EFFICIENT DSP IMPLEMENTATION OF MEDIAN FILTERING FOR REAL-TIME AUDIO NOISE REDUCTION

Stephan Herzog

Dept. of Digital Signal Processing
Technical University Kaiserslautern
Kaiserslautern, Germany

ABSTRACT

In this paper an efficient real-time implementation of a median filter on a DSP platform is described. The implementation is based on the usage of a doubly linked list, which allows effective handling of the operations needed for the running computation of a median value. The structure of a doubly linked list is mapped onto the DSP architecture exploiting its special features for an efficient implementation. As an application example, a real-time denoiser for vinyl record playback is presented. The application program consists of two main parts, namely a subsystem for click detection and a subsystem for click removal. Both parts can be implemented using median filters.

1. INTRODUCTION

Median Filters are well-known signal processing blocks that are used in various applications like image and speech processing, sound analysis [1], vocal separation [2] and audio noise reduction. Most applications use median filters for the removal of some sort of noise, e.g. salt and pepper noise in image processing, which is probably the most common application.
In audio signal processing, median filters are also commonly used for denoising, especially for the removal of noise of old vinyl records [3], [4]. Several implementations, e.g. as double median filters [5] and weighted and recursive median filters [6] and also adaptive variants [7], [8] have been presented in literature. Software programs for audio restoration of vinyl recordings like Diamondcut and GramoFile also make use of median filters for click removal. However, these programs process recorded audio data offline usually on a powerful computer, so the efficiency of the implementation is not critical. If a median filter is to be used online in an embedded DSP system, the performance of the implementation becomes an important issue. Being aware of this, Jones [9] has investigated and compared the performance of various mainly sorting based algorithms for the computation of median filters on a Texas Instruments DSP using C as the programming language. Under the term "finger trees" he also used a method based on doubly linked lists which performed well compared to the sorting based algorithms. However, the implementation in C exploits the hardware features of the DSP only as far as the compiler is capable. In our implementation we program a DSP directly in assembly language and make use of the specialized DSP architecture to efficiently map the structure of a doubly linked list onto it. Target hardware is a 24 bit Freescale Symphony DSP, namely the DSP56374 with a core clock frequency of about 150 MHz.
A realtime implementation on a DSP for online usage can be applied directly during playback of a vinyl record, which has the advantage that the record hasn't to be digitized and processed offline before playback and the "vinyl feeling" can be preserved by the use of the record player. A real-time denoiser can be applied as a small electronic device that can for example be attached into a tape record loop of a preamplifier and could be a viable solution for people who don't have the possibility to digitize their records. Furthermore applications outside the audio world, e.g. for the filtering of sensor signals can profit from an efficient realtime implementation, since the efficiency also affects power consumption in embedded applications.

2. MEDIAN FILTER FUNDAMENTALS

The median M of a set of values divides the set into two equally sized halves, so that as many values are smaller than M as there are values larger than M. If we consider a sorted list of N values $x(n)$, N odd, the median M is simply the middle element $x(\frac{N-1}{2})$. It is related to the mean value, but shows some important differences. The main differences between the median and the mean value are:

- The computation of median is a nonlinear operation.
- The median is robust regarding outliers, i.e. a single value in the list with a very large magnitude does not influence the median.

The first property makes it impossible to give a system function $H(z)$ for a median filter in a signal processing sense, but rather the definition

$$M = \begin{cases} x\left(\frac{N-1}{2}\right) & N \text{ odd} \\ \frac{1}{2}\left[x\left(\frac{N}{2}\right) + x\left(\frac{N}{2}+1\right)\right] & N \text{ even.} \end{cases} \quad (1)$$

Usually median filters are chosen to have odd length N. The explicit nonlinear nature of the median also makes a closed-form description of its effects on audio signals impossible in general. For example a simple moving-average filter, which computes the mean value of N samples has a constant delay of $\tau = \frac{N-1}{2}$ if N is odd, as it is a linear-phase FIR-Filter. An analytical expression for the delay τ of a median filter can not be given and as it is in the range $0 \leq \tau \leq \infty$, depending on the input sequence. The delay from the input to the output of the filter is zero, if the incoming sample represents the new median value and thus appears instantly at the output. The delay can be infinite, if an incoming value never equals the median value in a length-N block and thus will never be present at the output of the median filter. For many signals the delay will be close to $\frac{N-1}{2}$, e.g. if a ramp signal is considered as

an input.
The second property of the median, it's robustness to outliers, is the property, that is exploited in most applications. Impulse-like noise from various sources can be seen as outliers and a median filter can be applied for the removal of them. During the time the wanted signal is superposed by an impulsive noise, the output of a median filter can then be used as a substitute. The output of the median filter represents something like the average of the signal during a time interval N which includes the impulse, but through its robustness to outliers, the impulse does not affect the output and thus can be filtered out completely. If impulses with small magnitude are present, lowpass-filtering may be an appropriate method for denoising. However, a median filter is able to completely suppress impulses with large magnitudes.

3. EFFICIENT IMPLEMENTATION ON A DSP

A median filter computes the median M of the input sequence $x(n)$ over a length-N time window and outputs it as its output signal $y(n)$. Each time a new sample arrives and the oldest is discarded, the median computation has to start again. As the median is the middle value in a sorted list, one approach for median computation involves sorting of all values in the time window and taking the middle element. There are several well-known sorting algorithms like quicksort and heapsort in computer science, which have specific advantages and disadvantages. Sorting is a task not very well suited to the structure of a digital signal processor. Additionally, the execution time for a sorting algorithm can have significant variations, depending on the data and can become big. Hence the choice of the algorithm for the DSP implementation should avoid extensive sorting which can be achieved by the use of a doubly linked list. The processing of the data in a doubly linked list has the lowest maximum execution time for one sample of all algorithms compared in [9].

3.1. DSP56374 architecture

To motivate the use of a doubly linked list for the implementation of a median filter, a short overview of the relevant parts of the DSP56374 architecture will be given. The main features of the DSP used for the computation of the median filter are its address registers and the corresponding address arithmetic. The DSP56374 has eight special address registers `r0-r7`, associated with offset registers `n0-n7` and modulo registers `m0-m7` [10]. The modulo registers provide an easy and effective way to implement ring buffers without additional compare operations for the address pointer. If the address pointer is increased beyond the highest address in the ringbuffer, it automatically wraps around to the lowest address in the buffer. This address arithmetic is done in the Address Generation Unit (AGU), which is separated from the Arithmetic Logical Unit (ALU), in which the main computations are carried out and thus produces zero overhead in the main program. Consequently, a ringbuffer is used to hold the input data $x(n)$ for a window length of N samples. Inserting a new value into the buffer automatically overwrites the oldest value in the buffer.

3.2. Doubly linked lists on the DSP

A doubly linked list is a data structure, that consists of sequentially linked data. The data is stored in nodes, where a node additionally contains two fields with the addresses of the previous and the next list node, see fig. 1. The first and last nodes are special since they only have one neighbour. So one of their address field points to a special value NIL (Not In List), which marks the beginning and the end of the list. The advantage of a doubly linked list compared to

Figure 1: *Principle of a doubly linked list.*

an array is that elements can be easily inserted into the list and also removed from the list just by modifying some pointers and without the need to copy data between different memory locations. Compared to singly linked lists, doubly linked lists are easier to traverse and sort but need additional storage for a second set of pointers.
In high-level programming languages like C, one list node is a compound data type containing three elements, the pointers to the previous and next node and the data of the node. On the DSP programmed in assembly language, there are no data types at all and the architecture has to be exploited to reflect a compound element to represent a list node. For this purpose three memory

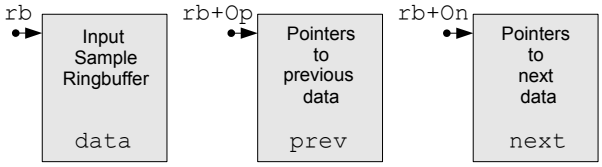

Figure 2: *Data organisation in DSP memory.*

sections are used to store the three components of a list node and are accessed via the DSP address registers (fig.2). The pointer `rb` points to the base address of the sample ringbuffer which contains the data. The memory regions which contain the pointers of the nodes are accessed relative to `rb` with two constant offsets. If we look at the three memory sections side by side, a node can be interpreted as a horizontal selection consisting of the input sample `x(rb)`, the `prev` pointer `prev(rb+Op)` and the `next` pointer `next(rb+On)`. The background of this data organization are the addressing capabilities of the DSP, which make it possible to access the offset memory sections with the use the offset registers `n0-n7` without changing the base pointers `r0-r7`, for example `MOVE x:(r0+n0),x0` leaves `r0` unchanged. Since the offsets `Op` and `On` are constant for a given filter length, only two different values have to be loaded into the offset register to access all elements of a node. For offsets smaller than 128, the DSP also offers the possibility to add an immediate displacement to an address, for example `MOVE x:(r0+63),x0`. Thus list nodes can be organized and accessed very efficiently using this structure.

3.3. Median filter algorithm

The median filter algorithm involves the standard operations for a doubly linked list like insertion and deletion of nodes and the corresponding pointer operations. In addition to the pointers to the smallest and largest element a pointer to the median element can be maintained. The idea behind the use of this pointer is, that once

the median is found, it only can change its position by one or stay in place if a new input sample is sorted into the list. Maintaining a pointer to the median avoids a new search after a new input value has been sorted into the list, only the pointer update is necessary.
To describe the median filter algorithm using a doubly linked list, the notation given in table 1 is used in the pseudocode. At the beginning, the data fields of all nodes are initialized with zeroes and the pointers are initialized with increasing values. For the computation of the median, the following procedures must be carried out on the doubly linked list:

Table 1: *Notation*

`ra`	Adress of sample $x(a)$.
`{ra}`	Node which contains the sample $x(a)$
`x{ra}`	Sample $x(a)$
`x{r=ra.prev}`	Sample previous to $x(a)$
`x{r=ra.next}`	Sample following sample $x(a)$
`{ra}.prev`	previous address field of the node containing $x(a)$
`{ra}.next`	next address field of the node containing $x(a)$
`rL, x{rL}`	Position and value of the largest element in the list
`rS, x{rS}`	Position and value of the smallest element in the list
`rM, x{rM}`	Position and value of the median element

- Replacement of the oldest audio sample in the length-N ringbuffer. The ringbuffer is handled automatically by the DSP address generation unit.
- Check if the case has occurred that the new sample overwrites one of the special values smallest or largest sample. Adjustment of pointers if this is the case.
- Search address where the new sample has to be sorted into the list. If the actual median has been overwritten, check if the new sample is smaller or larger than or equal to the actual median. Adjust median pointer accordingly.
- Sort the new sample into the list. Three cases have to be distinguished. Sorting into the left or right edge of the list and sorting somewhere into the middle of the list.
- Update of the median value (pointer) depending on the position of the newly inserted sample.

The first task in a new cycle of the median filter is the insertion of the new sample `x_new` into the ringbuffer by overwriting the oldest sample in the buffer. To keep the oldest sample it is stored in `x_old` before being overwritten. Furthermore the pointers to the smallest and largest value `rS` and `rL` are stored. `x{r0}` now holds the current input sample.

```
a) x_old := x{r0};

b) x{r0} := x_new;

c) rS_old := rS; rL_old := rL;
```

If the new sample overwrites the smallest or largest value in the list, the pointers `rS` and `rL` are updated to `{r0}.next` and `{r0}.prev` respectively. This means that the pointer `rS` is changed to point to the second smallest value (d1) and `rL` is changed to point to the second largest value (d2). The check if the new input data will be the new largest or smallest value will be carried out in the next step.
If the new value neither overwrites the smallest nor the largest value in the list (d3), the pointers of the nodes next to `{r0}` are changed to point to each other, meaning that the node `{r0}` is temporarily deleted from the list.

```
d) (1) if r0 = rS then
         rS := {r0}.next;
   (2) if r0 = rL then
         rL := {r0}.prev;
   (3) else
         {{r0}.prev}.next := {r0}.next;
         {{r0}.next}.prev := {r0}.prev;
       end;
```

Exemplary for a list operation, the deletion of an element (d3) is illustrated in fig. 3.

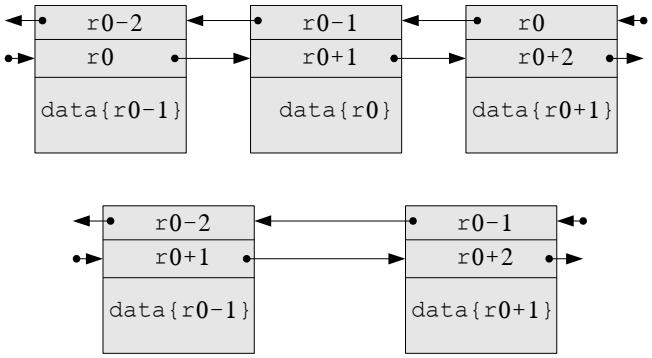

Figure 3: *Deletion of a node (d3).*

The next step in the algorithm is the first part of the median tracking, where it is tested if the median was overwritten by the new input value. The basis for this is the identification of the position the new sample will take in the list. It is searched and the address stored in `ra` (e). The median tracking itself distinguishes between two cases. In the first case, the new value is smaller than the actual median value, in the second case it is equal to or larger than the actual median value. If the new value is smaller than the actual median value, the pointer `rM` is replaced with the pointer to the next smaller value (f1). One can imagine this operation as if the values in a sorted list would have been shifted to the right and the pointer stayed fixed. If the new input value is equal to or larger than the actual median, `rM` is replaced with the pointer to the next value (f2). One can imagine this operation as a left-shift of the values.

```
e) Search ra with (x{{ra}.prev} <= x{r0}
                   and x{ra} > x{r0})
f) Part 1 of the median tracking, 2 cases:
   if rM = r0 then
   (1) if x{r0} < x{rM} then
         rM := {rM}.prev; end; end;
   (2) if x(r0) >= x{rM} then
```

```
        rM := {rM}.next; end; end;
```

After the cases where the special values smallest, largest or median value have been overwritten are now handled, the sorting of the new value into the list can be processed. If the new sample replaces the smallest or largest value in the list, its insertion into the list is quite simple, since the ordering of the list does not change. Just one of the border elements is affected by the operation. So the sorting is checking three cases. The first case is given when `x{r0}` is equal to or smaller than the smallest value and `rS_old != r0` or when `x{r0}` is smaller than the second smallest value and `rS_old = r0`. If the first set of criteria is met, a new node has to be appended at the left or the smallest value has to be replaced with the new one. In both cases the `next` field of `{r0}` has to be set to point to `{rS}` and the `prev` field of `{rS}` has to be set to point to `{r0}`. Then `rS` is updated with `r0` which represents the actual smallest value (g1).
The second case occurs if `x{r0}` is larger than or equal to the largest value or if it is larger than the second largest value with corresponding conditions for `rL_old` as in the first case. Here a new node has to be appended at the right or the largest value has to be replaced with the new one. Operations involved are analogue to the operations for the left side of the list (g2).
Finally the third part (g3) covers the case where none of the above conditions are fulfilled, i.e. the new value has to be sorted in somewhere in the middle of the list. Here the node `{ra}` found in step (e) pointing to the first node that contains data `x{ra}` bigger than `x{r0}` and its neighbour `{{ra}.prev}` are updated for the insertion of `{r0}`.

```
g) Sort in new element, 3 cases:
   (1) if (x{r0} <= x{rS_old}
          and rS_old != r0)
       or (x{r0} < x{{rS,old}.next}
          and rS_old = r0) then
              {r0}.next := rS;
              {rS}.prev := r0;
              rS := r0;
   (2) if (x{r0} >= x{rL_old}
          and rL_old != r0)
       or (x{r0} > = x{{rL_old}.next}
          and rL_old = r0) then
              {n}.prev := rL;
              {rL}.next := r0;
              rL := r0;
   (3) else
          {{ra}.prev}.next : =r0;
          {r0}.prev := {ra}.prev;
          {r0}.next := ra;
          {ra}.prev := r0;
       end;
```

The final step is the update of the median value after the new input value has been sorted into the list. In the first step of the median tracking it has been checked if the old median value has been overwritten. Now we have to determine the new median value after the new sample has been sorted into the list. The most simple case is when the newly inserted value by chance has been written to the place where it would have been sorted in. Then we don't have to update anything and the algorithm is terminated. Since only one new value has been inserted into the list, the median pointer `rM` has to be changed only by one, depending if the new inserted value is smaller or larger than the value `x{rM}`. Depending on the (oldest) value that has been overwritten, the median pointer is then updated and the new median `x{rM}` value can be sent to the output.

```
h) Part 2 of the median tracking, 2 cases:
   (1) if x{r0} < x{rM} then
          if x_old > x{rM} then
             rM := {rM}.prev; end;
       end;
   (2) if x{r0} >= x{rM} then
          if x_old <= x{rM} then
             rM := {rM}.next; end;
       end;
```

3.4. Performance

The described median filter has been evaluated on the DSP56374 running at an internal clock of 147.456 MHz with an audio sampling frequency of 48 kHz resulting in 3072 cycles that can be executed per sample. The performance has been estimated by tracking of the longest path in the algorithm and addition of the cycles the DSP needs to execute this path [12].
First the median filter algorithm without median tracking is considered. In our implementation the algorithm has an overhead with a fixed amount of 231 cycles independent of the length of the median filter and the longest path consumes 19 cycles per list element. Thus its performance allows to run a stereo median filter of length $N = 149$ in realtime. Without median tracking, the new median has to be searched in the list every time a new sample has been inserted, which increases the computation time needed per list element.
When the median tracking with the additional pointer `rM` to the median value is added, the computational load needed per list element reduces significantly to 9 cycles. However the overhead increases from 231 to 410 cycles. Using median tracking a stereo median filter with a length up to $N = 295$ can be realized. Due to the larger overhead median tracking is useful only for bigger values of N.

4. APPLICATION EXAMPLE

The median filter in its described implementation has been used for the removal of click and crackle sounds from old vinyl records in real-time [11]. Target system is the Freescale DSP56374 installed on a self-developed DSP-Board [12], programmed directly in assembly language.
The structure of the implemented system is shown in fig. 4 As

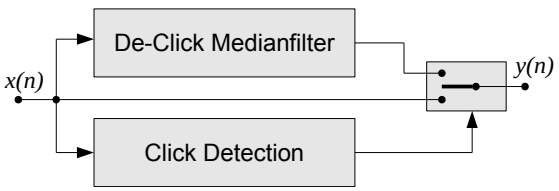

Figure 4: *Structure of the crackle removal system.*

can be seen, the system has two audio data paths, one direct path and one path through the median filter. A click detection algorithm in the sidechain switches the output signal $y(n)$ between these two signal paths. The separate path for the click detection is

necessary, since the application of the median filter to the whole music signal would degrade audio quality too much [13]. Practically the switching between the direct and the median-filter path is done using crossfading to avoid the introduction of switching artefacts. Several approaches for the detection algorithm have been evaluated in [11]. Among the algorithms are linear prediction, the application of a short median filter, a statistical approach and the usage of fixed and dynamic thresholds. The short median filter and linear prediction were both implemented on the DSP for click detection, where the median filter worked well with a length of $N = 5$ as a click detector and a static threshold. With this short filter length a direct implementation using sorting was used. For short filter lengths sorting is better or similar in performance compared to the use of a doubly linked list since there is no overhead for the handling of the pointers.
Clicks on vinyl recordings can have several different characteristics mainly regarding amplitude and duration. Furthermore it is not trivial to distinguish between impulsive sounds and unwanted click noise. Hence parameter setting for the click detection and also the click removal is a very important and extensive task. The use of a relatively simple algorithm like a median filter as a click detector is in most cases not able to detect all kinds of clicks properly without reacting to sharp musical transients. Besides its detection it must be considered how a click is replaced with the median filtered signal in a way that this process does not generate any additional sound degradation. This operation includes the determination of the click's parameters as e.g. its length. It is clear that the choice and parametrization of the click-detection algorithm mainly determines the performance of the whole click removal system.

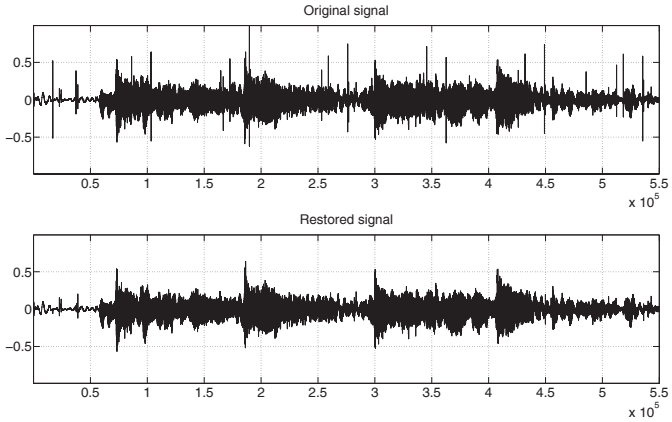

Figure 5: *Click removal from a piece of music with a median filter. Top: Original signal. Bottom: Restored signal.*

Another crucial parameter is the length of the de-click median filter. It determines besides the click detection algorithm the performance of the system. If the filter length is chosen to be very short, the application of the median filter may not be able to remove a detected click sound, if the sound persists for a relatively long time. On the other hand if the filter is chosen to be too long, it will not be able to restore the original waveform by filtering out short transients but rather smooth it too much. Generally it can be said that the length of the median filter must be at least twice as big as the length of the click to ensure that the disturbed samples do not influence the median value. In other words the median filter filters out any transients with durations less than half the length of the filter. Apparently the choice of the filter length has to be a compromise between the ability to remove clicks and the preservation of sound quality. In our application good results have been obtained with filter lengths of about 25 samples for clicks of short to medium duration and crackle noise.

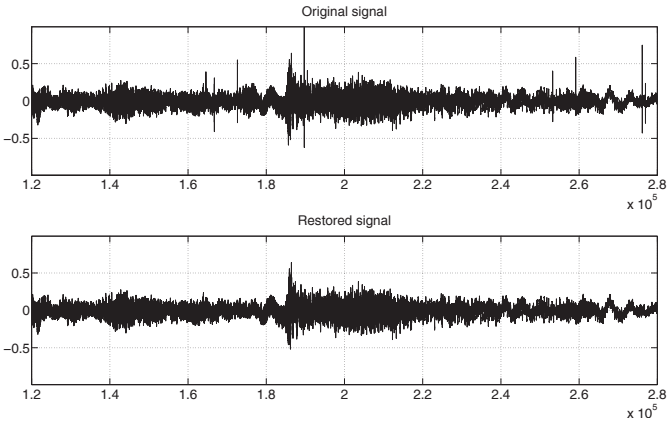

Figure 6: *Detail of fig.5. Click removal in the presence of transients. Top: Original signal. Bottom: Restored signal.*

The results of the described system are shown in figs. 5 and 6. The system is able to remove most of the clicks from the music signal. However at the beginning of the signal, not all clicks are removed, which is caused by the parameter setting and the relatively low complexity of the click detection algorithm in the sidechain. However the performance in distinguishing between musical transients works satisfactory as can be seen in 6.

5. CONCLUSIONS

A median filter can be efficiently implemented as a doubly linked list on a DSP in assembly language by mapping the structure of the list onto the DSP architecture. In combination with a click detection algorithm it can be successfully applied as a real time click and crackle removal tool for vinyl playback. The click removal performance is good enough to remove short clicks and crackle noise. However the determination of the parameters for click detection and removal is an extensive task. In a real-time system there is no possibility to analyse the noise characteristics in a first pass to determine the parameters as can be done in an offline application. For a real-time implementation, an adaptive parameter control would be a possibility to further improve the performance. Since the de-click median filter runs very efficiently, sufficient resources are available to implement a more complex click-detection algorithm.

6. ACKNOWLEDGMENTS

Many thanks to the staff and students at the department of digital signal processing that were involved in the development and implementation of the median filter.

7. REFERENCES

[1] W. Sethares, A. Milne, S. Tiedje, A. Prechtl, and J. Plamondon, "Spectral tools for dynamic tonality and audio morphing," *Computer Music Journal*, vol. 33, pp. 71–84, 2009.

[2] D. Fitzgerald and M. Gainza, "Single channel vocal separation using median filtering and factorisation techniques," *ISAST Transactions on Electronic and Signal Processing*, vol. No.1, Vol.4, pp. 62–73, 2010.

[3] T. Kasparis and J. Lane, "Adaptive scratch noise filtering," *IEEE Journal on Consumer Electronics*, vol. Vol. 39, No. 4, pp. 917–922, 1993.

[4] M. Kahrs and K. Brandenburg, eds., *Applications of Digital Signal Processing to Audio and Acoustics*. Kluwer, 1998.

[5] J. A. Bezemer, "Signal Processing in GramoFile," tech. rep., TU Delft, 1998.

[6] R. Holopainen, "Nonlinear Filters," tech. rep., Norwegian Network for Technology, Acoustics and Music.

[7] A. Nieminen, "Suppression and detection of impulse type interference using adaptive median hybrid filters," in *Acoustics, Speech, and Signal Processing, IEEE International Conference on ICASSP '87*, 1987.

[8] A. Nieminen, P. Heinonen, and Y. Neuvo, "Median-type filters with adaptive substructures," in *IEEE Transactions on Circuits and Systems*, 1987.

[9] M. B. Jones, "Real-time speech enhancement using median filters," in *Proceedings of SST 1994 Volume 1*, vol. 1, pp. 406–410, 1994.

[10] Freescale, www.freescale.com, *DSP56300 Family Manual*.

[11] S. Scholl, "Robuste Unterdrückung von Störsignalen für die Schallplattenwiedergabe," Master's thesis, Technical University Kaiserslautern, 2009.

[12] S. Schorz, "Entwicklung und Aufbau eines DSP-Boards für ein digitales Medianfilter," Master's thesis, Technical University Kaiserslautern, 2008.

[13] C. Chandra, M. S. Moore, and S. K. Mitra, "An efficient method for the removal of impulsive noise from speech and ausio signals," in *ISCAS '98 - Proceedings of the 1998 IEEE International Symposium on Circuits and Systems*, 1998.