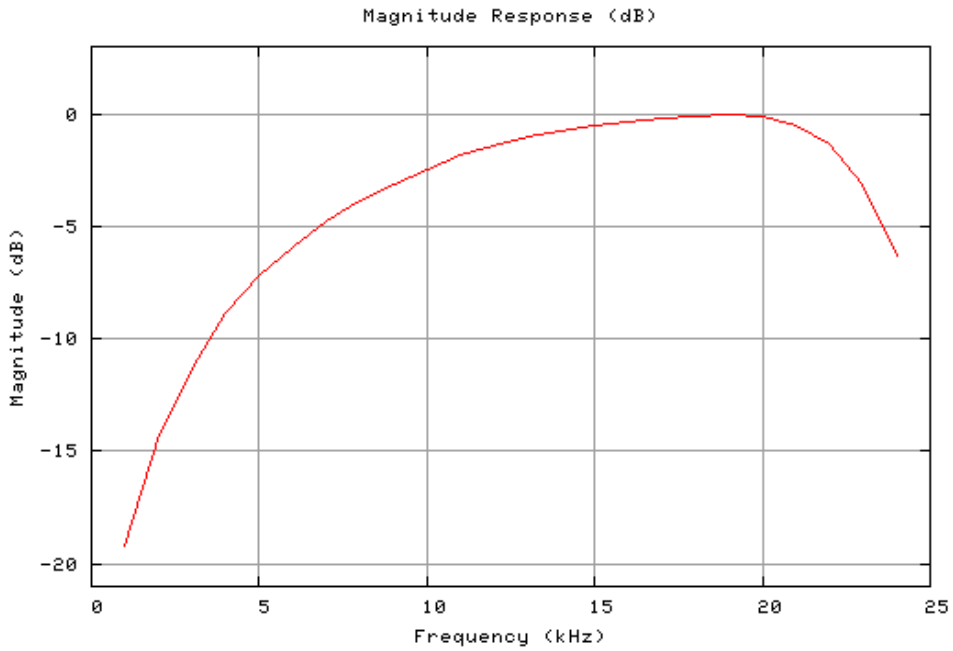

Figure 5: Actual magnitude response for a high-pass filter.

5.3 Conclusion

Unfortunately, the observed limit on the number of coefficients, caused by the various problems discussed in section 4, was quite severe and makes this embedded system not very attractive for industrial DSP applications.

In spite of these problems, however, the experiment per se was a success. We have developed a pure Java filter which runs on both desktop systems and this embedded system (and surely other embedded Java-capable systems on the market). It cannot be considered for industrial use because of performance issues, but it does work, and shows that pure Java programming of embedded systems, and DSP in Java is possible.

References

[1] Paul Andrews, Jürgen Stuber, Lawrie Griffiths, Brian Bagnall, Tim Rinkens, and Jose Solorzano. Java for the RCX. `http://lejos.sourceforge.net/`.

[2] James Gosling, Gilad Bracha, Bill Joy, and Guy L Steele. *The Java Language Specification*. Addison-Wesley Professional, second edition, 6 2000.

[3] Systronix Inc. JStamp Website. `http://www.jstamp.com`.

[4] IST/5. *ISO/IEC 9899:1999 Programming languages - C*. ISO/IEC, 4 2000.

[5] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Professional, second edition, 4 1999.

[6] Steve Meloan. The Java HotSpot Performance Engine: An In-Depth Look. `http://java.sun.com/developer/technicalArticles/Networking/HotSpot/`, 6 1999.

[7] SUN Microelectronics. LG Semicon Java Technology-Based Processor Announcement. *Pioneers' Progress with picoJava Technology*, 3, 1996.

[8] SUN Microsystems. J2ME Building Blocks for Mobile Devices. `http://java.sun.com/products/cldc/wp/KVMwp.pdf`, 5 2000.

[9] J. Michael O'Connor and Marc Tremblay. PicoJava-I: The Java virtual maschine in hardware. *IEEE Micro*, pages 45–53, March 1997.

[10] Martin Schöberl. JOP - Java Optimized Processor. `http://www.jopdesign.com/`.

[11] Martin Schöberl. I/O board for the DSP courses and SoC projects. `http://www.soc.tuwien.ac.at/courses/projects/dspio`, 2005.

[12] Martin Schoeberl. Evaluation of a Java processor. In *Tagungsband Austrochip 2005*, pages 127–134, Vienna, Austria, October 2005.

[13] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[14] Snijder Micro Systems. Java on the EJC. `http://www.embedded-web.com/products/java.html`.

[15] Rudolf Usselmann. AC 97 Controller IP Core. `http://www.opencores.org/projects.cgi/web/ac97/overview`, 2005.