# Digital Signal Processing for Smart Microphones

Authors

Heeb Thierry, SUPSI

Andrew Stanford-Jason, XMOS

Leidi Tiziano, SUPSI

Document Version: 1.0

SUPSI

University of Applied Sciences of Southern Switzerland

# Abstract

This whitepaper introduces the concept of smart microphone as an evolution of the classical microphone driven by the emergence of natural speech interfaces. From pure sound pressure to electrical signal converters, voice acquisition devices have evolved into complex systems featuring multiple capture channels coupled with digital signal processing capabilities. In this whitepaper, we focus on beamforming noise cancelling smart microphone applications.

Following a reminder of the fundamentals of array processing and adaptive filtering theory, a review of commonly used algorithms for beamforming, voice activity detection and noise/echo cancellation is provided. These algorithms can be combined to form complete solutions for beamforming noise calling smart microphone for a wide variety of applications. Possible implementations on the XMOS xCORE-200 architecture are subsequently analyzed in terms of computational complexity, memory usage and cores distribution. Our investigations show that these solutions based on the well-known HiRes Delay and Sum example should require only modest overhead, even if run at sampling rates as high as 48kHz. This demonstrates the practical usability of the xCORE-200 architecture for smart microphone applications.

# Table of contents

# From the microphone to smart microphone

Microphones have been an integral part of the audio ecosystem for a long time. Transforming acoustical information into an electrical signal, they are the front end of any audio processing system. For years they have been considered as pure pressure to voltage domain conversion devices but this is about to change drastically.

There are two main driving factors behind this change. The first one is the emergence of voice controlled devices such as TV sets or personal digital assistants. These not only require the user's voice to be captured but also to be processed to deliver an adequate response. Such processing is not trivial and includes the two major steps of voice extraction and speech analysis. Whilst voice extraction is mainly concerned with detection and separation of voice activity from background noise, speech analysis addresses the semantic processing side of the information. The second driving factor is the growing trend towards connected devices and ubiquitous data access. This calls for a move from a device-centric view of information processing systems to a distributed architecture. In such a system, part of the processing is done at the device level whereas the rest is done on remote servers in the cloud. In the case of a personal digital assistant, voice extraction would typically be done at device level and speech analysis would be carried out remotely.

Voice extraction usually requires multiple input sources (for instance for beamforming or noise cancellation) and hence microphones can no longer be considered individually but arrays of them must be taken into account. Their information must be aggregated and processed to provide meaningful information for subsequent stages of the system. The simple microphone is thus evolving into a multi-source system with embedded Digital Signal Processing (DSP) capabilities, running sophisticated algorithms in real-time. The term *smart microphone* is used to designate such a system providing voice extraction capabilities.

Using arrays of microphones clearly puts a challenge on system cost and the general trend has been to use PDM (Pulse Density Modulation) output MEMS (MicroElectroMechanical Systems) microphones. Whilst such devices are cheap and small, they require conversion of the high-speed 1bit data stream to low rate Pulse Coded Modulation to make it usable for subsequent processing stages. In addition, high out of band noise and non-flat frequency response at audio spectrum edges call for a high dynamic range conversion process. These are non-trivial tasks in terms of DSP resources and the interested reader is pointed to Ref [1] for a more in-depth coverage of the subject and its implementation on the xCORE-200 architecture.

Once transformed to PCM, the output signals of the different microphones are then processed to provide voice extraction. Many variations are possible but a smart microphone application typically includes following algorithms which will be reviewed in this whitepaper:

- Beamforming (BF): BF is central to the voice extraction process. It allows for focusing on the speaker's voice by steering the microphones array's directivity pattern towards the sound source.
- Voice Activity Detection (VAD): VAD, as its name suggests, provides detection of actual voice activity. This allows for triggering of subsequent processing only when needed or possible. VAD is also used to drive the operating mode of noise or echo cancellers.
- Noise/Echo Cancellation (NC/EC): NC is generally used to remove ambient noise from the captured voice signal to enhance speech intelligibility. EC is used to remove echoes that can occur in looped bi-directional systems such as speaker phones where the remote user's voice can be captured by the local microphone and fed back to the remote user.

Key Word Detection (KWD) is also sometimes integrated into smart microphone applications to provide recognition of certain voice command patterns. In the case of a personal digital assistant, KWD can for instance be used to trigger the start of an interaction by recognizing that the user has pronounced the interaction initialization word (typically the assistant's "name"). KWD is a vast and complex subject that cannot be covered within the limits of this whitepaper and hence will mostly be ignored in the rest of this document.

## Typical smart microphone application

In this section, we take a closer look at the requirements for a beamforming noise cancelling smart microphone targeted at voice applications. Such a smart microphone can be used in a variety of devices ranging from Smart TVs or conferring systems to network connected personal digital assistants. Following diagram shows the data flow of such a system (voice input path only):

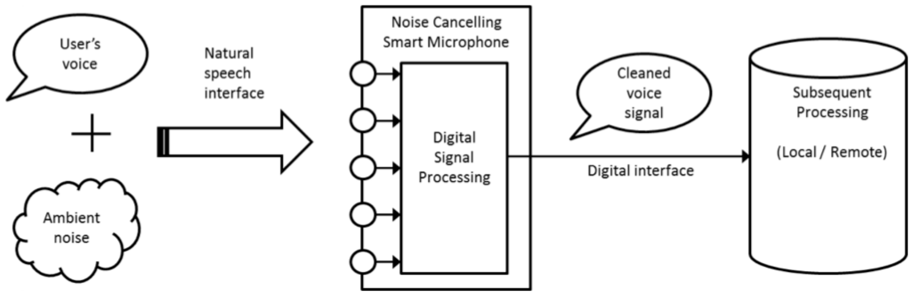

**Figure 1: data flow of noise cancelling smart microphone system**

The smart microphone picks-up the user's voice corrupted by ambient noise through an array of microphones. By using different algorithms, it processes the corrupted speech to provide a cleaned voice signal for subsequent processing such as semantic analysis. Subsequent processing can be run locally but is mostly off-loaded to remote servers. This is typically the case in a network connected digital assistant where request analysis and response generation are done in the cloud.

The digital signal processing path of a beamforming noise-cancelling microphone is shown in Figure 2.

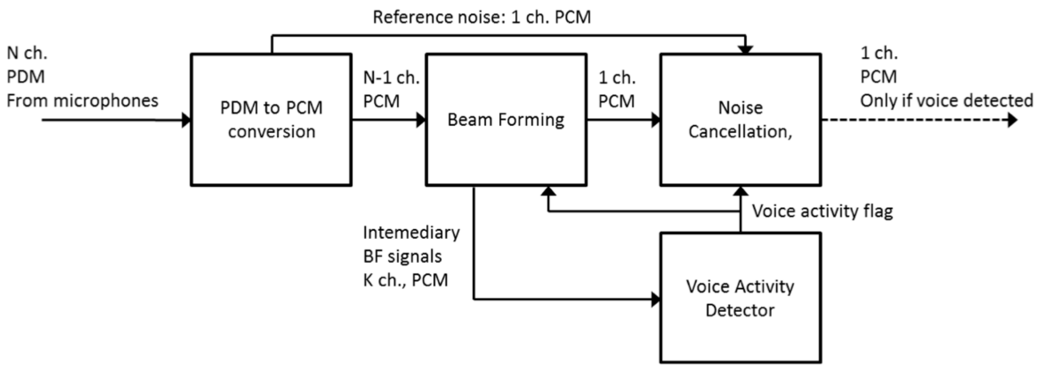

**Figure 2: digital signal processing path in beamforming noise cancelling smart microphone**

It includes the conversion process required to interface to the PDM microphones and the processing functions for Beamforming, Voice Activity Detection and Noise cancellation. In this whitepaper we study the corresponding DSP algorithms and their implementation on XMOS MCUs. We start by giving a reminder of array processing and adaptive filtering theory before presenting different (simple) algorithms for each function. Although focus is set on noise cancelling smart microphones, echo cancellation is also briefly presented. Finally we review possible implementations on the xCORE-200 architecture based on examples.

# Theory reminder

Proper understanding of the DSP algorithms involved in the voice extraction processing chain of smart microphone applications requires some knowledge of array processing (for beamforming) and adaptive filtering (for noise/echo cancellation). This chapter provides a reminder of some key theoretical concepts related to these topics.

## Fundamentals of array processing

Array processing can be defined as the processing of signals generated by a finite number of sensors in response to propagating waves. As such, sensor arrays can be considered as a discretized version of continuous apertures. Within the context of this whitepaper, the sensors are considered to be PDM generating MEMS microphones responding to acoustical waves.

Under the hypothesis of homogeneity and zero viscosity, *acoustical waves* in the air obey the following partial differential equation, known as the wave equation:

$$L(x(r,t)) = \frac{1}{c^2}\frac{\delta^2}{\delta t^2}x(t,r)$$

where $L(.)$ is the Laplacian operator, $r$ the spatial coordinates and $c$ the wave's propagation speed (about 343 m/s for a sound wave in the air at 20 °C). The solution for the case of planar waves for a single tone results in:

$$x(r,t) = A\,e^{j(\omega t - k\cdot r)}$$

where $A$ is the amplitude of the wave, $\omega = 2\,\pi f$ is the frequency in rad/s and $k$ is the wavenumber vector representing the direction of wave propagation. In the case of a 2D, respectively 3D wave, $k$ is given by:

$$k = \frac{2\pi}{\lambda}(\cos\phi, \sin\phi)\ \ \text{[2D case]} \qquad k = \frac{2\pi}{\lambda}(\sin\theta\cos\phi, \sin\theta\sin\phi, \cos\theta)\ \ \text{[3D case]}$$

where $\lambda = c/f$ is the wavelength and $\phi, \theta$ are the usual polar representation angles with $\phi$ being the azimuth angle (referred to the x-axis ) and $\theta$ being the elevation angle (referred to the z-axis).

Given the linearity of the wave equation, any superposition (i.e. linear combination) of such monochromatic waves will also be a solution. Time and space being linked by the wavenumber $k$ and signal information being preserved by wave propagation, we can conclude that a bandlimited signal can be fully reconstructed over time and space from its spatial samples at a given time. This is a key element of array processing.

*Apertures* are defined as spatial regions that receive or emit propagating waves. In our context, the aperture is represented by the location of the microphones. We define the aperture function (or sensitivity function) $A(\omega, r)$ as the transfer function linking the incoming wave $X(\omega, r)$ to the response along the aperture $X_R(\omega, r)$. This can be modelled as a spatial position depending linear filter, resulting in following frequency domain expression:

$$X_R(\omega, r) = X(\omega, r)A(\omega, r)$$

The aperture response $X_R(\omega, r)$ is naturally direction-dependent due to variations of the amount of signal received by the aperture with varying incidence angle. This is illustrated in the figure below:

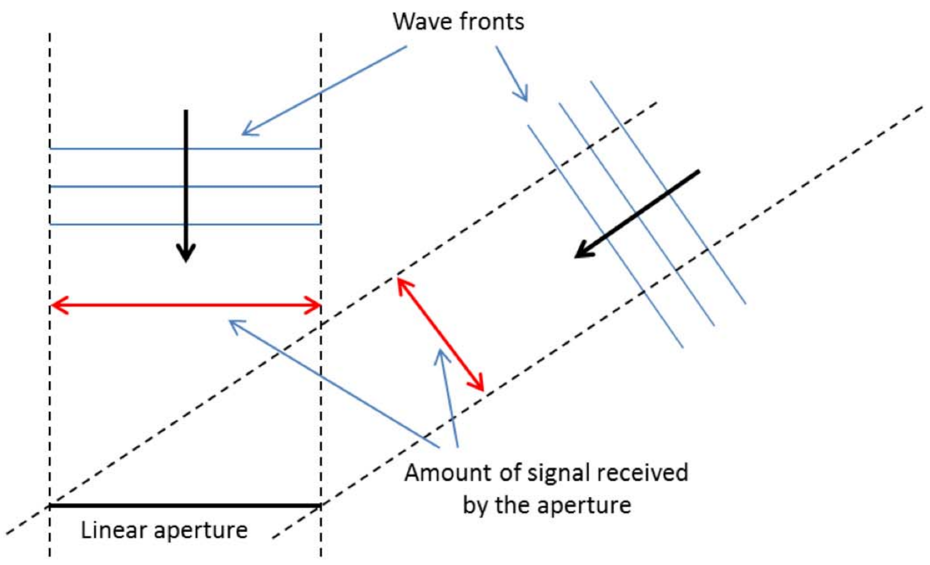

**Figure 3: Amount of signal received by aperture**

The aperture response as a function of frequency and incidence angle (or direction of arrival) is called the aperture's directivity pattern or beam pattern. Assuming a far-field situation, the directivity pattern can be expressed as:

$$D_R(\omega, \alpha) = \int_{-\infty}^{+\infty} A(\omega, r) e^{j2\pi\alpha \cdot r} \mathrm{d}r$$

where $\alpha$ is the direction vector of the wave. In the case of a 2D wave, we have $\alpha = k/2\pi = 1/\lambda\,(\cos\phi, \sin\phi)$, clearly showing the frequency dependency of the above expression as $\lambda = c/f$.

Considering the case of a linear aperture of length $L$ centered along the x-axis with uniform aperture function $A(\omega,\mathrm{x}) = \mathrm{rect}(\mathrm{x}/L)$ and exposed to far-field sources (that is if $|r| > 2L^2/\lambda$), the expression for the directivity pattern reduces to:

$$D_R(\omega, \alpha) = L\,\mathrm{sinc}(\alpha L)$$

Following picture shows the magnitude of the resulting directivity pattern where the characteristic shape of the underlying sinc function can be observed.

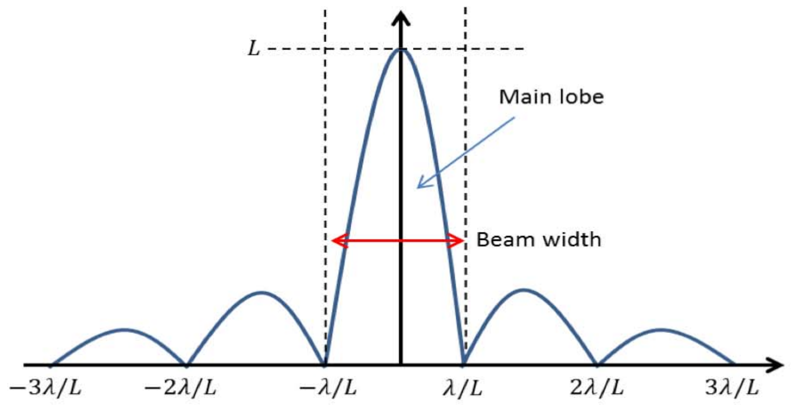

**Figure 4: directivity pattern (magnitude) of linear aperture with far-field source**

One can see that the directivity pattern exhibits zeroes located at points $m\lambda/L$ where $m$ is an integer. The central part located between $-\lambda/L$ and $\lambda/L$ is called the main lobe and its spread is called the beam width. In the case of a linear aperture, the beam width is given by $2\lambda/L$ which is clearly inversely proportional to frequency. Hence, for a given aperture length, beam width will be larger for low frequencies than for high frequencies. This frequency dependency plays an important role in the performance of beamforming algorithms as will be seen in the corresponding section below.

Microphone (or more generally sensor) arrays can be considered as sampled versions of continuous apertures where excitation of the aperture happens only at a finite number of discrete points in space. Considering each microphone as a continuous aperture (of infinitesimal size), the overall response can be expressed as the sum of the responses of each element in the array thanks to the superposition principle, resulting in an approximation of the equivalent continuous aperture.

As an illustration, let us consider a linear array of $N$ identical microphones having unity response of infinitesimal size and centered on the x-axis. The aperture function of the array can be expressed as ($N$ assumed to be odd for summation bounds notation simplicity):

$$A(\omega, r) = \sum_{n=-(N-1)/2}^{n=(N-1)/2} h_n(\omega)\delta(r - x_n)$$

where $h_n(\omega)$ is the complex weight of array element $n$, $x_n$ is the spatial location of element $n$ (located on the x-axis) and $\delta$ is the standard Dirac pulse. If we further consider that the array elements are all spaced uniformly with distance $d$, we can write the directivity pattern as:

$$D(\omega, \alpha) = \sum_{n=-(N-1)/2}^{n=(N-1)/2} h_n(\omega) e^{j2\pi\alpha\cdot nd}$$

Considering only diffusion in the horizontal plane, this reduces to:

$$D(\omega, \phi) = \sum_{n=-(N-1)/2}^{n=(N-1)/2} h_n(\omega) e^{j\frac{2\pi}{\lambda}nd\cos\phi}$$

Following figure shows the variations of the magnitude of the directivity pattern for different frequencies. An array with $N = 7$ and $d = 0.028$m is considered. $h_n(\omega)$ are normalized to provide unity gain for the main lobe.

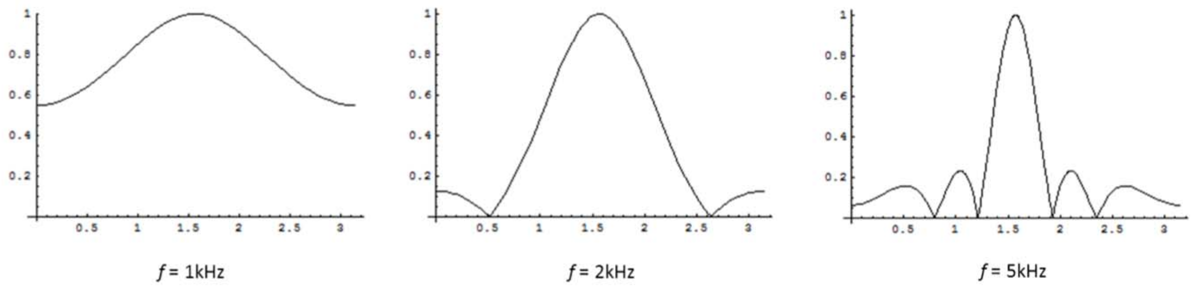

**Figure 5: Directivity patterns for linear array at different frequencies**

Closer examination of the directivity pattern equation reveals that:

- Side lobe level decreases with increasing spatial sampling density, i.e. side lobe level is linked to the number of elements $N$ in the array
- Beam width decreases with increasing array length, i.e. beam width is linked to the inter-element spacing $d$.

The last point to be considered in this review of array processing fundamentals is the concept of spatial aliasing. In the same way as time domain sampling of a signal is subject to the Nyquist criterion for perfect reconstruction, a similar criterion can be derived for spatial sampling. For linear arrays, the criterion is given by:

$$d < \frac{\lambda_{min}}{2}$$

This means that the distance between microphones in the array should be smaller than half the minimal wave length of the signal of interest. Violation of this criterion would result in spatial aliasing.

The interested reader is referred to Ref[2] for more details about array processing.

The general hypothesis of a *diffuse noise field* is assumed throughout this document unless otherwise specified. In a diffuse noise field, noise of equal energy is supposed to be propagating uniformly and simultaneously in all directions. This means that the different microphones in an array will receive lowly correlated noise signals but having approximately the same energy. Diffuse noise fields are good approximations for many real-world environments such as office or car background noise.

## Fundamentals of adaptive filtering

In this section, we give a short introduction to the theory of adaptive filtering and present some common filter coefficients update algorithms. We will limit ourselves to the case where the adaptive filter is of the FIR (Finite Impulse Response) type. The general block diagram of an adaptive filter is given in figure 6.

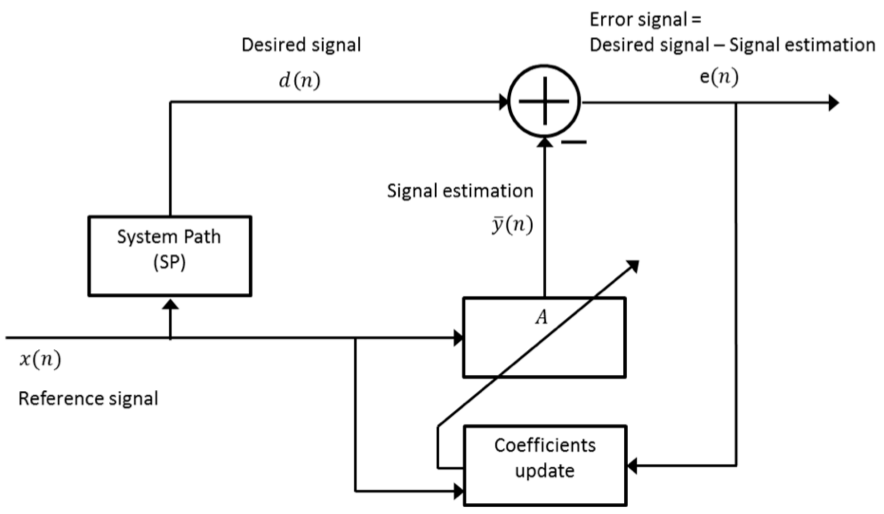

**Figure 6: Overview of adaptive filter**

A reference input signal $x(n)$ is passed through an unknown system symbolized by the System Path (SP) block in the diagram, resulting in a signal $d(n)$ called the desired signal. The goal of the adaptive filter is to compute the coefficients of filter $A$ so that it mimics the transfer function of the unknown System Path. In other words, the adaptive filter tries to identify the unknown System Path. The operating principle of an adaptive filter is the following:

- Starting from the reference signal $x(n)$, an estimation $\bar{y}(n)$ of the desired signal $d(n)$ is generated by applying filter $A$ to $x(n)$.
- An error signal $e(n)$ is computed as the difference between the desired signal and the estimation by $e(n) = d(n) - \bar{y}(n)$. If $\bar{y}(n)$ is a good estimate of the desired signal $d(n)$, the error signal $e(n)$ will tend to $e(n) = d(n) - \bar{y}(n) \cong 0$.

- The error signal $e(n)$ is then fed back to the coefficient update unit. This unit updates the coefficients of filter $A$ based on the optimization of a given criterion such as the mean square error.

The choices of optimization criterion and coefficients update algorithms have a significant impact in terms of system efficiency and are very context dependent. An algorithm with high update rate can lead to an unstable or oscillating system whereas too slow update rates can result in slow convergence and hence inefficient approximation of the System Path, especially under non-stationary conditions. We will now review two of the most common filter coefficients update algorithms used for adaptive filtering.

## LMS and related algorithms

The Least Mean Squares (LMS) algorithm is widely used in noise cancellation applications. It is a gradient based algorithm which adjusts the filter coefficients in the opposed direction of the instantaneous gradient of the output signal with respect to the coefficients vector. In other terms, it seeks to minimize the mean square error $MSE(n)$ due to corruption in the output signal (where $N$ is the considered signal length):

$$MSE(n) = \frac{1}{N}\sum_{k=0}^{k=N-1} e(n-k)^2 = \frac{1}{N}\sum_{k=0}^{k=N-1} (d(n-k) - \bar{y}(n-k))^2$$

Considering an adaptive FIR of length $P$, the LMS filter coefficients update algorithm can be computed as:

$$\bar{y}(n) = A(n) \cdot X(n)^T$$

$$e(n) = d(n) - \bar{y}(n)$$

$$A(n+1) = A(n) + \mu\, e(n)X(n)$$

where $A(n) = (a_0(n), a_1(n), \dots, a_{P-1}(n))$ is the adaptive filter coefficients vector (in other words its impulse response) at time $n$, $X(n) = (x(n), x(n-1), \dots, x(n-P+1))$ is the reference signal vector at time $n$, $(.)^T$ denotes the transposition operator and $\mu$ is the step size. It is recommended to select $\mu$ as:

$$0 < \mu < \frac{1}{P\, \|X(n)\|^2}$$

Where $P$ is the number of taps of the adaptive FIR filter and $\|X(n)\|^2$ represents the $L^2$ norm (or power) of the reference signal $X(n)$. The algorithm can be initialized with $A(0) = (a_0(0) = 0, a_1(0) = 0, \dots, a_{P-1}(0) = 0)$.

Note that for applications where the reference signal estimation $\bar{y}(n)$ undergoes a Forward Path (FP) before being combined with the desired signal into the error signal (typical case of a cancellation system where the error signal is picked-up by a microphone and the cancellation signal is emitted by a speaker), some adaptation to the LMS algorithm is required. Indeed the error signal will no longer consist in the reference signal passed through the System Path minus the adaptive filter output but the Forward Path needs to be taken into account resulting in $e(n) = d(n) - \bar{y}_{FP}(n)$ where $\bar{y}_{FP}(n)$ represents the adaptive filter output passed through the Forward Path. In order for the LMS algorithm to work in such a situation, the reference noise must be pre-filtered with a filter $F$ approximating the Forward Path's transfer function resulting in $x_F(n) = F\, x(n)$. Figure 7 shows the block diagram of an adaptive filter with Forward Path.

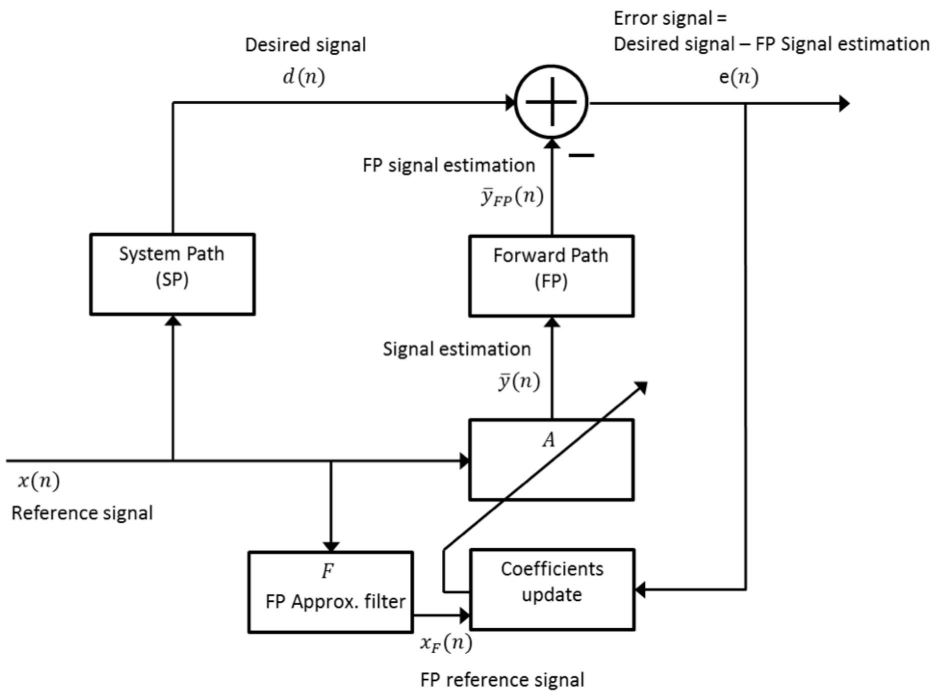

**Figure 7: Adaptive filter with forward path**

With these modifications, and using the usual matrix notation, the algorithm for coefficients update can be written as:

$$\bar{y}(n) = A(n) \cdot X(n)^T$$

$$e(n) = d(n) - \bar{y}_{FP}(n)$$

$$A(n+1) = A(n) + \mu\, e(n) X_F(n)$$

For such a system it is recommended (Ref[3]) to select $\varepsilon$ as:

$$0 < \mu < \frac{1}{(P + \Delta)\, \|X(n)\|^2}$$

where $P$ is the number of taps of the adaptive filter and $\Delta$ is the number of samples corresponding to the delay of the forward path. This variation of the LMS algorithm is called the **Filtered-x LMS (FxLMS)** algorithm.

Another commonly used version of the LMS algorithm is the **Normalized Least Mean Squares (NLMS)** algorithm. In the NLMS approach the step size is normalized with respect to the energy of the reference signal. The normalization allows maintaining both convergence speed and steady-state response, solving the instability issue of the LMS algorithm linked to variations in the power of the reference signal. The coefficients update step of the NLMS algorithms is defined by:

$$A(n+1) = A(n) + \mu \frac{1}{\|X(n)\|^2 + \delta}\, e(n) X(n)$$

where $\| \,.\, \|^2$ denotes the $L^2$norm and $\delta$ is an optional small positive integer to avoid division by 0 if $\|X(n)\|^2$ tends towards zero. Convergence requirements dictate that $0 < \mu < 2$ (Ref[4]). A typical values for $\mu$ is 0.02.

Finally, in the case of poorly conditioned reference signals which may cause the standard LMS algorithm to diverge due to bias accumulation in the coefficients of the adaptive filter, a leakage technique may be used resulting in the **Leaky Least Mean Squares (LLMS)** algorithm. A leakage term $\upsilon$ is introduced in the update step of the LMS approach as follows:

$$A(n+1) = \upsilon A(n) + \mu\, e(n)X(n)$$

For optimal operation, both $0 < \upsilon < 1$ and $1 - \upsilon < \mu$ should be verified.

Clearly, both the NLMS and LLMS aproaches can be extended to take a Forward Path into account in the same way as the LMS algorithm is extended into the FxLMS algorithm. This results in the **Filtered-x NLMS (FxNLMS)** and **Filtered-x LLMS (FxLLMS)** algorithms.

## RLS algorithm

The Recursive Least Squares (RLS) is a recursive coefficients update algorithm that computes the least squares solution for the adaptive coefficients $A(n) = \big(a_0(n), a_1(n), \ldots, a_{P-1}(n)\big)$ of a $P$ taps FIR filter. The RLS algorithm provides a computationally efficient method for finding the least square solution at each iteration $n$. The RLS algorithm generally exhibits better steady-state performance and faster convergence time than the (N)MLS algorithms but its computational complexity is higher.

The RLS coefficient update algorithm is given by:

$$K(n) = \frac{\lambda^{-1}Q(n-1)\cdot X(n)}{1 + \lambda^{-1}U(n)^T \cdot Q(n-1)\cdot X(n)}$$

$$\bar{y}(n) = A(n)\cdot X(n)^T$$

$$e(n) = d(n) - \bar{y}(n)$$

$$A(n+1) = A(n) + e(n)K(n)$$

$$Q(n) = \lambda^{-1}Q(n-1) - \lambda^{-1}K(n)\cdot X(n)^T \cdot Q(n-1)$$

where $K(n)$ is the gain factor (vector) and $Q(n)$ is the inverse of the reference signal autocorrelation matrix. $\lambda$ is the forgetting factor and must satisfy $0 < \lambda \leq 1$. Typical values used in practice are between 0.98 and 1. The algorithm can be initialized with $A(0) = (a_0(0) = 0, a_1(0) = 0, \ldots, a_{P-1}(0) = 0)$ and $Q(0) = \delta^{-1}I$, where $\delta$ is a small positive number and $I$ denotes the $P{\times}P$ identity matrix.

In the same way as the LMS algorithm can be adapted for situations where the reference estimation signal $\bar{y}(n)$ undergoes a Forward Path before being combined into the error signal $e(n)$ resulting in the FxLMS algorithm, the RLS approach can also be adapted to such a situation. The resulting algorithm is called the **Filtered-x RLS (FxRLS)** algorithm. Using the same notations as for the FxLMS approach, the FxRLS coefficient update steps can be expressed as:

$$K(n) = \frac{\lambda^{-1}Q(n-1)\cdot X_F(n)}{1 + \lambda^{-1}X_F(n)^T \cdot Q(n-1)\cdot X_F(n)}$$

$$\bar{y}(n) = A(n)\cdot X(n)^T$$

$$e(n) = d(n) - \bar{y}_{FP}(n)$$

$$A(n+1) = A(n) + e(n)K(n)$$

$$Q(n) = \lambda^{-1}Q(n-1) - \lambda^{-1}K(n)\cdot X_F(n)^T \cdot Q(n-1)$$

Again here the modifications are due to the fact that the Forward Path between the estimation signal emission and error signal capture needs to be compensated for correct adaptive filter coefficients computation.

# DSP algorithms for smart microphones

In this section, we review the main signal processing algorithms involved for voice extraction in smart microphones applications. As this is a very vast and fast evolving subject, we restrict ourselves to an overview of some possible solutions.

The typical voice extraction processing chain of a smart microphone application is shown in figure 2. Taking the PDM signals from a MEMS microphones array as input, it applies the steps of PDM to PCM conversion, Beamforming, Voice Activity Detection and Noise / Echo Cancellation to pre-process the signal for subsequent stages such as Key Word Detection and Semantic Analysis. In this section, we review the DSP algorithms involved in the different steps of the voice extraction process.

## PDM to PCM conversion

Due to their small size and low cost MEMS microphones are very well suited for the microphone arrays needed in smart microphone applications. Typically based on delta-sigma modulation, they produce a high speed (typically around 3MHz) Pulse Density modulated bit stream which needs to be converted to Pulse Coded Modulation (PCM) for subsequent processing. This conversion implies hefty low-pass filtering to remove high-frequency modulation noise combined with decimation to produce a PCM stream at the system's processing sampling rate (typically between 8kHz and 48kHz). Figure 8 illustrates the PDM to PCM conversion process.

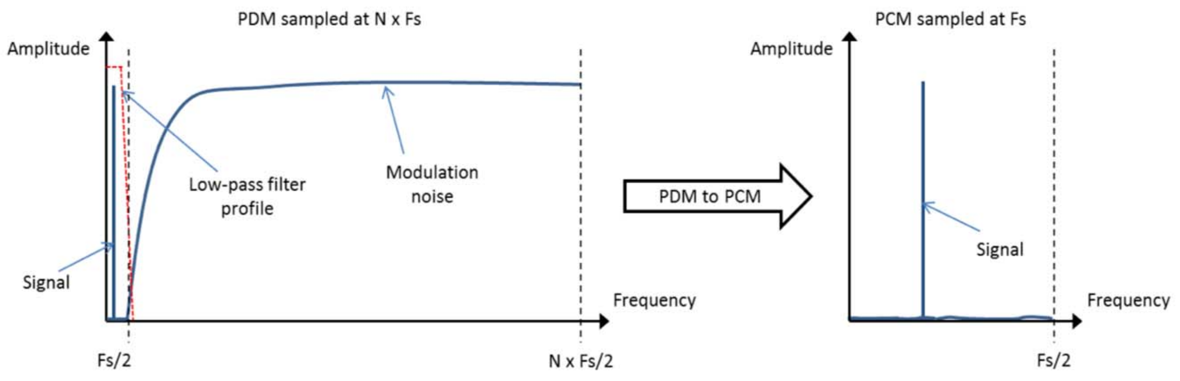

**Figure 8: PDM to PCM conversion process**

PDM to PCM conversion is usually implemented as cascade of decimation low-pass filters and is independent of array geometry. For instance, let's consider conversion from PDM sampled at 64 x Fs to PCM sampled at 1 x Fs, where Fs = 48kHz. This can be achieved using three stages of FIR (Finite Impulse Response) filtering as follows:

- 1st stage: 64 x Fs (3.072MHz) to 8x Fs (384kHz) decimator based on a 48 taps FIR.
- 2nd stage: 8 x Fs (384kHz) to 2 x Fs (96kHz) decimator based on a 16 taps FIR.
- 3rd stage: 2 x Fs (96kHz) to 1 x Fs (48kHz) decimator based on a 64 or 126 taps FIR (depending on application).

For more details, the interested reader is pointed to Ref[5] and Ref[1] which provide in-depth information about the theory and implementation of such a PDM to PCM converter on the xCORE-200 architecture.

## Beamforming (BF)

In this section, we review some of the fundamental beamforming algorithms, concentrating on the Delay-Sum, Filter-Sum and Sub-band approaches. Combination of beamforming with spatial noise cancellation in the Generalized Sidelobe Canceller beamformer architecture will be discussed in a separate section after the review of Noise / Echo Cancellation algorithms.

### Delay-Sum Beamforming

Delay-Sum beamforming is the simplest microphones array beamforming approach. We recall from the introductory section about Array Processing that the horizontal directivity pattern for a uniformly spaced array centered on the x-axis is given by:

$$D(\omega, \phi) = \sum_{n=-(N-1)/2}^{n=(N-1)/2} h_n(\omega) e^{j\frac{2\pi}{\lambda} nd \cos\phi}$$

The Delay-Sum designation comes from the fact that in this approach to beamforming, the time-domain input from each sensor is first delayed by an amount of $\tau_n$ seconds before being summed together. Considering an amplitude gain of $1/N$ for each weighting element $h_n(\omega)$ and setting $\tau_n$ to the time it takes for the wave to travel from the reference sensor (located at coordinates origin position) to sensor $n$ results in

$$\tau_n = \frac{nd \cos\phi'}{c} \qquad and \qquad h_n(\omega) = \frac{1}{N} e^{-j\frac{2\pi}{\lambda} nd \cos\phi'}$$

We can now express the output of the system by the sum of weighted array channels in the frequency domain:

$$y(\omega) = \frac{1}{N} \sum_{n=-(N-1)/2}^{n=(N-1)/2} x_n(\omega) e^{-j\frac{2\pi}{\lambda} nd \cos\phi'}$$

Or equivalently in the time domain:

$$y(t) = \frac{1}{N} \sum_{n=-(N-1)/2}^{n=(N-1)/2} x_n(t - \tau_n)$$

This is the well-known time-domain equation of the Delay-Sum beamforming algorithm, which is for instance used in XMOS' High-Resolution Delay-Sum example code (Ref[6]) and related application note (Ref[7]).

Figure 9 shows both the normalized non-steered ($\phi = \frac{Pi}{2}$) and steered ($\phi = \frac{Pi}{4}$) directivity pattern magnitude in the horizontal plane of a Delay-Sum beamforming system with $N = 7$ and $d = 0.028$m. A 5kHz tone is considered as input.

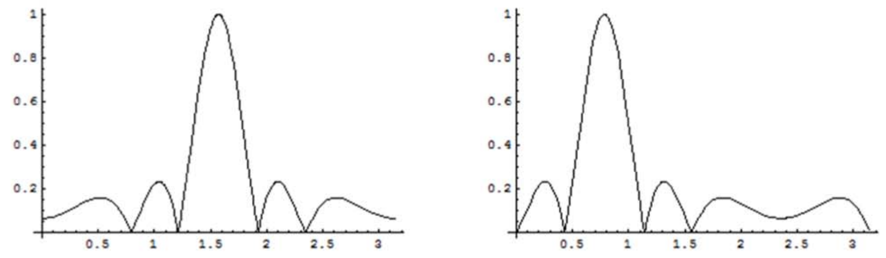

**Figure 9: non-steered and steered directivity pattern magnitude (Delay-Sum)**

## Filter-Sum Beamforming

In the Delay-Sum beamforming system described above, the magnitude responses of the weights $h_n(\omega)$ are identical across nodes and constant with respect to frequency. Allowing said magnitude responses to be frequency (and node) dependent results in the family of Filter-Sum beamformers. The output signal of a Filter-Sum beamforming system is given by:

$$y(\omega) = \sum_{n=1}^{N} h_n(\omega) x_n(\omega)$$

Using matrix notation with $x(\omega) = (x_1(\omega),\ x_2(\omega), \dots, x_n(\omega))^T$ and $h(\omega) = (h_1(\omega),\ h_2(\omega), \dots, h(\omega))^T$ where $(.)^T$ designates matrix transpose, this can be rewritten as:

$$y(\omega) = h(\omega)^T \cdot x(\omega)$$

Figure 10 shows a diagram of a typical Filter-Sum beamformer.

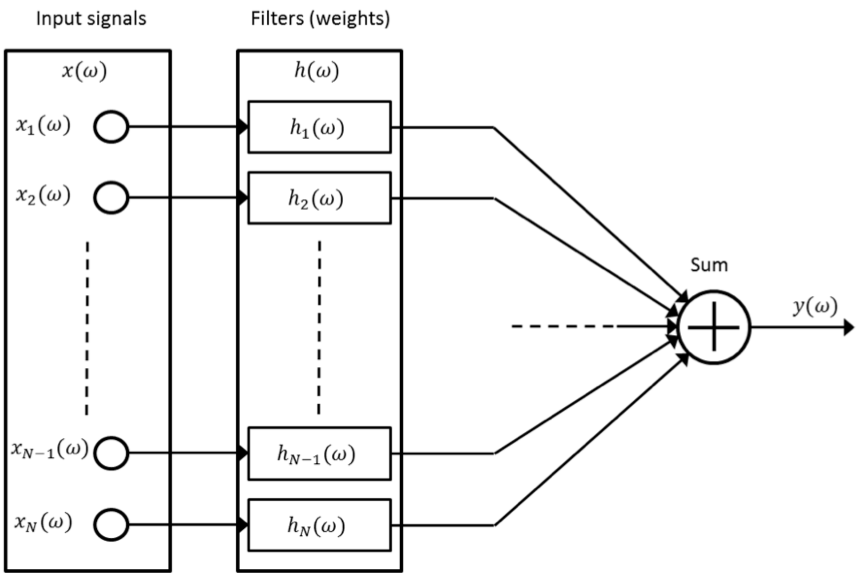

**Figure 10: Filter-Sum beamformer**

The main difference in terms of implementation with respect to the Delay-Sum approach is that the time domain delays are replaced by filters resulting in following time-domain expression:

$$y(t) = \sum_{n=1}^{N} h_n(t) * x_n(t)$$

where $h_n(t)$ is the impulse response of weighting filter $h_n(\omega)$ and $*$ represents the convolution operator.

## Sub-Array Beamforming

As discussed in previous sections, the overall array response is dependent on:

- The frequency of interest
- The number of sensors in the array
- The distance between sensors

This means that the array's response characteristics (such as beam width and sidelobe levels) can only be considered as constant for narrow band signals. We are however interested by speech signals (with main energy spread over a 5kHz band) which are to be considered as broad band signals. As such, a single linear array design is not capable of providing a frequency invariant beam pattern for a complete voice signal.

A simple method to improve performance for broad band signals is to implement the array as a set of sub-arrays, each of which being itself a uniformly spaced array acting on a defined frequency band. Maintaining constant beam width as frequency increases requires a smaller distance between array elements and keeping constant sidelobes level across frequency bands calls for a constant number of microphones in the different sub-arrays. The sub-arrays can be implemented in a nested structure where microphones are shared among different sub-arrays. A band-pass filter is applied to each sub-array to restrict its operation to a given frequency range. The outputs of all sub-arrays are then combined (i.e. summed) to form the output of the overall broad band array.

Using the matrix notation introduced for Filter-Sum beamformers, the output of sub-array $s$ can be written as:

$$y_s(\omega) = h_s(\omega)^T \cdot x(\omega)$$

where $h_s(\omega) = (h_{s,1}(\omega), h_{s,2}(\omega), \dots , h_{s,N}(\omega))$ is the weight filter vector for sub-array $s$. Note that $h_{s,i}(\omega)$ is set to 0 if microphone $i$ is not part of sub-array $s$.

The overall broad-band array output is then given by:

$$y(\omega) = \sum_{s=1}^{S} y_s(\omega)$$

where $S$ is the number of sub-arrays considered.

In terms of implementation, $h_s(\omega)$ typically represent band-pass filters with selectivity corresponding to the frequency band of interest of the corresponding sub-array. They can be implemented using standard FIR or IIR techniques. In practice, individual sub-arrays are commonly implemented as simple Delay-Sum beamformers followed by the corresponding band-pass filters. As such, the processing load for a sub-array beamformer can be computed as the sum of the loads for each sub-array; each of these consists in the load of the corresponding Delay-Sum beamformer and band-pass filter.

Figure 11 illustrates a typical 3 band sub-array beamforming system consisting in 3 nested arrays of 5, 5 and 3 microphones respectively, for a total of 7 microphones. The output of each sub-array is processed by a Delay-Sum beamformer before being passed through the respective band-pass filter. The outputs of the band-pass filters are then summed to produce the overall, broad-band beamformer output.

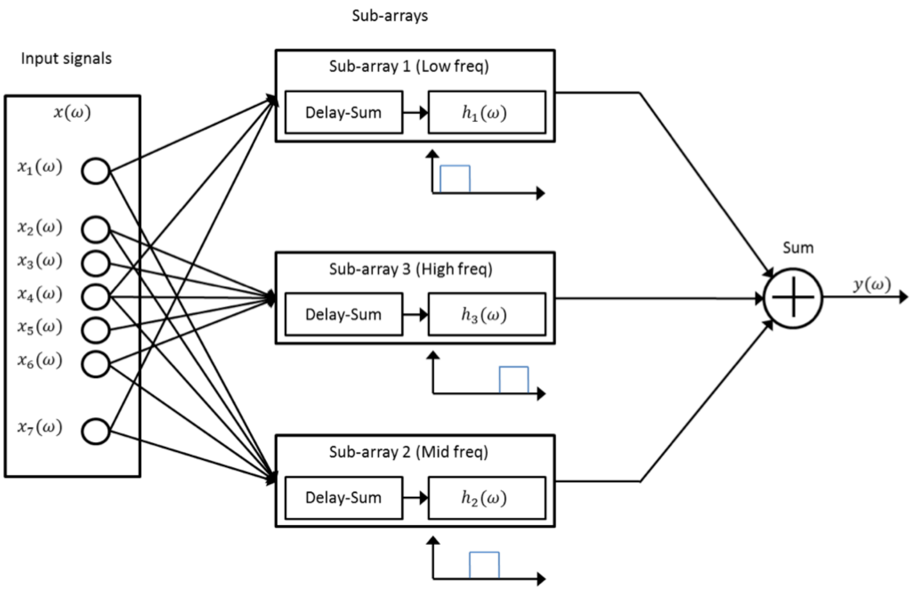

**Figure 11: Example of sub-array beamforming system**

## Voice Activity Detection (VAD)

The goal of Voice Activity Detection (VAD) is to identify the speech and non-speech segments in an audio signal. It is an integral part of many voice applications and is for instance used as a front end to noise cancellation systems for switching the adaptive filter coefficients update process on or off. In a typical smart microphone noise cancellation scheme, the adaptive filter coefficients are only updated during non-speech segments of the audio signal. This is required so that the adaptive process effectively tracks the background noise and not a superposition of speech and noise.

There are many different types of VAD algorithms but most follow the general structure illustrated in Figure 12.

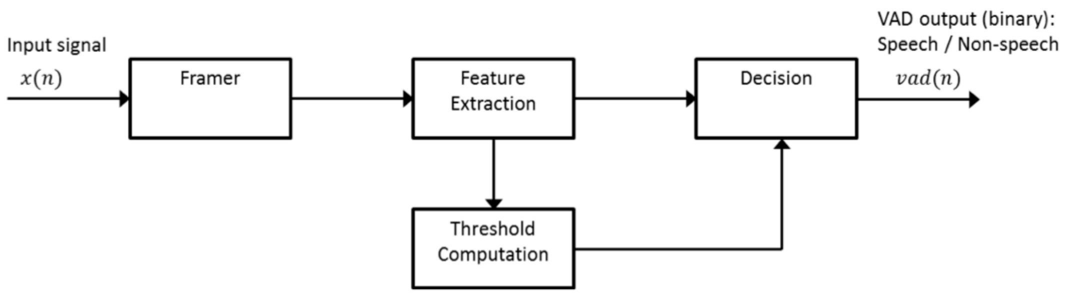

**Figure 12: General structure of a VAD algorithm**

The steps involved in the VAS process are the following:

- The input signal is first built into (overlapping) frames of typically 5 to 50ms by the Framer
- The Feature Extraction block is then used to extract one or more features from the received data frames
- Thresholds for the different features are computed by the Threshold Computation block
- Finally, based on the results of the comparison(s) between extracted feature(s) and threshold(s), frames are tagged as speech or non-speech, completing the VAD process.

An optional decision smoothing stage can be added to avoid fast toggling between speech and non-speech in situations where the decision is uncertain. This can simply be implemented using a hang-over counter providing a hysteresis on the decision process by making sure that detection is coherent over a certain number of frames before changing the actual output state of the VAD.

The type of features extracted and the computation of the threshold are key differentiator among VAD algorithms. In this review, we focus on some simple VAD methods suitable for real-time implementation. In particular, methods using advanced speech related statistics or pattern recognition are not covered in this paper and the interested reader is directed to Ref[8] and Ref[9] for some examples.

## Linear Energy-based Detector (LED) VAD

The Linear Energy-based Detector is one of the simplest forms of VAD. It is based on the assumption that the energy of the signal is higher when voice is present in addition to the background noise. The feature extraction is based on the computation of the current frame signal energy $E$ (or squared $L^2$ norm) defined as:

$$E = \sum_{i=0}^{N-1} x(i)^2$$

where $N$ is the frame length and $x(0), x(1), \ldots x(N-1)$ are the audio samples of the current frame. A frame is tagged as speech if following condition is verified:

$$E > k\, E_r$$

where $k > 1$ and $E_r$ is the energy threshold. Since background noise is usually non-stationary, $E_r$ is computed dynamically as:

$$E_r(k+1) = (1-p)E_r(k) + pE_{noise}$$

Where $E_r(k)$ designates the energy threshold of frame $k$, $p$ is a scalar with $0 < p < 1$ and $E_{noise}$ is the energy of the last frame having been tagged as non-speech.

Whilst being very simple to implement, the LED VAD's performance is far from optimal under varying background noise conditions or in high-noise environments. Hence more sophisticated VADs are generally used in practical situations.

## Adaptive Linear Energy-based Detector (ALED) VAD

One of the issues of the LED VAD is that the factor $p$ used in the update of the energy threshold $E_r$ is independent of the noise statistics and hence does not take noise variations into account. The ALED VAD is a first attempt to overcome this issue. The basic principle of ALED is to maintain a buffer of the last $m$ frames tagged as non-speech and to use the variance $\sigma$ of the energy in these frames to dynamically adapt the factor $p$ of the LED VAD. Let $\sigma_k$ define the value of $\sigma$ for frame $k$. A sudden change in background noise results in $\sigma_{k+1} > \sigma_k$ and $p$ is made dependent on the ratio $\sigma_{k+1}/\sigma_k$ with larger values of the ratio corresponding to larger values of $p$. This provides better tracking of varying noise conditions when compared to the LED VAD.

## Zero-Crossing Detector (ZCD) VAD

A major drawback of both LED and ALED VAD schemes is that they often fail to detected low-level unvoiced speech as it is fails to trigger the decision threshold. The Zero-Crossing Detector VAD is designed to address this problem. It is based on the fact that the number of zero-crossings of a voice signal lies in a pretty well defined range for a certain frame duration. This property allows defining the decision rule for voice activity in frame for ZCD VAD as:

$$R_{min} \leq N_{zc}(k) \leq R_{max}$$

Where $N_{zc}(k)$ is the number of zero-crossings in frame $k$ and $R_{min}$ and $R_{max}$ define the lower, respectively upper bound of the number of zero-crossings for a speech signal over the duration of the frame. For a 10ms frame, we would typically have $R_{min} = 5$ and $R_{max} = 15$ (Ref[10]).

A drawback of ZC VAD is that it often makes wrong decisions, as noisy frames may have a comparable number of zero crossings as a speech frame. Hence ZCD VAD is rarely used alone but can for instance be combined with LED or ALED systems to improve voice activity detection for low-level unvoiced speech signals.

## Linear Sub-band Energy Detector (LSED) VAD

The Linear Sub-band Energy Detector VAD basically consists in a multi-band extension of the standard LED VAD. The operating principle is that the input signal is divided into multiple frequency bands and the energy (or squared $L^2$ norm) $E^B$ in each frequency band $B$ is computed for each frame according to:

$$E^B = \sum_{i=0}^{N-1} x^B(i)^2$$

where $N$ is the number of samples in the frame and $x^B(i)$ are the samples of the input signal band-pass filtered for frequency band $B$. Conditions similar to the condition of LED VAD are defined for each frequency band as:

$$E^B > k\, {E_r}^B$$

where $k > 1$ and ${E_r}^B$ is the energy threshold for band $B$. These thresholds are dynamically updated as in a LED VAD by:

$${E_r}^B(k+1) = (1-p){E_r}^B(k) + p{E_{noise}}^B$$

where ${E_r}^B(k)$ designates the energy threshold of band $B$ for frame $k$, $p$ is a scalar with $0 < p < 1$ and ${E_{noise}}^B$ is the energy in band $B$ of the last frame having been tagged as non-speech. As voice signals are mostly located in the lower part of the spectrum, a decision rule for voice activity can be expressed as:

- $E^0 > k\, {E_r}^0$: Energy must be higher than the threshold in the lowest frequency band – and -
- $E^b > k\, {E_r}^b$: Energy must be higher than the threshold for a certain number of bands other than the lowest frequency band. For instance, if a 3 band system is considered, this second condition could be that at least one of the remaining bands present higher energy than the threshold.

Working with sub-bands matched to voice signals, the LSED VAD outperforms the standard LED approach. In addition, an adaptive version of LSED VAD can be constructed by applying the principles of ALED to each frequency band $B$.

## Combined VAD

In order to enhance the overall performance of VAD, different algorithms can be combined. As an example, we propose a VAD combining LSED and ZCD. The thresholds for each of the algorithms remain the same but the overall decision process for speech or non-speech frames is a combination of the decision processes of the different algorithms. Following pseudo code describes the resulting combined algorithm and decision process:

```
For each input frame
     compute LSED
     if LSED indicates 'Speech'
          return 'Speech'
     compute ZCD
     if ZCD indicates 'Non Speech'
          return 'Non Speech'
     return 'Speech'
```

## Noise and Echo Cancellation (NC and EC)

The goal of both noise and echo cancellation is to remove unwanted components from a corrupted signal to recover the original source. Cancellation can be implemented by passive means such as physical sound absorbers or by active generation of a cancelling signal. A major drawback of the passive approach is the size and cost of such installations, especially when cancellation of low frequency noise is considered as is the case for many acoustical environments such as cars, offices or conference rooms. Active cancellation is based on the superposition principle where an algorithmically generated anti-waveform is used to cancel out the undesired signal part. This principle is illustrated in Figure 13.

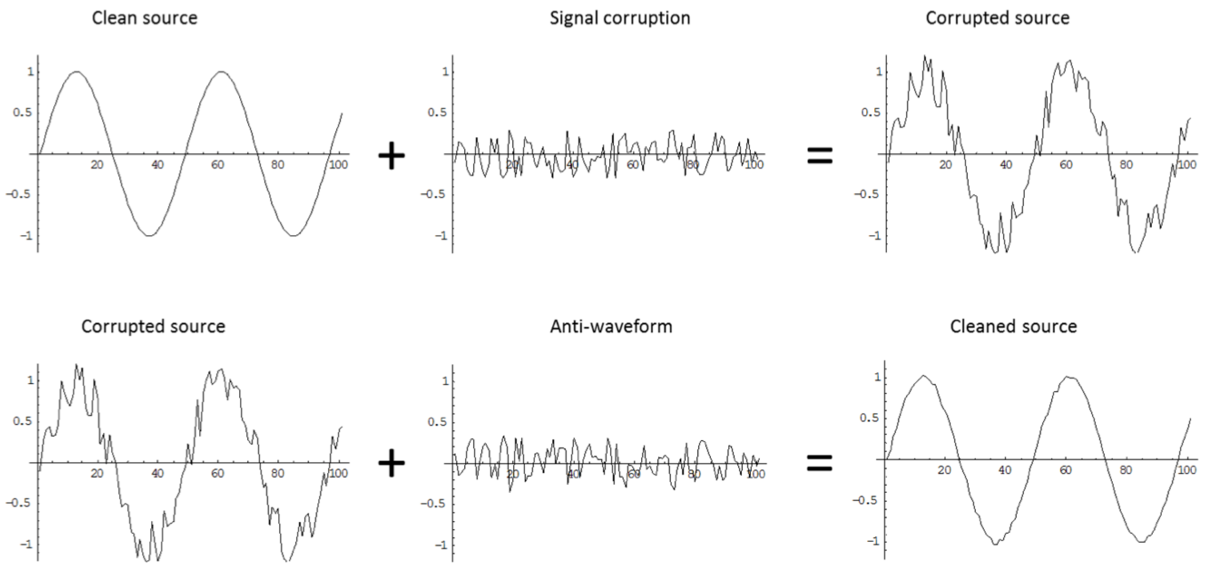

**Figure 13: Superposition principle for undesired signal cancellation**

Whilst the principle shown in Figure 13 may seem very simple, generation of the anti-waveform is non-trivial in practice due to non-stationary acoustical and noise conditions. This means that generation of the anti-waveform must be adaptive. Both noise and echo cancellers rely on adaptive generation to produce the cancelling signal and hence use similar DSP building blocks but their control signals and structure are slightly different.

### Noise cancellation

The principle of noise cancellation through anti-waveform generation has first been introduced by Paul Lueg in 1936 (Ref[11]) using a microphone and analog circuitry to drive a loudspeaker for generation of the cancelling waveform. Practical implementation of digital domain noise cancellation started in the early '80s with the advance in DSP technology and the introduction of adaptive algorithms

(Ref[12],Ref[13]). Figure 14 shows the general principle of a digital domain noise canceller for a smart microphone application.

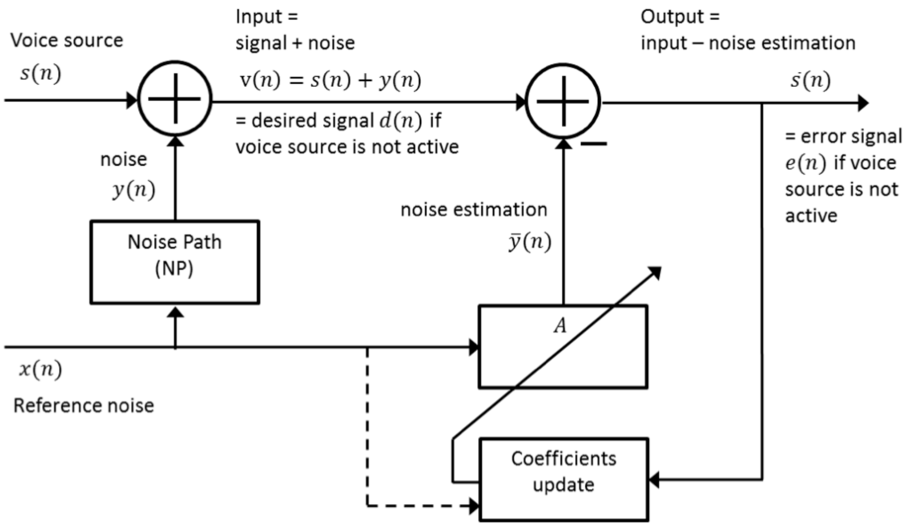

**Figure 14: Noise canceller block diagram**

A corrupted input signal $v(n) = s(n) + y(n)$ consisting in the original voice signal $s(n)$ corrupted by additive noise $y(t)$ is picked-up by a microphone and forms the primary input to the system. The additive noise $y(n)$ captured by the microphone is the result of a reference noise source $x(n)$ passed through an unknown Noise Path corresponding to the transfer function between the reference noise source and the microphone. The reference noise source $x(n)$ is passed through an adaptive filter $A$ to generate an estimate $\bar{y}(n) = A\,x(n)$ of the additive noise. This estimate is then subtracted from the corrupted input signal to produce the system's output. If $\bar{y}(n)$ is a good estimation of $y(n)$, this results in recovery of the original voice input $s(n)$ by:

$$\bar{s}(n) = v(n) - \bar{y}(n) = s(n) + y(n) - \bar{y}(n) \cong s(n)$$

When the input signal $s(n)$ is 0, the noise cancellation system actually reduces to the topology of a standard adaptive filter as introduced in the theory reminder section. More specifically, the correspondence is as follows:

- The Noise Path of the noise canceller corresponds to the System Path of the adaptive filter
- The reference noise signal of the noise canceller translates to the reference signal of the adaptive filter
- The input signal $v(n)$ of the noise canceller becomes the desired signal $d(n)$ of the adaptive filter
- The output signal $\bar{s}(n)$ of the noise canceller is the error signal $e(n)$ of the adaptive filter

When no voice input signal is present, the output $\bar{s}(n)$ (which in this case corresponds to the error signal in the adaptive filter topology) is fed back to a coefficient update block which tunes the adaptive filter $A$ to optimize cancellation of $y(n)$ by $\bar{y}(n)$. Information from the noise source $x(n)$ such as noise distribution or frequency response may also be used for this optimization process. A typical optimization criterion used for voice applications is the minimization of the mean square error. Under the condition of no voice input ($s(n) = 0$), the mean square error is given by:

$$MSE = \frac{1}{N}\sum_{k=0}^{k=N-1}(\bar{s}(n-k) - \bar{y}(n-k))^2 = \frac{1}{N}\sum_{k=0}^{k=N-1}(s(n-k)+y(n-k) - \bar{y}(n-k))^2$$
$$= \frac{1}{N}\sum_{k=0}^{k=N-1}(y(n-k) - \bar{y}(n-k))^2 = \frac{1}{N}\sum_{k=0}^{k=N-1}e(n-k)^2$$

The noise source signal $x(n)$ is usually provided by a dedicated microphone picking up the ambient noise (acoustical background). In a typical noise cancelling microphone application, there would hence be two capturing paths: a first one focused on the voice source (for instance by beamforming) to pick-up the voice signal (corrupted by the noise at the microphone's input) and a second one, not focused on the voice, capturing the ambient noise. The first path provides the signal input to the system whereas the second path provides the reference noise input.

Adaptation of the filter coefficients is done during the time when there is no voice input signal. Hence operation of the noise canceller is linked to the Voice Activity Detector (VAD). When the VAD signals no voice activity, the system adapts the filter coefficients and applies the filter to the reference noise input. When the VAD detects voice activity, adaptation of the filter coefficients is stopped but the filter is still applied to the reference noise input. This is necessary to avoid the adaptive filter trying to track the voice signal which would result in cancellation of the speech signal. Following table summarizes the operating modes of a noise canceller for microphone applications:

| VAD output | Noise canceller mode | Noise canceller operations |
|---|---|---|
| No voice activity | Update and filter | Updates adaptive filter coefficients<br>Applies adaptive filter to noise reference |
| Voice activity detected | Filter only | Applies adaptive filter to noise reference |

**Table 1: Noise canceller operating modes**

## Echo cancellation

Echo cancellation relies on the same anti-waveform cancellation principle as noise cancellation but the overall system and signals are slightly different. Echo cancellation can be applied in multiple situations ranging from electrical line echo cancellation to acoustical echo cancellation. We illustrate the case of acoustical echo cancellers but the general principles remain unchanged for all applications. Figure 15 shows a block diagram of such an echo cancellation system in the context of a speaker phone (i.e. involving both local and remote sound sources).

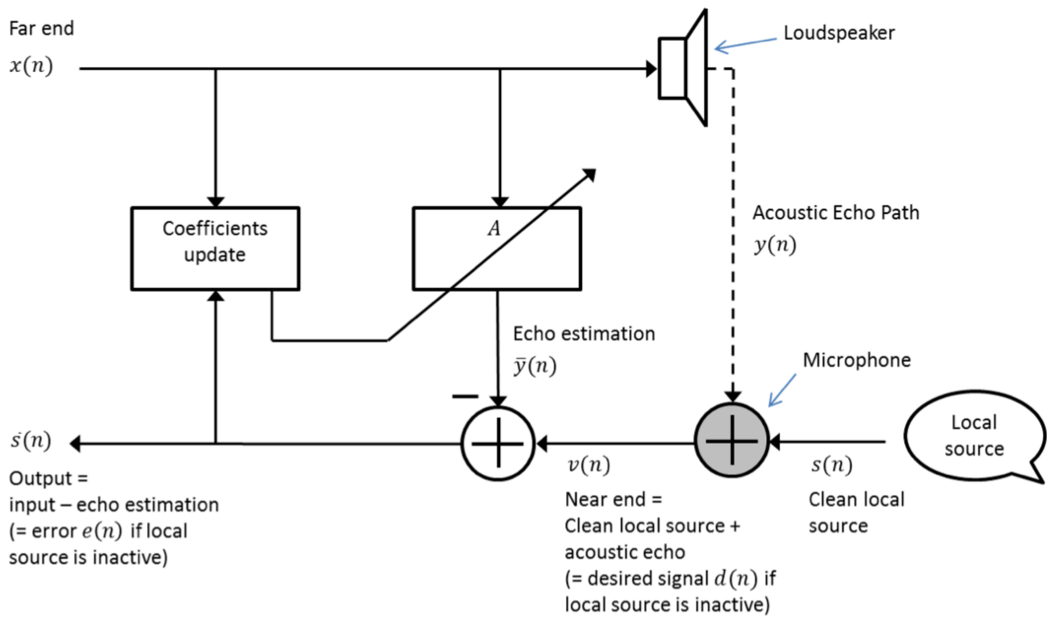
Far end
$x(n)$
Loudspeaker
Acoustic Echo Path
$y(n)$
Coefficients update
$A$
Echo estimation
$\bar{y}(n)$
Microphone
Local source
$\bar{s}(n)$
Output = input – echo estimation (= error $e(n)$ if local source is inactive)
$v(n)$
Near end = Clean local source + acoustic echo (= desired signal $d(n)$ if local source is inactive)
$s(n)$
Clean local source

**Figure 15: Acoustical echo canceller block diagram**

A far end speech signal $x(n)$ is played through a loudspeaker so that all people in the local audience can hear it. Speech from local users is collected by a microphone and transmitted through a communication channel to remote users. The problem with such a setup is that the microphone also picks-up the signal from the loudspeaker and its possible reflections on the room's floor, wall and ceiling. These signals constitute the acoustic echo $y(n)$ and corrupt the clean speech signal $s(n)$ to form the system's near end signal $v(n) = s(n) + y(n)$. If this signal was transmitted to far end users without further processing, they would hear their own voice delayed by the sum of the acoustical and communication channel delay. The presence of acoustic echo in the communication chain makes the participants feel interrupted by their own echo. Given that the whole system is a looped bidirectional system, this process would repeat over and over again which is very unpleasant in terms of user experience.

To address this issue, the far end signal $x(n)$ is passed through an adaptive filter $A$ to create an estimation $\bar{y}(n) = A\,x(n)$ of the acoustic echo signal $y(n)$. This estimation is then subtracted from the input signal $v(n)$ to generate the clean local speech signal estimation output signal $\bar{s}(n)$ to be transmitted to the remote users. If $\bar{y}(n)$ is a good enough estimation of $y(n)$, the output signal $\bar{s}(n)$ will be a useful approximation of the clean local speech signal $s(n)$ as:

$$\bar{s}(n) = v(n) - \bar{y}(n) = s(n) + y(n) - \bar{y}(n) = s(n) + y(n) - A\,x(n) \cong s(n)$$

When the local source is inactive (and there is voice activity on the far end), the echo cancellation system becomes a standard adaptive filter as follows:

- The Acoustic Echo Path corresponds to the System Path of the adaptive filter
- The far end signal $x(n)$ is considered as to the reference signal of the adaptive filter
- The near end signal $v(n)$ becomes the desired signal $d(n)$ of the adaptive filter
- The output signal $\bar{s}(n)$ translates to the error signal $e(n)$ of the adaptive filter

The signals $e(n)$ and $x(n)$ are used as inputs to a coefficient update block which dynamically tunes the coefficients of filter $A$ in order to optimize the acoustic echo estimation signal $\bar{y}(n)$.

As such, the signal processing operations involved in echo cancellation are very similar to the operations involved for noise cancellation. In both cases, the key operations lie in the adaptive filtering and related dynamic coefficients update. One major difference between both types of cancellation lies in the length of the adaptive filter. Whereas filters of a few tens or hundreds of taps work fine for noise cancellation, the acoustical impulse response in a typical echo cancellation application may last up to 80 milliseconds (or more!) resulting in very long filters (3840 taps or more at 48kHz sampling). Computing the output of such long filter in real-time can be very challenging in terms of computing resources if a straight forward time domain implementation is used. Real-time computation of long FIR filters at drastically lowered computing resources can be achieved using frequency-domain algorithms such as the FFT-based partitioned convolution algorithm presented in Ref[14]. In addition, adaptation of long filters can be slow resulting in sub-optimal convergence time. More elaborate filter coefficients update strategies may have to be used in such a case.

Another way to reduce the computational load of echo cancellation is to apply it at a reduced sampling rate. For instance, instead of using 48kHz, the echo canceller could be run at a sampling rate of 8kHz. In this case, the number of taps of the FIR filter is reduced from 3840 to 640 and the number of CPU cycles between successive samples is multiplied by 6. This results in a corresponding increase of computational resources per sample and makes real-time operation possible using straight forward time domain implementation. By using this down-sampling trick, implementation of an echo canceller turns out to be very similar to the implementation of a noise canceller and hence will not be further detailed in this whitepaper.

Operation modes of an echo canceller are also driven by voice activity detection as is the case for a noise canceller. However, the situation is slightly more complex as there are actually two types of voice activity detection involved: near-end voice activity detection which reports speech activity by a local user and remote-voice activity detection which reports speech activity at the remote end.

Table 2 shows the operating modes of an acoustical echo canceller:

| **VAD output** | **Noise canceller mode** | **Noise canceller operations** |
|---|---|---|
| No voice activity or Near-end voice activity only | Freeze | Does nothing |
| Far-end voice activity only | Update and filter | Updates adaptive filter coefficients<br>Applies adaptive filter to far-end speech |
| Both near-end and far-end voice activity | Filter only | Applies adaptive filter to far-end speech |

**Table 2: Echo canceller operating modes**

## Spatial domain noise cancellation: the GSC Beamformer

The Generalized Sidelobe Canceller (GSC) is a major representative of the family of data-dependent (adaptive) beamforming techniques. These are based on spatial domain noise cancellation to pass the signal from the desired direction while rejecting noise coming from other directions (i.e. reducing the sidelobes of the directivity pattern). Optimization of the adaptive filters is often realized by mean-square error minimization. Unfortunately, traditional Least Mean-Square (LMS) optimization can lead to sub-optimal results as there is no constraint placed on the distortion of the desired signal; only the mean-square error is minimized. Additional constraints in terms of transfer function of the desired signal must be placed on the optimization problem, resulting in a constrained mean-square optimization problem. GSC is a beamforming topology (see Figure 16) that can be used to implement a variety of linearly constrained adaptive array beamforming algorithms. Its main feature is the separation of processing into two paths:

- The "Signal" path, which implements a fixed beamformer (with constraints on the desired signal)
- The "Noise" path, which is the adaptive part of the system, optimizes a set of filters to minimize noise power in the output. A blocking matrix $B$ is inserted at the entry of the "Noise" path to eliminate the desired signal, making sure that only noise power is minimized at the output.

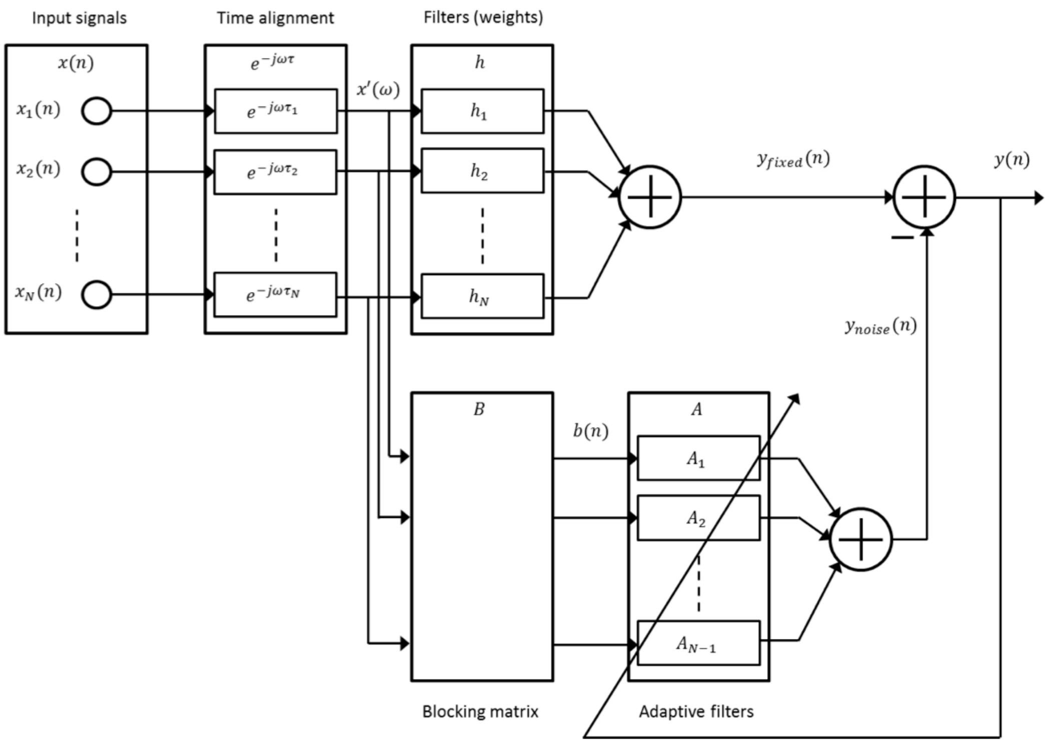

**Figure 16: GSC beamforming system block diagram**

Looking at the "Signal" path, the input signal $x(n)$ is first time-aligned to produce the signal $x'(n) = (x'_1(n), \dots, x'_N(n))$. This signal is passed through a set of real valued weighting filters $h = (h_1, \dots, h_N)$ resulting in the constrained, fixed beamformed signal $y_{fixed}(n)$. This can be expressed in the time domain as:

$$y_{fixed}(n) = \sum_{i=1}^{N} h_i(n) * x'_i(n)$$

where $h_i(n)$ is the impulse response of weighting filter $h_i$ and $*$ represents the convolution operator. The fixed beamformer is often implemented as a simple Delay-Sum beamformer resulting in:

$$y_{fixed}(n) = \frac{1}{N} \sum_{i=1}^{N} x'_i(n)$$

The "Noise" (or adaptive) path consists in two major sections: the blocking matrix and the adaptive filter. The blocking matrix is used to remove the desired signal from the "Noise" path. As the desired signal is time-aligned on all inputs, blocking will happen if the rows of the matrix sum to zero. A widely used blocking matrix is the $(N-1) \times N$ Griffiths-Jim matrix (Ref[15]) given by:

$$B = \begin{pmatrix} 1 & -1 & 0 & 0 & \dots & 0 \\ 0 & 1 & -1 & 0 & \dots & 0 \\ \dots & \dots & \dots & \dots & \dots & \dots \\ 0 & \dots & 0 & 1 & -1 & 0 \\ 0 & \dots & 0 & 0 & 1 & -1 \end{pmatrix}$$

After the blocking matrix, the noise signal $b(n)$ is passed through a set of real valued adaptive filters $A = (A_1, \dots, A_{N-1})$ to give the "Noise" path output $y_{noise}(n)$ which can be computed in the time domain as:

$$y_{noise}(n) = \sum_{i=1}^{N-1} A_i(n) * b_i(n)$$

where $A_i(n)$ is the impulse response of the adaptive filter $A_i$, $*$ represents the convolution operator and $b_i(n)$ is the $i^{\text{th}}$ component of the vector $B \cdot x'(n)^T$. The overall system output is then given by

$$y(n) = y_{fixed}(n) - y_{noise}(n)$$

Given that the "Signal" path contains the constrained estimation of the desired signal and that the "Noise" path contains only noise and interference signals, optimizing the adaptive filter $A$ is indeed equivalent to finding the linearly constrained minimum variance beamforming solution. In addition, as the constraints are applied in the "Signal" path, unconstrained (N)LMS optimization can be used to adapt the "Noise" path filter coefficients as follows (for NLMS):

$$A_{i,j}(n+1) = A_{i,j}(n) + \mu \frac{1}{\|b_i(n)\|^2 + \delta} y(n) b_i(n-j)$$

where $\mu$ is the NLMS optimization step size $A_{i,j}(n)$ is the $j^{\text{th}}$ element ($j = 0, 1, \dots L-1$) of the impulse response of the adaptive filter $A_i(\omega)$ and $\| \, . \, \|^2$ represents the squared $L_2$ norm of the signal $b_i(n)$ over the length $L$ of the adaptive filters. $\delta$ is an optional small positive value to avoid division by zero. As in traditional noise cancelling, the adaptive filter coefficients are updated during speech absence. A set of optimal filters depending on steering direction can be maintained by the host application. They can be initialized at start-up be panning through possible steering directions.

# Implementation study on xCORE-200

In this section, we study the implementation of different smart microphone algorithms for noise-cancelling microphones array applications, including the stages of beamforming (BF), voice activity detection (VAD) and noise cancellation (NC) on the XMOS xCORE-200 architecture.

## Study of main building blocks

We start by reviewing the computational load of the main building blocks involved, Finite Impulse Response (FIR) filters and Adaptive FIR filters.

### Finite Impulse Response (FIR) filters

One of the key building blocks is the classical Finite Impulse Response filter. It's time domain implementation is given by:

$$y(n) = h * x(n) = \sum_{i=0}^{L-1} h(i)x(n-i)$$

where $L$ is the number of taps of the filter, $x$ is the input signal and $h = (h(0), h(1), \dots, h(L-1)$ is the filter's impulse response. On the xCORE-200, the computational complexity of an FIR can be estimated as $2.5L + 10$ where $10$ represents an uncompressible overhead to setup the filter and assuming double word fetch of filter coefficients. Its uses $L$ words for coefficients storage and $L$ or $2L$ memory words for delay line storage depending on implementation (Ref[1], Ref[16]). For the rest of this document we will assume that $3L$ memory words in total are used for coefficients and delay line.

### Adaptive FIR filters

The computational and memory load for the application of an $L$ taps adaptive FIR filter $A$ are clearly the same as for a fixed FIR. Hence we only need to study the load for the filter coefficients update part. Considering the NLMS algorithm, the filter coefficients update equations are given by:

$$\bar{y}(n) = A(n) \cdot X(n)^T$$

$$e(n) = d(n) - \bar{y}(n)$$

$$A(n+1) = A(n) + \mu \frac{1}{\|X(n)\|^2 + \delta} e(n)X(n)$$

where $A(n) = \left(a_0(n), a_1(n), \dots, a_{P-1}(n)\right)$ is the adaptive filter coefficients vector at time $n$, $X(n) = \left(x(n), x(n-1), \dots, x(n-P+1)\right)$ is the reference signal vector at time $n$.

The first equation just represents the application of the $L$ taps FIR filter and hence accounts for $2.5L + 10$ instructions and $3L$ memory words. The second equation can be estimated to 2 instructions (fetching $d(n)$ and subtraction).

The third equation is a bit more complex as it involves computing the inverse of the squared $L^2$norm of the signal $X(n)$. As $\|X(n)\|^2 = \sum_{i=0}^{L-1} x(n-i)^2$, the squared $L^2$ norm can be computed using a running average over a circular buffer containing the elements $x(n-i)^2$ and requires about 15 instructions per sample (by noticing that the running average can be computed by taking the previous running average value, adding the square of the new input sample and subtracting the one that has just been pushed out of the buffer). Two instructions (load and add) must be considered for the addition of $\delta$. The inverse operation is estimated to require 35 instructions (including shifts for data alignment for maximum dynamic range). Hence the term $\mu \frac{1}{\|X(n)\|^2+\delta}\, e(n)$ can be computed using $15 + 2 + 35 + 3 = 55$ instructions (where the additional 3 instructions cover the load of $\mu$ and the multiplications by $\mu$ and $e(n)$. Finally, the update of the filter coefficients vector requires fetching the corresponding elements in $A(n)$ and $X(n)$, performing the multiply-add operation and writing the coefficient back for each coefficient. By ensuring double word alignment for $A(n)$ fetches and writes of filter coefficients can be done using double word load/store instructions. If performed on 32bits words, the multiply-add operation requires the operations of saturations and shift for alignment. This results in a computational complexity of $5L + 10$ instructions, where $10$ represents an incompressible overhead. Hence computation of the third update equation requires a total of $55 + 5L + 10 = 5L + 65$ instructions. The only additional memory required is the buffer to store the squared magnitude samples which represents $L$ words.

By adding the computational requirements of the three update equations, the overall requirements for the update of the coefficients of an $L$ taps FIR filter using the NLMS algorithm can be estimated as:

- $2.5L + 10 + 5L + 65 = 7.5L + 75$ instructions
- $4L$ memory locations ($1L$ for the coefficients, $2L$ for the filter's delay line and $1L$ for the squared samples used for signal energy computation)

## Delay-Sum based implementation

Let's consider an equally spaced linear array with $N = 7$ microphones and $d = 0.028$m feeding a simple Delay-Sum beamformer coupled to a Combined VAD presented earlier in this document. A noise canceller based on a 128 taps NLMS adaptive filter receives its input from the beamformer and its reference from an additional microphone picking up the ambient noise. The LSED VAD is implemented using three bands (50Hz to 1.1kHz, 1.1kHz to 2,8kHz and 2.8kHz to 6kHz), covering the speech bandwidth. Above 6kHz both the background noise and voice energy will be very low in typical environments. The sampling rate of the system set to 48kHz. The block diagram of the resulting noise cancelling microphone application is shown in figure Figure 17.

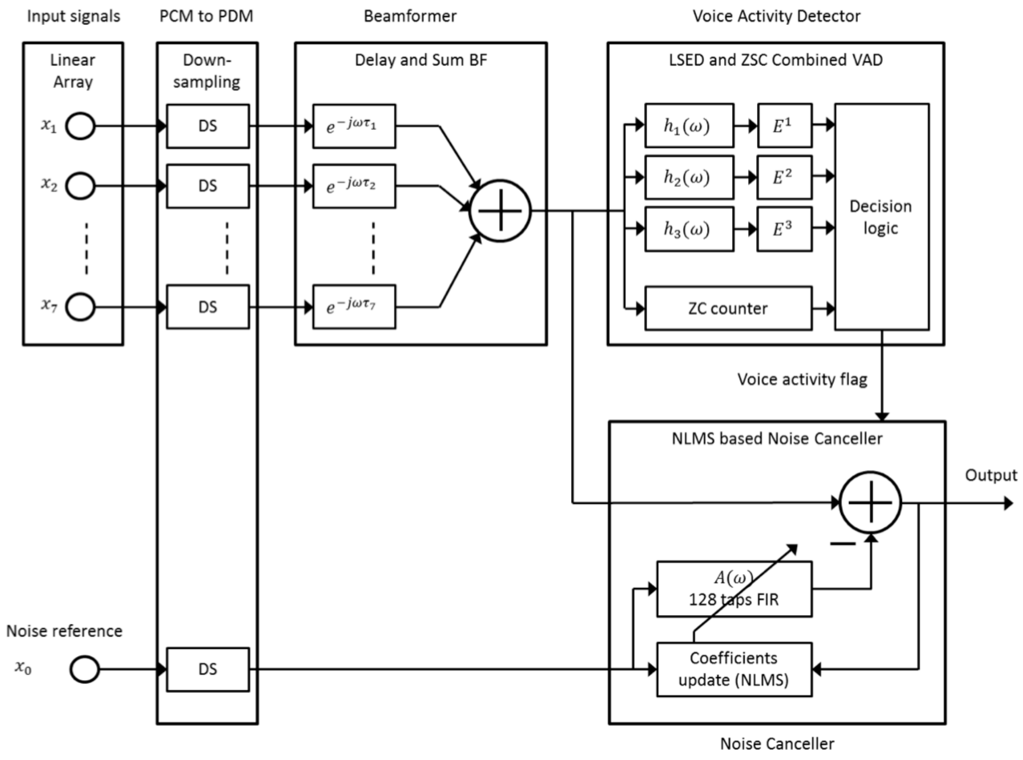

**Figure 17: Delay-Sum based implementation**

The complete system front-end consisting in PDM signal acquisition, PDM to 48kHz PCM conversion and delays (and sum) can be taken directly from the HiRes Delay-Sum example Ref[6], Ref[7]. It can thus be implemented using five 62.5MHz cores of an xCORE-200 MCU, where the 5th core is responsible for channel summing (and output gain) is only lightly loaded in terms of real-time signal processing tasks.

***Note:*** *For diagram clarity reasons, Figure 17 shows the delays as following PDM to PCM conversion, whereas both processes are actually interleaved to enable high resolution. This note also applies to Figures 19 and 20.*

The computational load of the noise canceller can be computed using the estimates established in the previous section. Based on 128 taps NLMS filter, the noise canceller requires:

- $2.5L + 10 = 2.5 \times 128 + 10 = 330$ instructions per sample when coefficients are not updated (i.e when voice is detected by the VAD). At a sampling rate of 48kHz, this requires about **15.9 MIPS**.
- $7.5L + 75 = 7.5 \times 128 + 75 = 1035$ instructions per sample when coefficients are updated (i.e. when no voice is detected by the VAD). At a sampling rate of 48kHz, this requires about **49.7 MIPS**
- $4L = 4 \times 128 = 512$ words for filter coefficients and delay lines. These are considered to be 32-bit words, resulting in **2kBytes**.

Given these figures, implementation of the 128 taps NLMS noise canceller using a single 62.5MHz core of an xCORE-200 MCU seems realistic.

Finally, we need to analyze the computational requirements of the VAD consisting in the combination of a LSED (Linear Sub-band Energy Detector) and a ZCD (Zero Crossing Detector). The LSED first splits the input signal into three bands covering for instance 50Hz to 1.1kHz, 1.1kHz to 2.8kHz and 2.8kHz to 6kHz respectively. The band crossovers $h_i(\omega)$ are designed using band-pass filters implemented as four cascaded biquad cells (two for high-pass and two for low-pass, resulting in 4th order filters at both ends of each band). This accounts for a total of 12 biquad cells and each biquad can be computed in approximately 40 instructions (Ref[17]) taking some overhead (10 instructions) to setup the filters into account. 9 32-bits words of memory are required to store coefficients and states per biquad. Hence, the separation of the input signal into the three different frequency bands requires about $12 \times 40 = 480$ instructions per sample and $12 \times 9 = 108$ 32-bit words of memory. For each band $B$ the energy $E^B$ over a time frame of $K = 480$ samples (10ms at 48kHz sampling rate) is computed. This is similar to the $L^2$ norm computation step in the coefficients update of an NLMS adaptive filter and hence energy computation for the three bands LSED requires $3 \times 15 = 45$ instructions. $3 \times K = 3 \times 480 = 1440$ 32-bit memory words ($\cong 5.8$ kBytes) are required to store the squared magnitude samples for running average energy computation.

The decision process of LSED according to $E^B > k\, {E_r}^B$ requires about $4$ instructions per band, resulting in an estimation of $12$ instructions across all bands. The energy threshold update for each band is computed as ${E_r}^B(n+1) = (1-p){E_r}^B(n) + p{E_{noise}}^B$ requiring about 7 instructions per band, resulting in $3 \times 7 = 21$ instructions per sample. A few (8) memory words are required to store $p$, $1-p$, ${E_r}^B$ and ${E_{noise}}^B$ .

The ZCD VAD is much simpler to implement as it just counts the number of signal sign change over a given period of time. For a typical speech signal, the number of zero-crossing in a given frame can be characterized by the following formula which gives the decision condition of the ZCD VAD:

$$R_{min} \leq N_{zc}(n) \leq R_{max}$$

For a frame length of 10ms, we have $R_{min} = 5$ and $R_{max} = 15$. Computation of $N_{zc}(n)$ can be achieved simply by maintaining a buffer of sign changes over the frame length. Each time a new sample is received a '1' is written to this buffer if a sign change happened between the previous and current sample. $N_{zc}(n)$ is then simply given by the number of '1's in the buffer which can be computed on the fly by adding the new value and subtracting the one being pushed out of the buffer. The complete ZCD VAD thus requires $480$ bytes of memory for the sign change buffer (1 byte per sign change over 10ms at 48kHz) and about 30 instructions per input sample.

Combining the above and adding an overhead of 50 instructions per sample and 40 32-bit words for logic and control of the LSED and ZSC combination, the computational requirements of the VAD can be summarized as:

- $480$ (biquads for band separation in LSED) $+ 45$ (energy per band computation in LSED) $+ 12$ (decision process in LSED) $+27$ (energy per band update in LSED) $+30$ (ZSC) $+50$ (overhead) $= 644$ instructions. At a sampling rate of 48kHz, this corresponds to about **31MIPS**.
- $108$ 32-bit words (for biquads for band separation in LSED) $+1440$ 32-bit words (for energy per band computation in LSED) $+ 480$ bytes (for sign change buffer in ZSC) $+40$ 32-bit words (overhead) $\cong$ **6.8kBytes** of memory.

The above figures show that the combined VAD can be implemented using only a fraction of a single 62.5MHz core of an xCORE-200 MCU.

Combining figures for the front-end (implementing the Delay-Sum beamformer), the noise canceller and the VAD, the full implementation of the noise-cancelling microphone solution can be implemented using **6 cores running at 62.5MHz** on the xCORE-200 architecture and using **less than 10kBytes of additional data memory** when compared to the HiRes Delay-Sum example. This assumes that the VAD runs on the same core as the channel summing and output gain of the HiRes Delay-Sum example (see Figure 21).

## Enhancement using Sub-band beamforming

As shown in Figure 5 using an equally spaced linear array with a simple Delay-Sum beamformer results in high frequency dependency of the directivity pattern. This is undesirable for good speech quality. We therefor propose to slightly modify the array topology to be suitable for Sub-Band beamforming.

We still consider a linear array with $N = 7$ microphones but this time we space the $5$ inner microphones with $d = 0.028$m and the $2$ outer microphones by $d' = 0.056$m. We consider a 3 band sub-band beamformer consisting in three nested arrays of 3 microphones (low frequency band, mic spacing = 0.112m), 5 microphones (mid frequency band, mic spacing = 0.056m) and 5 microphones (high frequency band, mic spacing = 0.028m) respectively as shown in Figure 11. Figure 18 shows the enhanced directivity patterns for the different bands at frequencies of 1kHz, 2kHz and 5 kHz respectively (to be compared with Figure 5).

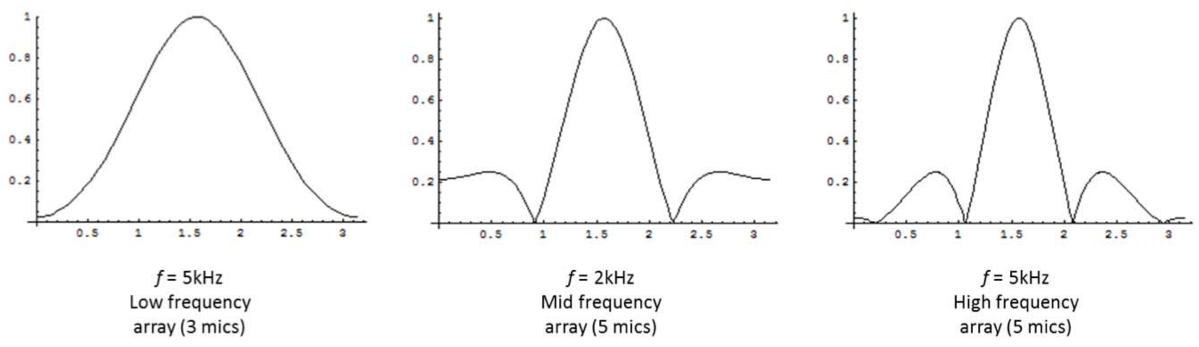

**Figure 18: Directivity patterns for sub-band array beamformer**

The frequency band filters ($h_1(\omega)$, $h_2(\omega)$ and $h_3(\omega)$) are implemted as band-pass filters using 4th order IIR filters implemented as two cascaded biquad cells for high-pass and low-pass at band edges (identical to the band-pass filters of the LSED described above). By setting the array sub-bands to the same ranges as the VAD sub-bands, the complete system including front-end, beamformer, voice activity detector and noise canceller can be implemented as shown in Figure 19.

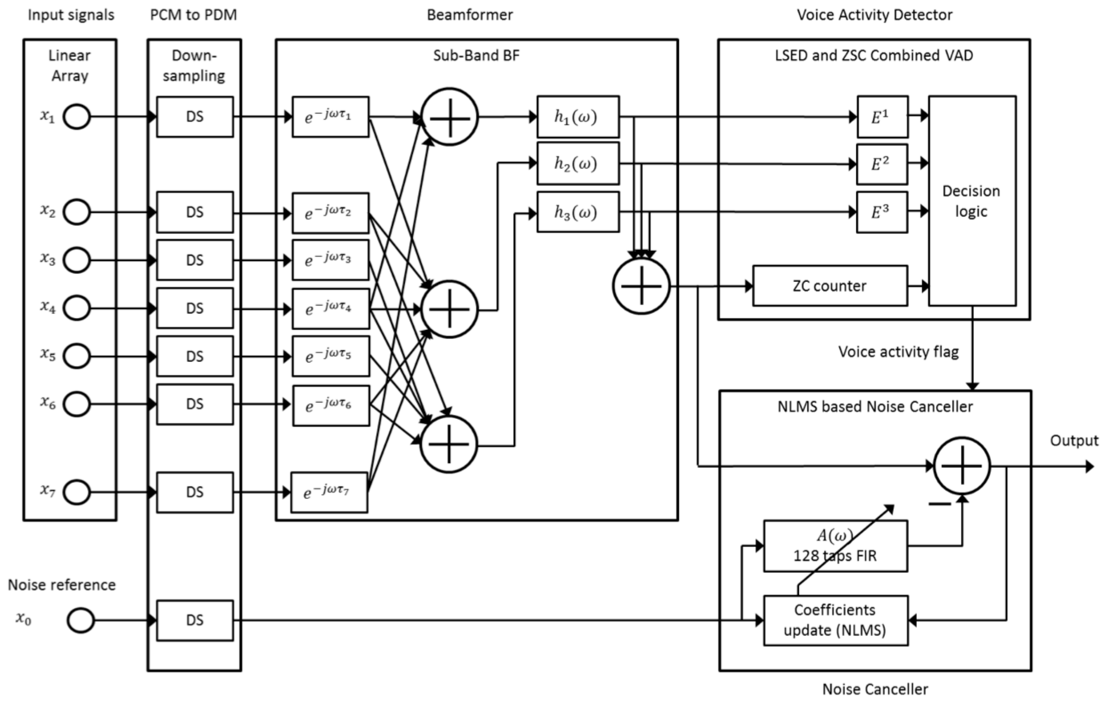

**Figure 19: Sub-Band beamforming implementation**

An interesting property of the Sub-Band beamforming based implementation is that it comes almost for free in terms of computing resources when compared to the basic Delay-Sum beamformer implementation:

- Instead of outputting a single Sum signal, the front end based on the HiRes Delay-Sum example is modified to output 3 Sum signals, corresponding to the 3 nested arrays. This is a minor modification and should not change the resource usage of five 62.5MHz cores on the xCORE-200 architecture.
- The frequency band filters used in the Sub-Band beamformer are actually already computed in the LSED VAD of the basic implementation and hence they come for free! The only added requirement is the computation of the sum of the outputs of these filters to feed the ZSC VAD and the noise canceller.
- No other changes are required.

As such, implementation of the Sub-Band beamforming based noise cancelling microphone application on the xCORE-200 architecture can also be implemented using **6 cores at 62.5MHz** and **a premium of about 10kBytes of memory** compared to the HiRes Delay-Sum example (see Figure 21).

## Introducing Sub-Band GSC beamforming

A further improvement (although not for free in terms of computational resources) can be implemented by integrating GSC beamforming for each of the nested arrays as shown in Figure 20.

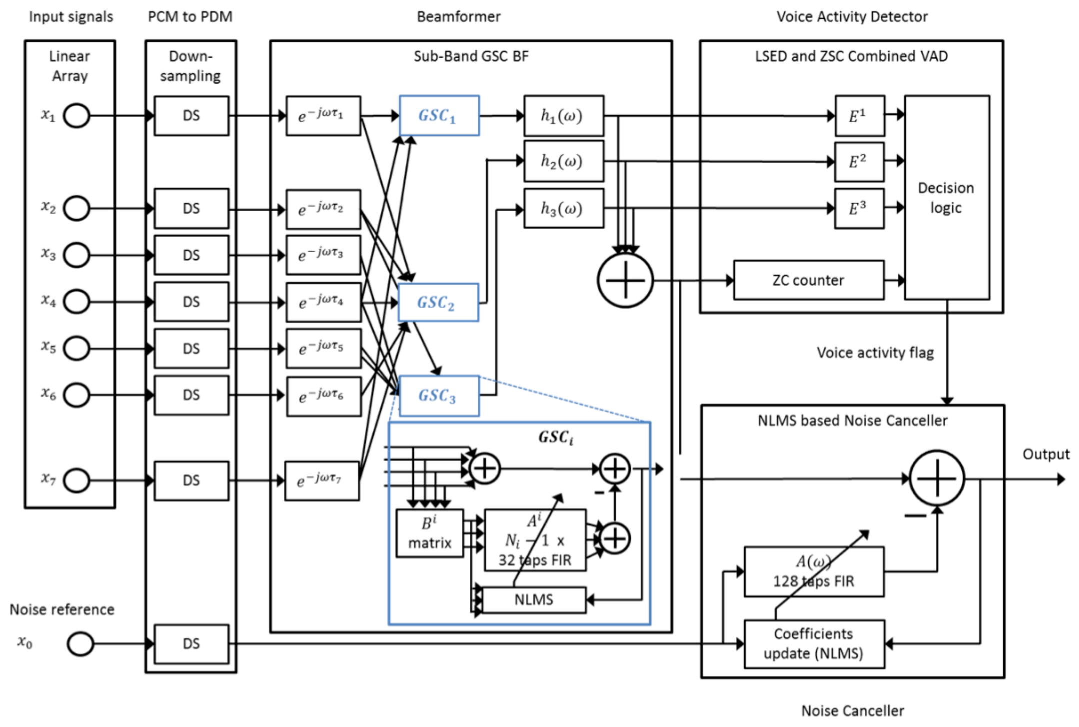

**Figure 20: Sub-band GSC beamforming implementation**

This topology is similar to the one presented in Figure 19, except for the addition of the adaptive paths (blocking matrixes $B^i$ and adaptive filters $A^i$) in the respective sub-bands Delay-Sum beamformers.

Computation of the output of the blocking matrixes $B^i$ requires about $2\ (N_i - 1) + 5$ instructions, where $N_i$ is the number of microphones in sub-array $i$. In our example case we have $N_1 = 3$, $N_2 = 5$ and $N_3 = 5$, resulting in $2\times(3-1)+5+2\times(5-1)+5+2\times(5-1)+5 = 35$ instructions per sample.

There are $N_i - 1$ adaptive filters per band to be computed. This represents $(3-1)+(5-1)+(5-1) = 10$ adaptive filters in total in our case. Assuming that each filter has 32 taps, the corresponding computational load for updating the coefficients can be estimated as $10\times(7.5\,L' + 75) = 10\times(7.5\times32+75) = 3150$ instructions per sample. Computation of the filters without coefficients update requires about $10\times(2.5\,L' + 10) = 10\times(2.5\times32+10) = 900$ instructions. In terms of memory the requirements for the bank of adaptive filters would require about $10\times4L' = 10\times4\times32 = 1280$ 32-bit words (about **5.1kBytes**).

At 48kHz sampling rate, the computational load for computing the additional adaptive filters (without coefficients update, i.e. when voice is present) accounts for about 43.2MIPS which can easily be fitted into a single core at 62.5MHz. On the other hand, coefficients update (applied when no voice is detected) requires about 151.2MIPS which is clearly too high for a single core. An elegant workaround is to use a round-robin approach to adaptive filters coefficients update, where only the adaptive filters for a single band are updated when a sample is received and moving to the next band with the next sample. This effectively reduces the MIPS load to the one of the largest array in the system. In our case where the largest array has 5 microphones, this would reduce the load to $4\times(7.5L' + 10) = 4\times(7.5\times32+75) = 1260$ instructions. At a sampling rate of 48kHz this corresponds to **60.5 MIPS**, which allows for implementation on a single 62.5MHz core.

With the round-robin approach to GSC beamformer coefficients update, the proposed noise cancelling microphone application based on Sub-Band GSC beamforming can be implemented using just one additional core, resulting in a total of **7 cores at 62.5MHz** on an xCORE-200 MCU and about **15kBytes of additional memory** when compared to the HiRes Delay and Sum example.

## Cores distribution

Following diagram shows the proposed cores distribution for the complete Sub-band GSC beamforming based noise cancelling microphone application.

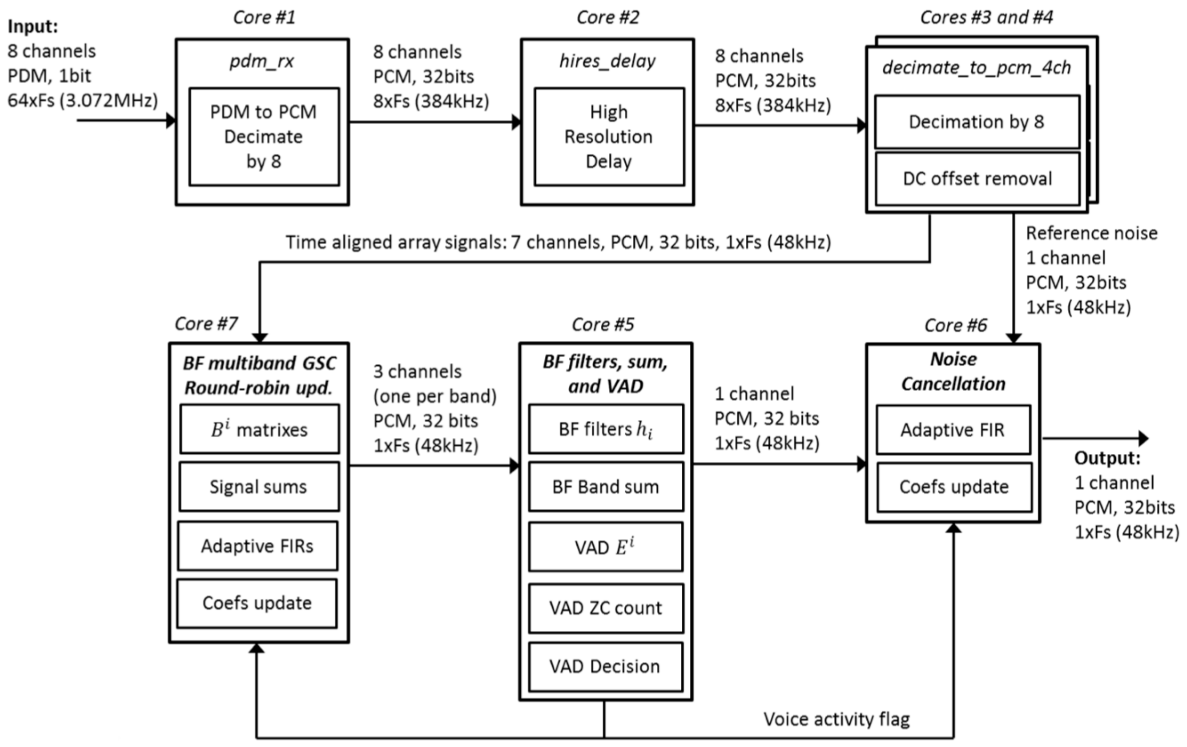

**Figure 21: core distribution of noise cancelling microphone example applications**

The distribution is as follows:

- Cores #1 to #4 run the same software as the Hi Res Delay and Sum example, converting the PDM inputs to PCM at 48kHz, applying delays for time alignment and removing DC offset.
- Core #5 runs the GSC for the different nested arrays of the beamformer.
- Core #6 implements the band filters and the final sum for the beamformer. It also implements the Voice Activity Detector.
- Core #7 is dedicated to the Noise Canceller

For an implementation of the Sub-band beamforming based solution, Core #7 would be removed and the 7 channels time aligned array signals would be linked directly to Core #5 (with the additional task of summing the time aligned array signals per band).

An optional Echo Canceller can be added at the output of the noise canceler. This would require an additional far-end speech input (assumed to be received at 48kHz). Let's consider a maximum echo time of 80ms. This corresponds to 640 samples at 8kHz sampling rate. After decimation by a factor 6 to a sampling rate of 8kHz, a 640 taps NLMS Echo Canceller can be implemented using $7.5 \times 640 +$

$75 = 4875$ instructions per sample for both coefficients update and adaptive filter computation, representing about **39 MIPS**. $4 \times 640 = 2560$ 32-bit memory words (**10kBytes**) are required for filter coefficients and delay lines. Implementation of the 2 channels decimator by cascaded FIRs should require about **10 MIPS** and **a few kBytes** of memory. Based on the figures for the VAD running at 48kHz, an 8kHz far-end VAD could be implemented using about **4 MIPS**. After adaptive filtering, the echo cancelling signal must be up-sampled back to 48kHz (and time aligned) to be subtracted from the main input signal to create a 48kHz output. This can again be implemented using cascaded FIRs and should not require more than **6 MIPS** and **a few kBytes** of memory. These figures show that such an echo canceller could also be implemented using a single core.

Table 3 summarizes the proposed core distribution assuming 62.5MHz xCORE-200 cores:

| Core | Algorithms | Remarks |
|---|---|---|
| #1 | 8 channels PDM to 32-bit PCM converter | 64xFs (3.072MHz) to 8xFs (384kHz) |
| #2 | 8 channels high resolution delay | 2.6us (1/384kHz) resolution |
| #3 | 4 channels decimator by 8 and DC removal | 8xFs (384kHz) to 1xFs (48kHz) |
| #4 | 4 channels decimator by 8 and DC removal | 8xFs (384kHz) to 1xFs (48kHz) |
| #5 | Sub-band beamformer<br>Combined LSED and ZCD voice activity detector | 3:5:5 nested array of 7 microphones<br>3 bands beamforming<br>Drives the Voice activity flag |
| #6 | NLMS Noise cancellation | Coefficients update driven by Voice activity flag |
| #7 | Sub-band GSC support for beamformer | Optional<br>Required only for GSC support<br>Coefficients update driven by Voice activity flag |
| #8 | *2 channels Decimator by 6*<br>Echo cancellation<br>Far-end voice activity detection<br>1 channel Up-sampler by 6<br>Input delay (time alignment) and up-sampled echo cancelling signal subtraction | *Optional*<br>Not shown in figure 21,<br>Connected after the noise canceller and requires a far end speech input<br>Echo cancellation operates at Fs/6 (8kHz)<br>Coefficients update driven by Voice activity flag and Far-end voice detection flag |

**Table 3: Cores distribution summary**

An additional benefit of the presented smart microphone solutions is their low latency. According to Ref[1], the complete front-end processing (PDM to 48kHz PCM conversion and high resolution delay) and system I/O exhibits a latency below 2ms (possibly below 1ms if minimal phase filters are used). As the complete system is implemented in the time-domain, it is well suited to SSB (Sample-by-Sample Based) processing. Hence the only additional latencies in the system are linked to the band filters $h_i$ and core-to-core data passing.

Finally, if the whole system (except the echo canceller) was run at 24kHz, this would immediately result in 50% savings on the MIPS budget as it is proportional to the sampling rate. However, savings would be even higher as FIR filter lengths (i.e. number of taps) could also be reduced by a factor 2 to keep the same impulse response duration. Hence reducing the sampling rate to 24kHz would save about 2/3 of the MIPS budget, allowing to run Sub-band GSC based beamforming, voice activity detection and NLMS noise cancelling on a single 62.5MHz core.

## About spatial aliasing

In the review of the fundamentals of array processing, we have seen that the distance $d$ between individual microphones in a linear array should satisfy

$$d < \frac{\lambda_{min}}{2}$$

in order to avoid spatial aliasing (where $\lambda_{min}$ is the wavelength of the highest frequency to be considered by the system). Given that speech signals have most of their energy located below 5kHz and that background noise typically follows a $1/f$ spectrum (at least above 1kHz), we can conclude that the amount of energy in the signals captured by the microphones above 6kHz is very low. The array elements distance chosen in our examples ($d = 0.028$m) corresponds to a minimum wavelength $\lambda_{min} = 0.056$m, which corresponds to a frequency of 6.125kHz (assuming a propagation speed of 343m/s for sound in the air). Hence spatial domain aliasing should not be much of a concern as almost no energy above 6kHz should be present in the system. For the Sub-band and Sub-band GSC beamforming cases, this is even less of a concern as the band-filters will remove frequencies above 6kHz from the signals of interest.

# Conclusion

This whitepaper has introduced the concept of smart microphone as an evolution of the classical microphone. From pure sound pressure to electrical signal converters, voice capture devices have evolved into complex systems featuring multiple capture channels coupled with digital signal processing capabilities.

In the context of voice user interfaces, beamforming noise cancelling smart microphones are widely used. Their purpose is to deliver a clean speech signal suitable for subsequent processing like key word detection or semantic analysis. Such a smart microphone typically embeds DSP functionalities for focusing voice capture to the direction of the speaker (beamforming), detecting voice presence (voice activity detector) and removing background noise (noise cancellation). Echo cancellation may also be part of the embedded DSP capabilities, especially for smart microphone applications involving bi-directional, closed-loop communication such as conferencing systems.

Commonly used DSP algorithms for beamforming, voice activity detection and noise/echo cancellation have been presented, preceded by a reminder of the fundamentals of array processing and adaptive filtering theory.

Finally, implementation of complete beamforming noise cancelling smart microphone solutions on the xCORE-200 architecture has been reviewed. Based on the existing HiRes Delay-Sum beamformer example, the studied solutions extend it by implementing more sophisticated beamforming algorithms and adding voice activity detection and noise cancelling features. Our estimations show that complete beamforming noise cancelling smart microphone solutions can be implemented on the xCORE-200 architecture with a small overhead of 1 or 2 cores (depending on beamformer choice) running at 62.5MHz compared to the basic HiRes Delay-Sum example (which is using 5 cores). This corresponds to a relative overhead of 20% to 40%. As such, a complete smart microphone solution can be implemented using 6 or 7 cores, resulting in a resource usage of about 38-44% on a 16 cores 500MHz device. Echo cancellation can be added at the expense of an additional core, resulting in 50% resources occupancy. One has to keep in mind that the studied examples have been considered to operate at a sampling rate of 48kHz (except for echo cancellation). By using lower sampling rates, the overhead due to advanced beam forming, voice activity detection and noise cancellation would be reduced accordingly.

This shows that the xCORE-200 architecture is a premium choice for the implementation of smart microphone applications.

## References

1. Real-Time Digital Signal Processing on the xCORE 200 architecture, by T. Heeb, A. Stanford-Jason and T. Leidi, whitepaper, 2016: http://www.xmos.com/published/real-time-dsp-on-xcore200-architecture?version=latest
2. I.A. McCowan. "Robust Speech Recognition using Microphone Arrays," PhD Thesis, Queensland University of Technology, Australia, 2001
3. S.J. Elliot and P.A. Nelsson, "Active noise control." IEEE Signal Proc. Mag, pages 12-35, 1993
4. M. H. Hayes, "Statistical Digital Signal Processing And Modeling." Hoboken, NJ: John Wiley & Sons, INC, 1996
5. XMOS Microphone array library user's guide: http://www.xmos.com/published/lib_mic_array-userguide?version=latest
6. XMOS Microphone array library source code: https://www.xmos.com/published/lib_mic_array-sw?version=latest
7. XMOS *High Resolution Delay and Sum* example (part of the XMOS Microphone array library package). Refer to application note AN00218: http://www.xmos.com/published/an00218-high-resolution-delay-and-sum?version=latest
8. A. Davis, S. Nordholm and R. Togneri, "Statistical voice activity detection using low-variance spectrum estimation and an adaptive threshold," Audio, Speech, and Language Processing, IEEE Transactions on, vol. 14, pp. 412-424, 2006.
9. B. Atal and L. Rabiner, "A pattern recognition approach to voiced-unvoiced-silence classification with applications to speech recognition," Acoustics, Speech and Signal Processing, IEEE Transactions on, vol. 24, pp. 201-212, 1976.
10. L.R Rabiner and M.R. Sambur, "An Algorithm for determining End-points of Isolated Utterances", Bell Technical Journal, Feb 1975, pp 297-315
11. P. Lueg, "Process of silencing sound oscillations." U.S. Patent 2 043 416, 09-Jun-1936.
12. B. Widrow, P.F. Titchener, and R.P. Gooch, "Adaptive Design of Digital Filter", Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, 6:243-246, March 1981
13. E.R. Ferrara, Jr., and B. Widrow, "The Time-Sequenced Adaptive Filter", IEEE Transactions on Acoustics, Speech and Signal Processing, 29(3):679-683, June 1981, and IEEE Transactions on Circuits and Systems, CAS-28(6):519-523, June 1981.
14. E. Armelloni, C. Giottoli and A. Farina, "Implementation of Real-Time Partitioned Convolution on a DSP Board", IEEE Workshop on Applications of Signal Processing to Audio and Acoustics October 2003, New Paltz, NY, USA.
15. L. Griffiths and C. Jim, "An alternative approach to linearly constrained adaptive beamforming", IEEE Transactions on Antennas and Propagation, vol. 30(1), pp. 27–34, January 1982.
16. xCORE-200: The XMOS XS2 Architecture ISA: http://www.xmos.com/published/xs2-isa-specification?version=latest
17. Digital Signal Processing on the XCore XS1 for Embedded Developers http://www.xmos.com/published/digital-signal-processing-on-the-xcore-xs1-for-embedded-developers?version=latest