A Realtime Multichannel Environment for Microphone Arrays

M. Eichler, A.Lacroix

Institute of Applied Physics, J.W.Goethe-University, Frankfurt, Germany
eichler@iap.uni-frankfurt.de

Abstract: Microphone arrays are widely used in applications such as beamforming, sound field analysis, sound source localization, speaker tracking, noise cancellation and others. Aiming at implementing a realtime application, a software environment running on Microsoft Windows XP has been created for full-duplex realtime data processing. The system comprises a data-handling class library (coded in native C++) and a graphical user interface (GUI) written in C#.NET. The data-handling framework supports any number of audio I/O devices (and thus, channels); all active devices are controlled and run synchronously. Filter algorithm development is supported by a C++ programming interface which allows implementing custom filter code in satellite dynamic link libraries (DLLs), each of which again may handle any number of input and output channels. At runtime, multiple filter DLLs can be active simultaneously and are controlled via the GUI. In this paper, the concept of the developed software environment is outlined. Also, using the delay-and-sum beamformer as example, the working principle is demonstrated including analysis tools like oscilloscope and geometry editor.

1. Introduction

A microphone array consists of two or more microphones arranged in a fixed geometric pattern, uniformly or non-uniformly spaced in one or more dimensions. Processing the sound signals from each microphone, the directivity of the array as a whole can be shaped and controlled (beamforming) ([1]). Using beamforming and other signal processing techniques, detection of sound sources is possible by determining the respective direction of arrival (DOA) of sound waves impinging on the array. Conference systems can be equipped with microphone arrays that allow for microphone-independent voice transmission, thus giving more movability and comfort to the speaker. Other applications lie in the field of hearing aids (direction filtering), speech enhancement for ASR systems, automotive telephony and voice control, mobile devices and others. With the goal to obtain an evaluation and testing tool for microphone array applications, a real-time full-duplex environment has been developed which will be described in this paper.

2. Objective

The main objective in designing the system was to obtain a scalable framework running on a MS Windows XP platform, which could be easily used and extended. The term “scalable” means that signal acquisition should be possible for an arbitrary number of channels synchronously using standard components (no dedicated hardware); extendibility means that new filter algorithms or array algorithms should be easy to integrate into the system using one standard programming interface. Also, there should be a means to modify filter parameters

intuitively and visualize/analyse arbitrary signals via the graphical user interface (GUI). In total, the following functionality is required:

- System shall work in real time
- Arbitrary number of input and output channels
- Arbitrary number of filters
  - Filters shall be controllable via the GUI
  - New filters shall be easy to add using a dedicated programming interface (API)
- Arbitrary signal routing between input/output channels and filters
- Multi-channel recording and replay
- Time-domain analysis functions (oscilloscope)
- Frequency-domain analysis functions (FFT, frequency response)
- Parametrization tool for microphone arrays

The hardware setup used is shown in fig. 1. The microphones used are beyerdynamic MCE 60 with spherical sensitivity characteristics; the microphone signals are pre-amplified and converted to optical digital format (ADAT) by an RME Octamic-D module and fed into the PC via an optical PCI bus card interface. Monitor or sound source signals are output via another PCI audio interface, in our case an M-Audio Delta 1010 module. The software described here uses entirely standard Windows I/O multimedia API calls (API = application programmer's interface) and could hence also work with any other audio I/O hardware.

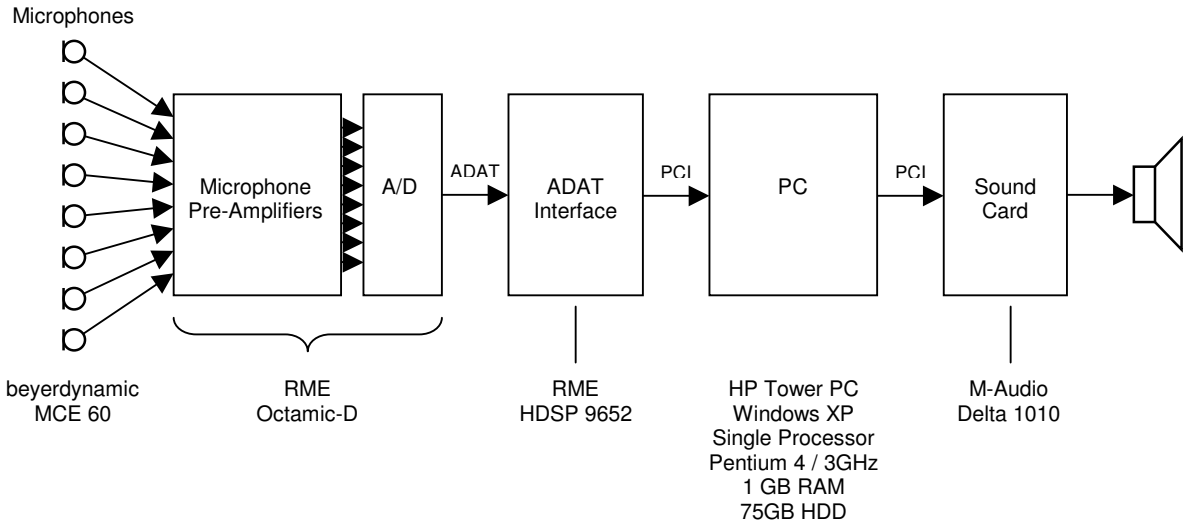

Figure 1: Hardware setup. The component types given were used at our institute but are not compulsory for using the developed software.

3. Solution

Since an MS Windows system by standard maps audio hardware – regardless of its channel number – as a set of 2-channel stereo devices, the basic concept of the audio data handling core is to handle and synchronise the data flow from/to a given subset of the present stereo devices (fig. 2). For this, a multithreaded class hierarchy was developed which allows selection of the desired devices (and thus, channels) and which cares for the synchronisation of all devices selected. All data streams are converted to double precision floating point format, which is used by the filter algorithms for internal data processing. Data is made

available to the filter algorithms by `CDataSourceBuffer` objects; output data is written to `CDataSinkBuffer` objects. These objects can be linked mutually in order to "wire" filters and audio devices, thus routing the signals in any desired way (see fig. 4).

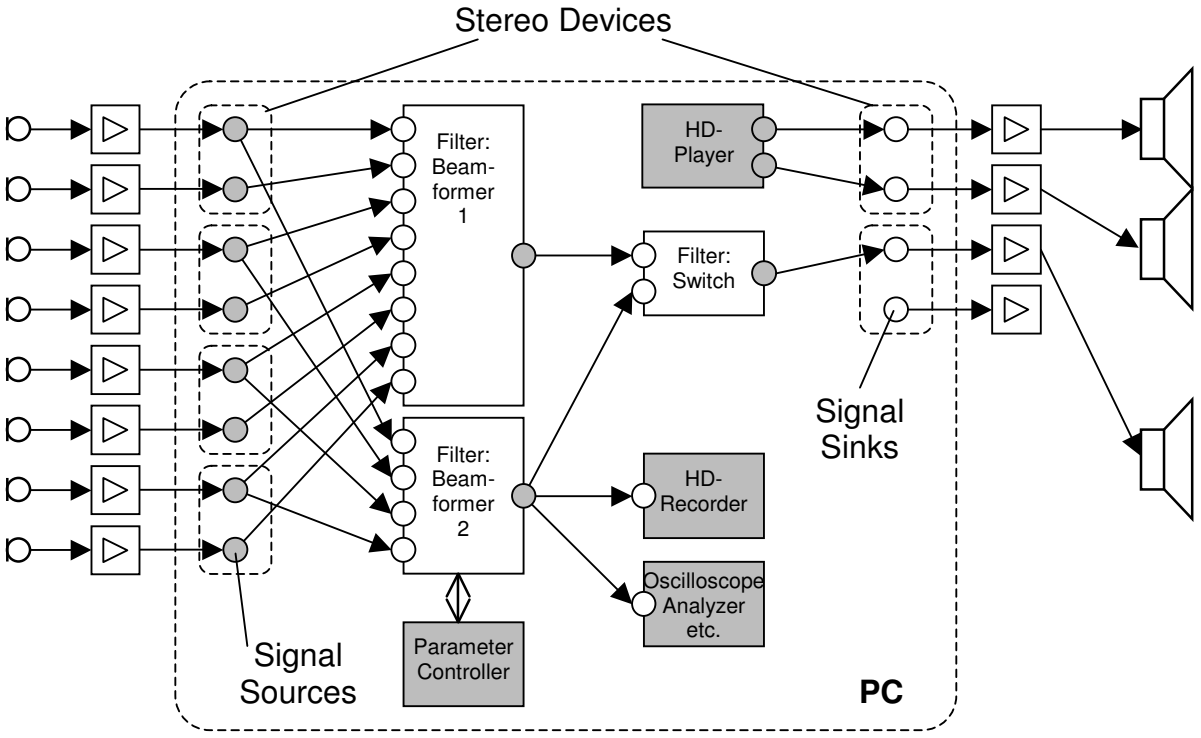

Figure 2: Signal flow (example). Input and output channels are bundled by the OS and accessible through stereo devices; data source and data sink objects are used to route the signals freely between I/O channels and filters. The data format used inside the framework is floating point, double precision.

The programming languages used for the described system are shown in fig. 3. For the real-time processing part, it was considered necessary to give the programmer the full control over memory allocation and data type handling. Thus, native C++ was chosen. This is also a good choice as most of the available high-performance mathematical libraries (which might be necessary for implementing a certain filter algorithm) can be used directly with C++ while a .NET interface is not yet available. On the other hand, implementing the GUI in C#.NET proved a good choice as this language offers a very flexible way for programming MS Windows applications: Both the garbage collection mechanism and the windows class library allow for a very convenient design of the GUI framework, while the concept of the C# language itself reduces the effort of implementation and code changes. As development environment, the MS Visual Studio/VC++/C# 2005 Express was used ([2, 3]).

There are two class hierarchies implemented in the C++.NET interfacing layer and the native C++ real-time layer (see fig. 4 and fig. 5). As the above and fig. 2 suggest, each two-channel audio device is managed by a corresponding `CRealtimeDeviceHandler/CRealtimeDevice` object pair. The devices used at a time are synchronized by a `CAudioProcessorKernel/CAudioProcessor` object pair. Filters are instanciated from external filter DLLs and controlled through `CFilterHandler/CFilter` objects. Corresponding to each of its input and output channels, each object handling audio data (I/O devices, filters, etc.) possesses `CSignalSinkBuffer` and `CSignalSourceBuffer` and objects, respectively, each corresponding to one input/output channels. The data flow between devices and filters is established by connecting `CSignalSourceBuffer/CSignalSource` and `CSignalSinkBuffer/CSignalSink` objects.

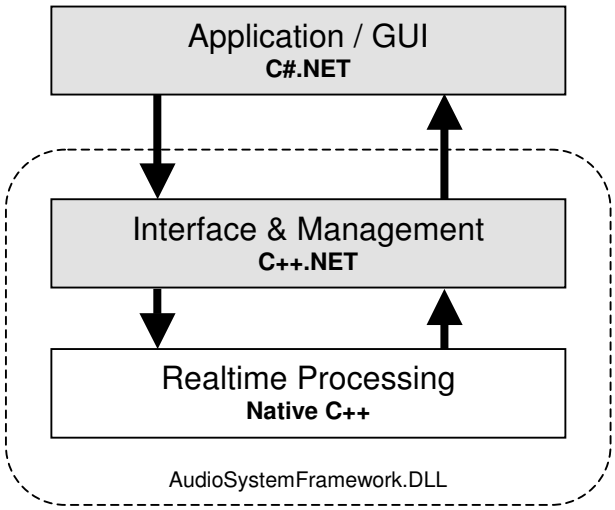

Figure 3: Programming languages used. For real-time data processing, native C++ code was chosen in order to give the programmer full control over data types, time-critical execution and memory handling. For the GUI, C#.NET was used due to its flexibility and efficiency.

```
class CUnmanagedObject;
    class CAudioProcessorKernel;
    class CAudioDataHandler;
        class CFilterHandler;
        class CRealtimeDeviceHandler;
            class CWaveAudioDeviceHandler;
                class CWaveAudioInputDeviceHandler;
                class CWaveAudioOutputDeviceHandler;
        class CHarddiskRecordingDeviceHandler;
            class CHarddiskRecorderHandler;
            class CHarddiskPlayerHandler;
    class CSignalBuffer;
        class CSignalSourceBuffer;
        class CSignalSinkBuffer;
            class CSignalMonitorBuffer;
```

Figure 4: Class hierarchy, native C++ part. Any object handling audio data is derived from the class `CAudioDataHandler`, which provides access to input and output data buffers through `CSignalSourceBuffer` and `CSignalSinkBuffer` objects. Filter algorithm DLLs are accessed through `CFilterHandler` objects while `CRealtimeDeviceHandler` objects care for the data transfer from/to the audio hardware. Currently, the system works solely based on the wave audio API, using `CWaveAudioDeviceHandler` and its subclasses; support for e.g. ASIO interfaces could be added by deriving suitable classes from `CRealtimeDeviceHandler`.

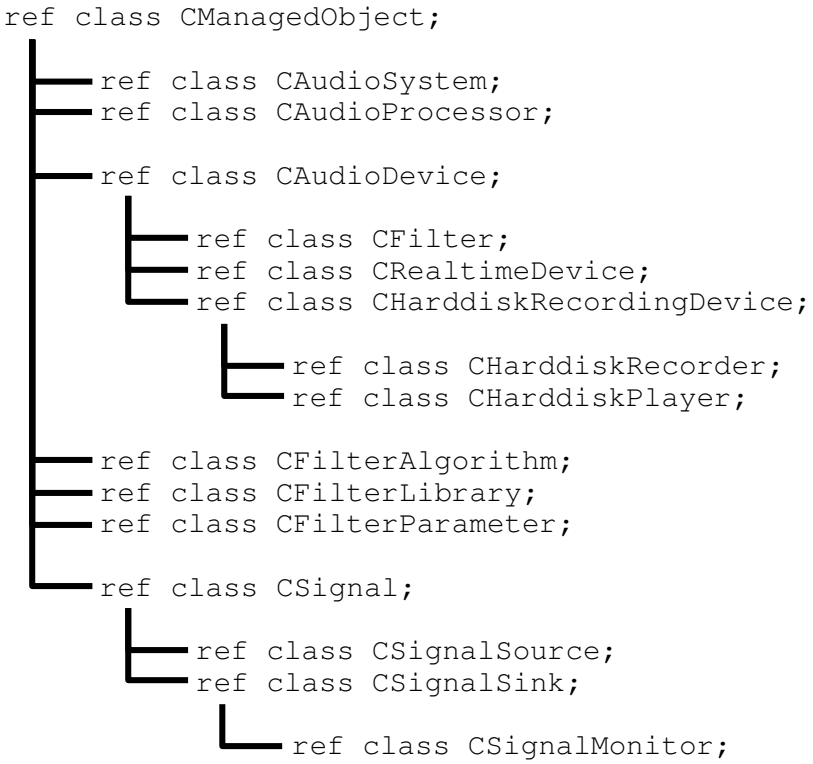

Figure 5: Class hierarchy, C++.NET part. These are the objects provided by the AudioSystemFramework DLL. The application initializes a single `CAudioSystem` object which then provides a list of all audio devices (`CAudioDevice`). Any subset of audio devices can be `selected` and activated by an `CAudioProcessor` Object. Filters can be instanciated by selecting a `CFilterAlgorithm` object from the library (`CFilterLibrary`). The created `CFilter` object can then be wired with input/output devices by connecting the respective `CSignalSource` and `CSignalSink` objects. Filter parameters can be read and modified through objects while real-time signals can be monitored/displayed by capturing data from `CSignalMonitor` objects.

4. Filter API

A filter algorithm is implemented as a single native C++ DLL which is read by the framework at runtime. This makes it possible to add new filter algorithms without modifying/recompiling the entire application. Each filter DLL provides two functions through which a filter object with a standard interface can be created and deleted. The filter object itself possesses member functions which provide information to the framework about what number of input/output channels are supported and what parameters are defined. Further, there are get/set routines for reading and modifying parameter values and the core filter function `ProcessBlock()` which does the actual filtering and will be called from the multi-threaded real-time kernel. In order to implement a new filter, all the programmer has to do is implement the following member functions:

```
void CMyFilter::Initialize( void )
void CMyFilter::SetDefaults( void )
void CMyFilter::ProcessBlock(double *pInput[], double *pOutput[])
```

`Initialize()` and `SetDefaults()` are used to initialize/reset the entire filter and restore each parameter to default settings. `ProcessBlock()` receives pointers to the input

data and output buffers for each channel and performs the actual filtering. The number of input/output channels as well as parameter values can be read from dedicated member variables.
Besides the above functions, some static data structures are also defined which are filled with basic information about the filter (such as filter name, information about what the filter does, number of input/output channels, and so on). This information is displayed to the user and is used by the GUI to build up a generic controller window through which parameters can be accessed and modified.
Once the DLL is compiled, it can be read at runtime and is available for instantiation from the Filter Library (fig. 6). It is possible to have multiple instances of the same filter; also, multiple filters from different DLLs can be active at the same time.

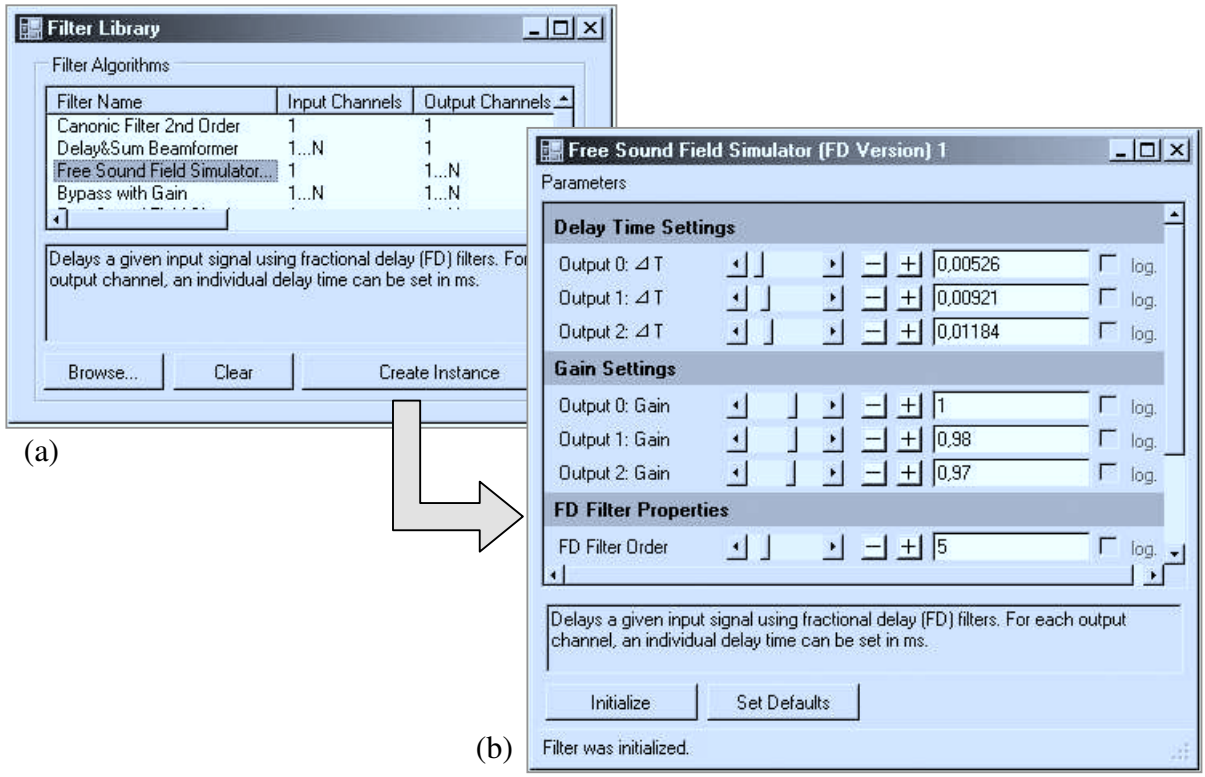

Figure 6: Filter Library (a) and generic filter control window (b). When a filter DLL is selected from the library (a), the user can select how many input and output channels the filter shall have. Then, the filter is created and a control window (b) is shown.

5. Example

On the following pages, some example screens will be shown to illustrate the functionality of the software developed. It is built around the real-time processing kernel described above, and offers a couple of operating and analysis tools. In our example, a delay-and-sum beamformer having eight channels will receive input from eight hardware input channels. For this, an Audio Processor window is opened and the necessary audio devices are selected (fig. 7). After creating the eight-channel delay-and-sum beamformer filter, the hardware input channels are connected to their corresponding filter input channel counterparts on the “Signal Flow” tab (fig. 8). Now, after pressing the “Start” button in the Audio Processor window, the microphone array can be exposed to a sound field and the output signal of the beamformer can be made audible using the output channel connected to it.

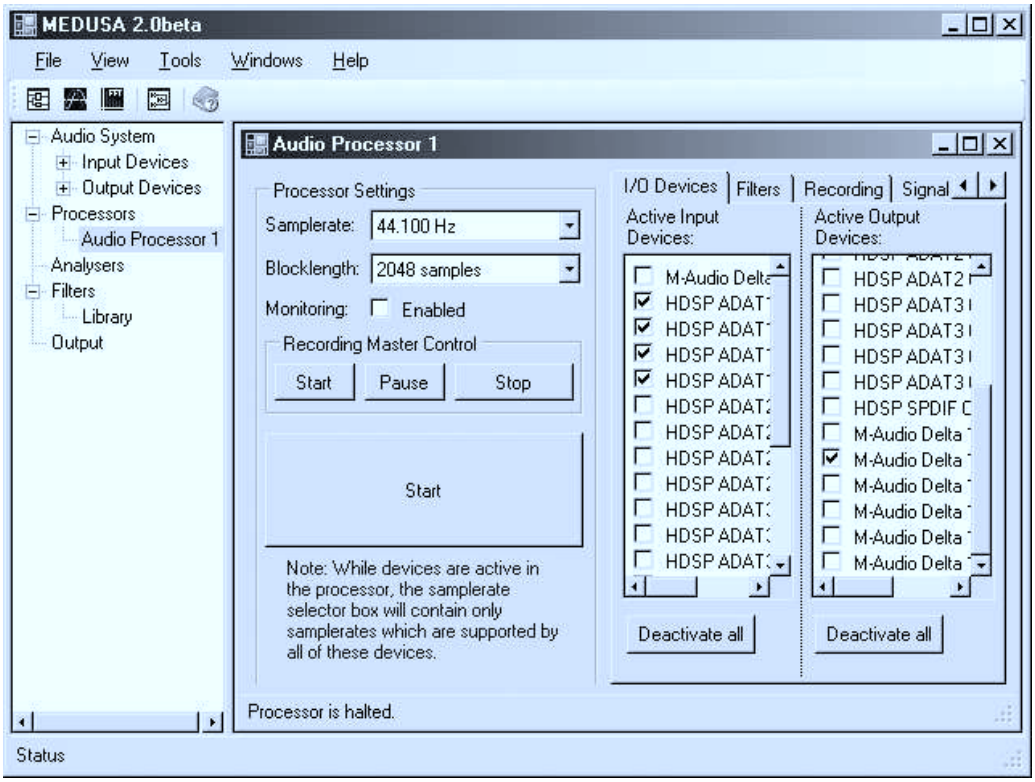

Figure 7: Application main window with Audio Processor; on the right-hand side of the Audio Processor window, four input audio devices and one output audio device are selected and thus activated. Blocklength and samplerate are selected for all active devices simultaneously.

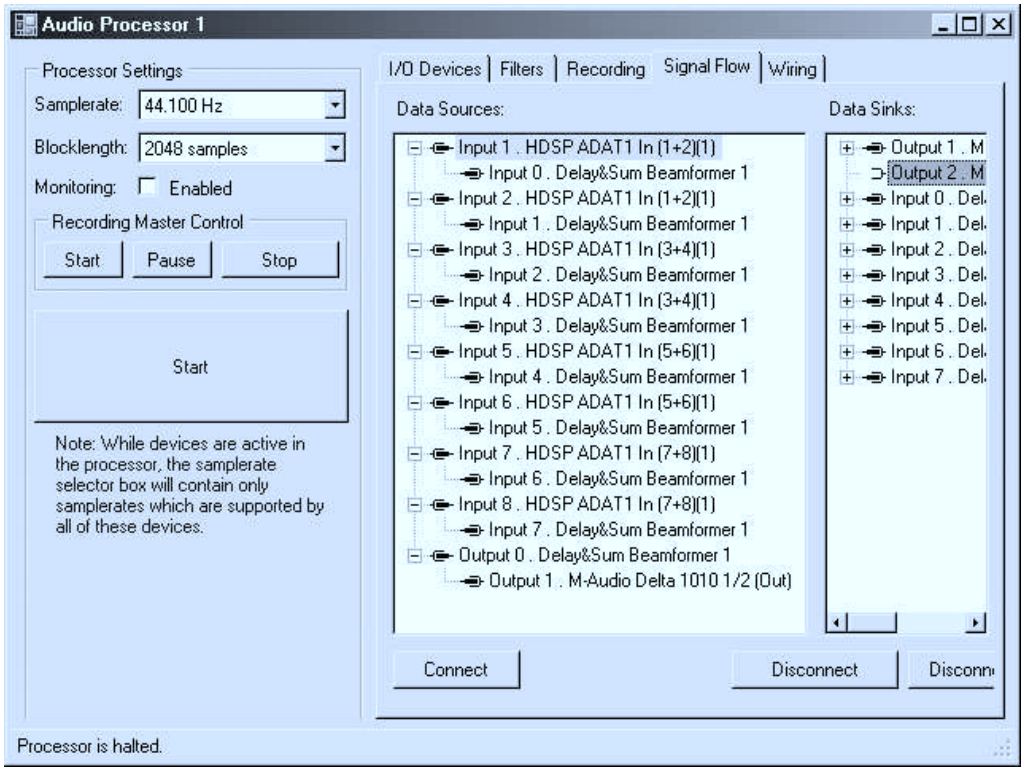

Figure 8: Connections of eight input channels to the corresponding input channels of a delay-and-sum beamformer filter. The output of the beamformer is connected to an audio output channel. Real-time processing can be started and halted by pressing the “Start” button on the left.

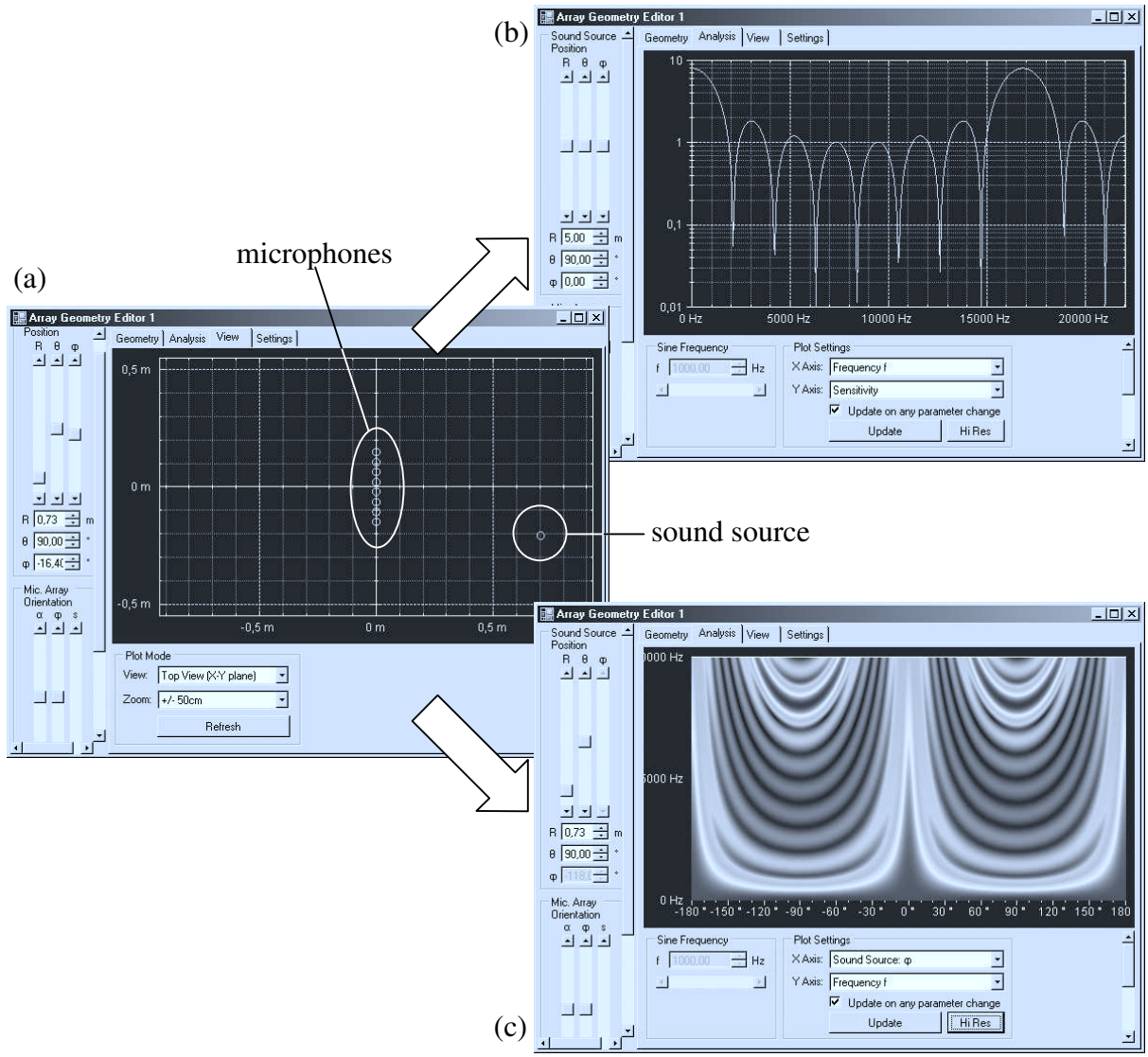

Figure 9: Microphone array simulation. (a) Geometry Editor; (b) frequency response; (c) angular frequency response.

For steering of the beamforming algorithm, a dedicated Geometry Editor is provided which allows definition of an arbitrary spatial microphone arrangement; the resulting parameters for controlling the beamformer are generated and transferred to the beamforming filter automatically (fig. 9 (a)). Thus, when any settings are altered in the editor, the result is audible in real time immediately. The Geometry Editor also calculates the frequency response of a delay-and-sum beamformer dependent on frequency, array orientation and sound source location (fig. 9 (b), (c)).

It is also possible to simulate the effect of a microphone array exposed to a free soundfield when no microphones are present: Using a suitable multichannel fractional delay filter, the wave propagation from the sound source to each microphone can be simulated ([4, 5]). In this case, a sound signal produced by a built-in signal generator is input to the multichannel-delay filter, and each of its outputs is connected to one beamformer input – instead of a microphone signal. Again, the necessary geometric information is obtained from the Geometry Editor. Now, the sound source can be moved around the microphone array in virtual space and the effect of the beamformer is made audible in real time.

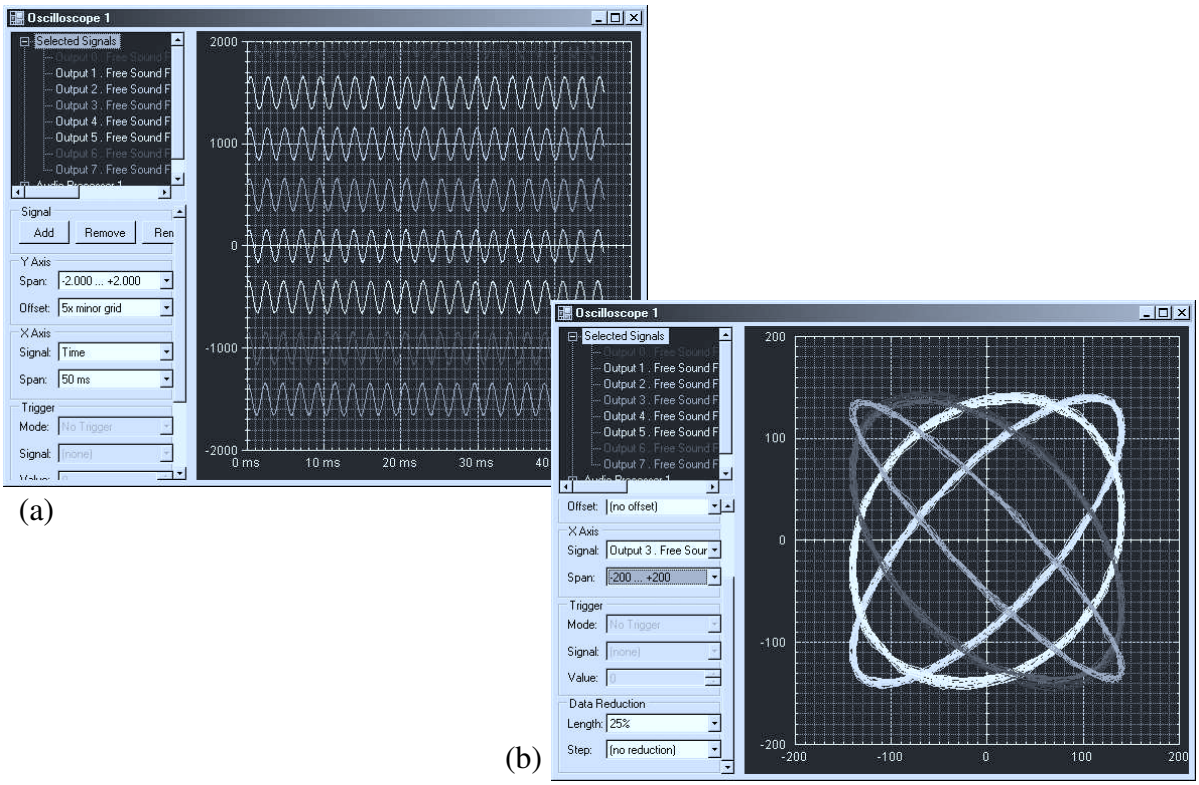

Figure 10: Time-domain analysis. (a) Oscilloscope; (b) X-Y plot. The multitude of several signals on the time axis is visualized conveniently by applying vertical offsets. Phase correlations between signals get visible by picking one signal and plotting al other signals against it in an X-Y plot.

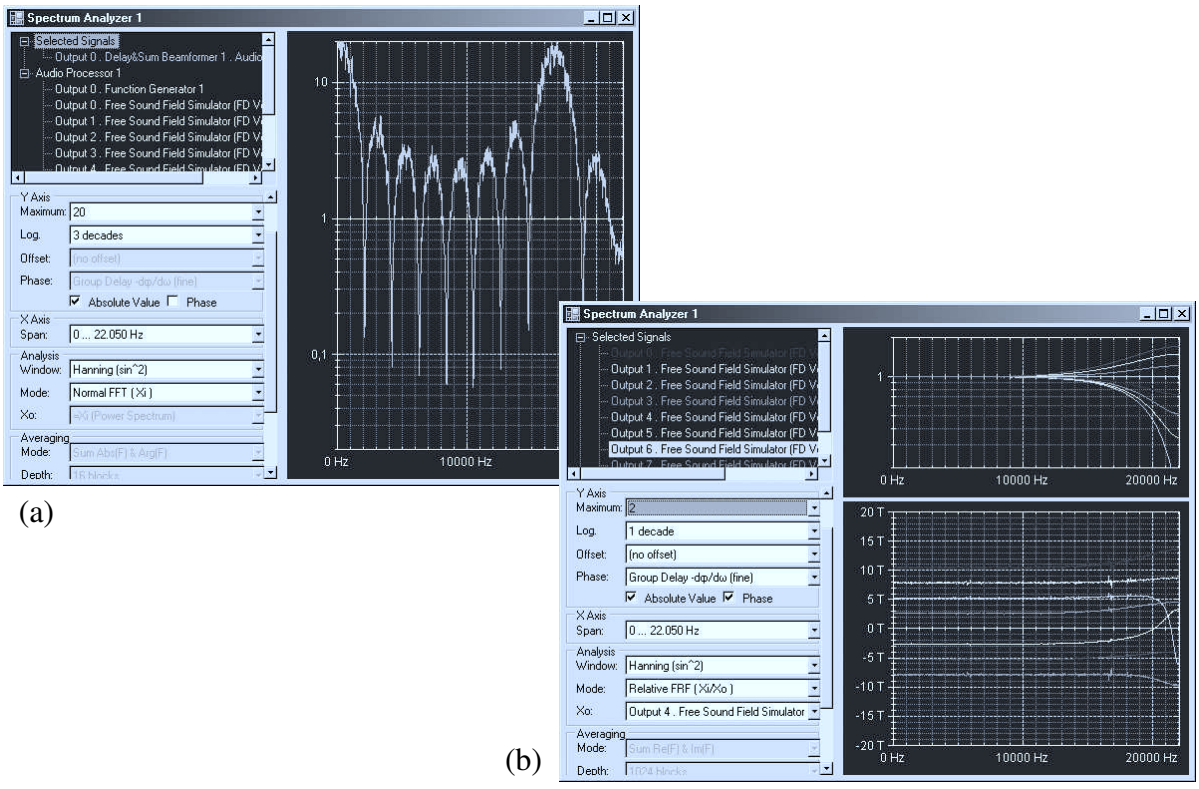

Figure 11: Frequency-domain analysis. (a) Fourier transform; (b) transfer function analysis. Cross spectra and relative transfer functions are measured by applying arithmetic operations to pairs of single-signal fourier transforms.

A variety of time-domain and frequency-domain analysis functions has also been realized which allow visual analysis of the signals being processed or being heard. These oscilloscope and spectrum analyzer functions include:

- Single-channel and multi-channel oscilloscope
- X-Y plot, plotting a group of signals versus one selected signal
- Single-channel and multi-channel FFT
- Power spectrum, cross spectrum and relative frequency response function
- Phase, phase delay and group delay measurement

Some examples are shown in fig. 10 and fig. 11. Also, Recorder and Player functions exist which allow to record multichannel-data which can then be played / replayed to one or more filter algorithms later.

6. Summary

A software system based on MS Windows XP has been developed capable of multi-channel, full-duplex and real-time audio data processing. The system is intended as a prototyping tool for real-time filter algorithm development and is suitable especially for microphone arrays. By choosing a certain software architecture and certain programming languages, special care was taken to make the system easily extensible with regard to audio APIs/drivers, the user interface and the filter algorithms themselves: C# allows for an easy extension of the GUI, while a C++ framework provides means to easily implement new filter DLLs, and also offers classes from which new handlers might be derived enabling access to ASIO or other audio driver architectures. The system is part of a research project and will be used for real-time beamforming algorithm evaluation.

7. References

[1] M. Brandstein, D. Ward (eds.): *"Microphone Arrays – Signal Processing Techniques and Applications"*, Springer-Verlag, Berlin, Heidelberg, New York, Jan. 2001

[2] A. Hejlsberg, Sc. Wiltamuth and P. Golde, *"The C# Programming Language"*, ISBN 0-321-15491-6, Addison-Wesley, 2003.

[3] G. Hogenson, *"C++/CLI – The Visual C++ Language for .NET"*, ISBN 1-59059-705-2, Springer-Verlag, Berlin, Heidelberg, New York, 2006.

[4] T. I. Laakso, V. Välimäki, M. Karjalainen and U. K. Laine, *"Splitting the Unit Delay"*, IEEE Signal Processing Magazine, vol. 13, no. 1, pp. 30-60, Jan. 1996.

[5] M. Eichler, A. Lacroix, *"Maximally Flat FIR and IIR Fractional Delay Filters With Expanded Bandwidth"*, in Proc. EUSIPCO 2007, Poznań, Poland, pp.1038-1042, Sept. 2007.