The AlgoFlex Manual

TC Group | Research

Last updated: 2009-05-05

Table of contents

1 Introduction

[Copied with minor editing from: http://wiki/bin/view/TCDesign/AlgoFlexIntroduction?skin=plain]

1.1 What is AlgoFlex?

The following is based on excerpts from the internship reports of Jean-Baptiste Berruchon (section II.3), and of Bertrand Bouzigues:

1.1.1 Goals of AlgoFlex

The first short-run goal of AlgoFlex is to provide a base for prototyping new audio processing algorithms, at the C/C++ development stage. However, on a longer run it is aimed at the real-time processing, not only locally but through a network, with the ability to share the processing between several computers. This sharing will make the system scalable. The last goal is to make it embeddable, possibly to make audio racks out of PCs. Several racks of this kind already exist, as Manifold's Plugzilla or Muse Research's Receptor.

1.1.2 Design of AlgoFlex

Block-diagram model

The user constructs a processing chain by connecting processing blocks together. The block perform audio operations like filtering, delaying, reveberating etc. The connections carry audio data between the blocks, and parameter information. These blocks are contained in separate binaries that are loaded into the main application, AlgoFlexServer. Therefore they are dynamically linked libraries. Each block is a class having the same interface as the other blocks. The server/blocks model can be seen as a host/plugins model, except that the blocks don't have the same features as traditional audio plug-in formats : the biggest difference is that they work on a sample basis.

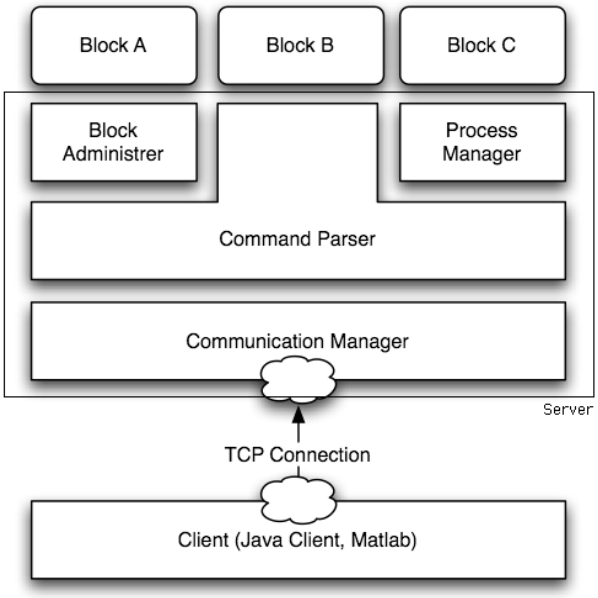
Sample-based processing

The interest of calling the process function on a sample base rather than on a buffer base is that it ensures that the block programmer will be forced to develop an algorithm that can be ported to a DSP processor, which work on a sample basis.

Client-Server model

The AlgoFlex system needs a client that gives orders to the server binary, AlgoFlexServer. The client starts by opening a connection, then can create blocks, set their parameter values, set the sample rate, then start the engine. The engine can run either for a given number of samples, either for an indeterminate period of time, ended when the client sends a Stop command. The server provide feedback to the client through text messages. The connection between the client and the server is a socket port.

Being a client-server architecture, the AlgoFlex system may be better understood by means of the figure below [BB]:



The client and the server can run on the same or on different computers. I both cases the connection between them is a TCP/IP socket.

Distributed

Several servers can be started at the same time. They each hold a part of the whole processing chain. Special blocks allow to stream data between the multiple servers. This feature is still in development.

1.1.3 Anatomy of a Block

A block can be seen as black box with audio inputs, audio outputs, parameters that can be read and set, and some parameter outputs that can send parameter events to other blocks.

Methods

The mother class of all blocks is called AlgoBase. It contains many methods managing the block input/output connection and parameter. The programmer of a block needs only to define a small set of methods:

- **SignalProcess** : this is the one which does the actual job, by taking one multichannel sample at the input, processing it and putting it at the block's output.
- **IsReady** : between the creation of the block and the moment the client application gives the starting order, the engine is not running. Before starting the engine, the server checks that all the blocks are ready with this method. This is where parameter coherency checkings are done, and sometimes resources allocated.
- **Start** : the server informs the block that the processing will start with this method.
- **Stop** : the block releases here the resources allocated in IsReady and Start.
- **ConfirmSampleRate** : this is called before IsReady to check that all the blocks are expecting to work at the same sample rate.
- **WaitForNextSample** : this is called before each SignalProcess. This allows the block to hold the engine when the block is synchronized on some external event. This is specially intended for block communicating with the sound card.

Beside these methods, the block programmer is asked to give callbacks that the server calls when the value of a parameter has to be changed.

(end of excerpts)

1.2 AlgoFlex selling points

The characteristic features of AlgoFlex are:

- Feature set according to real needs
    - ”customers” = researchers, developers, product managers
- Rapid prototyping
- Predictable path to product implementation
- Flexible – reusable processing-blocks
- Scalable – one or more CPUs
- Client/Server architecture – separate GUI from signals
- Portable (Windows and Linux platforms – possibly others)
- Integration with high-level development tool: MATLAB
- No licensing cost

1.3 About this manual

Prior to the birth of this manual (Sept. 2008), the available AlgoFlex documentation was scattered over TWiki pages, online help texts, test/demo scripts, and various documents – or only in the heads of the creators of AlgoFlex. It was therefore quite difficult for new AlgoFlex users/developers to find the information they needed. This Manual is an attempt to remedy this situation.

Certain chapters in this manual begin with a link to the TC Wiki. In some cases the Wiki page is the ‘reference’ i.e., the Wiki page is updated first, and then mirrored into the manual occasionally; in other cases the Wiki text has been copied into the Manual to be edited and merged with other text.

This document is work-in-progress. So **please do *not* just print the manual**, without careful consideration! ;-)

This manual was written by Esben Skovenborg, except for chapters on AFGUI by Søren H. Nielsen; the TWiki pages may have other contributors as well.

2 Tutorial – Getting Started

CONCEPTS:

- **block**: a processing-block with audio-inputs and -outputs, parameters, and a help-text
- **audio connection**: a ‘cable’ between an output-channel of some block and an input-channel of another block; an output-channel can be connected to many inputs, but an input-channel can only be connected to one output.
- **algorithm**: one or more blocks connected together, with suitable values for the parameters of the blocks
- **execution-sequence** (or exe-sequence): the order in which the blocks are scheduled; every block of an algorithm must be present in the exe-sequence exactly once
- **sync-block**: a special kind of block which can ‘synchronize’ the AlgoFlex server to some clock source, such as an audio-i/o device; an algorithm needs a sync-block for real-time processing
- **data-parameter**: a parameter of a block that always has a value, which is an AFData object.
- **transmitter-parameter**: a transmitter-parameter generates or transmits parameter objects (i.e. events), and does not retain a value or ‘state’.
- **parameter connection**: one block’s transmitter-parameter is connected to another block’s data-parameter.

2.1 Using AlgoFlex with MATLAB

Getting started usually just involves running the AlgoFlex Installer, and calling SetupAlgoFlex.m from your MATLAB (SetupAlgoFlex.m is found in the AlgoFlex\Client directory). For a full description on how to install AlgoFlex, see the appendix.

The following 2 examples demonstrate many of the AlgoFlex basics. For a full description of each available command, see the appendix.

EXAMPLES:

- AlgoFlex/client/DemoScripts/FilterTestScript.m
- AlgoFlex/client/DemoScripts/GliderGain.m

2.2 A filter example

```
% File: FilterTestScript.m
%
% This script demonstrates some basics of the AlgoFlex client/server system.
% An IIR filter block is created and used to filter an impulse, and the response
% is then 'downloaded' into MATLAB and verified.
%
% Demonstrates:
% - Creating, connecting, configuring blocks in an algorithm
% - Batch processing (non-realtime simulation)
% - Designing an IIR filter in MATLAB, simulating it in AlgoFlex

% Esben Skovenborg / Lars Arknaes, 2005; revised 2008

% ---------------------------------------------------------------------------
```

Connect to an AlgoFlex server

```
ServerIpName = 'localhost';   % name of the PC on which the AlgoFlex server is
running
srvId = AlgoFlexClient('OpenServCon',ServerIpName,4242)  % 4242 is the default
port
```

```
Now connected to the AlgoFlex server: localhost (port 4242)
srvId =
     6
```

Get a list of common server commands

```
AlgoFlexClient(srvId,'Help')
```

```
ans =
COMMON SERVER COMMANDS:
    [blockID,blockName] = Create(libName, nIn, nOut, blockName)
...

                          SetData(blockID|blockName, parmName, data)
                [data] = GetData(blockID|blockName, parmName)
...
                          SetSamplerate(fs)
       [expandedExeSeq] = SetExeSeq([blockID1,blockID2,...,blockIDn] |
[blockName1,...,blockNamen])
                          SetExeRateFactor(blockID|blockName, factor)
...
                          ConnectAudio(srcBlockID,srcPort, dstBlockID,dstPort)
...
                          (in the above 6 commands, a blockName can be used instead of a
blockID)

                          Start()
                          Start(timeSecs)
                          Start(timeSecs,isBlocking)
                          Stop()

...
            [infoText] = BlockInfo(blockID|blockName, verbosity)  optional verbosity = 1 (all),
2, 3
            [infoText] = Help(libName)
            [infoText] = Help()
      [data1,data2,...] = Echo(data1,data2,...)
                          Quit()
```

Get the help-text of the blocks we need

```
AlgoFlexClient(srvId,'Help','MatrixPlayer')
AlgoFlexClient(srvId,'Help','IirFilter')
AlgoFlexClient(srvId,'Help','MatrixRecorder')
```

```
ans =
BLOCK LIBRARY: MatrixPlayer
```

```
The Matrix Player will play the n-channel audio samples supplied in a matrix.
The block has the following data-parameters:
* Samples
  The nSamples x nChan matrix.
* PlaybackMode
  The available playback modes are: single | repeat | standby.

ans =
BLOCK LIBRARY: IirFilter
The IirFilter block implements a cascade of second order IIR filters.
The block has the following data-parameters:
* Coefficients:
  To set the filter-coefficients, a 3D-matrix is used, having the dimensions:
  [nBiquads,6,nChannels].
* FlushStates:
  A boolean to specify whether the filters' state is reset when coefficients
  are updated, and at every Start.

ans =
BLOCK LIBRARY: MatrixRecorder
The Matrix Recorder will record a segment of n-channel audio samples into a matrix.

The block has the following data-parameters:
* SamplesToRecord
  The number of samples to record.
* RecordMode
  The available recording modes are: single | repeat | standby.
* Samples
  After recording, you can obtain the recorded samples in an nSamples x nChans
  matrix, using a GetData('Samples') command.
  The Samples parameter is updated at a Stop command and if the block is set in
  standby mode.
```

Create an instance of each block

```
idMP = AlgoFlexClient(srvId,'Create', 'MatrixPlayer',   0, 1, 'MyPlayer');
idMR = AlgoFlexClient(srvId,'Create', 'MatrixRecorder', 1, 0, 'MyRecorder');
idIF = AlgoFlexClient(srvId,'Create', 'IirFilter',      1, 1, 'MyFilter');
```

Connect the blocks

```
AlgoFlexClient(srvId,'ConnectAudio', 'MyPlayer',1, 'MyFilter',1)
AlgoFlexClient(srvId,'ConnectAudio', 'MyFilter',1, 'MyRecorder',1)
```

Configure the MyPlayer and MyRecorder blocks

```
n = 100;  % number of samples
x = [1 zeros(1,n-1)]';  % the impulse
AlgoFlexClient(srvId,'SetData','MyPlayer','Samples',x)
AlgoFlexClient(srvId,'SetData','MyRecorder','SamplesToRecord',n)
AlgoFlexClient(srvId,'SetData','MyRecorder','RecordMode','single')
```

Design an IIR filter, and upload the coefficients to the MyFilter block

```
[b,a] = butter(4, 0.1);  % a 4th order lowpass digital Butterworth filter
sos = tf2sos(b,a);  % convert to second-order sections
AlgoFlexClient(srvId,'SetData','MyFilter','Coefficients',sos)
```

Set the execution-sequence and sample-rate

```
fs = 44100;
AlgoFlexClient(srvId,'SetSampleRate',fs)
AlgoFlexClient(srvId,'SetExeSeq',{'MyPlayer' 'MyFilter' 'MyRecorder'});
```

Check the state of the MyFilter block

```
AlgoFlexClient(srvId,'BlockInfo','MyFilter')

ans =
```

```
BLOCK: IirFilter, version 2008-08-18.
       ID= 3, NAME= MyFilter, nIn= 1, nOut= 1, ExeRateFactor= 1
---------------------------------------------------------------------
Data-parameters for block 'MyFilter':
               Coefficients: [2 x 6] = 0.0004165992, 1, 0.0008331984, 2, 0.0004165992, 1, 1, 1,
-1.479674, -1.700964, 0.5558215, 0.7884997
                FlushStates: [1] = 1
           GLD_IirCoefGlider: (null)  *

Transmitter-parameters for block 'MyFilter':
       (none)

Probes:
       (none)

Input/output ports:
       (none)

Audio-connections for block 'MyFilter':
Inputs:
       chan. 1 <-- bl. 1 ch-1
Outputs:
       chan. 1 --> bl. 2 ch-1;
```

Now run the algorithm

By using a blocking, time-limited Start command, a Stop command is unnecessary. Note that this simulation runs faster-than-realtime (batch processing) because no Sync-block (such as an audio i/o device) is included.

```
AlgoFlexClient(srvId,'MultiStart',length(x)/fs,true);  % time-limited, blocking

%AlgoFlexClient(srvId,'Stop')
```

Download the impulse response

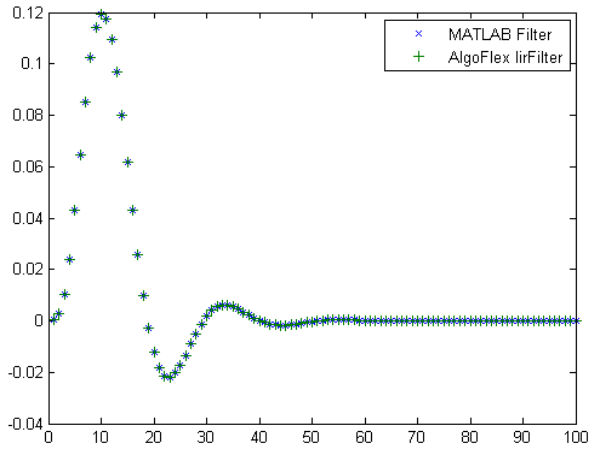
- and run the equivalent filter in MATLAB

```
resp = AlgoFlexClient(srvId,'GetData','MyRecorder','Samples')';

respML = filter(b,a,x);
```

Plot the response, and measure the difference between the AlgoFlex and MATLAB filters

```
plot([1:n],respML,'x', [1:n],resp,'+')
legend({'MATLAB Filter','AlgoFlex IirFilter'})
disp(['Mean abs difference: ' num2str( mean(abs(respML-resp)) )])
```

```
Mean abs difference: 7.6841e-016
```



Delete the blocks, with connections, parameters and everything

```
AlgoFlexClient(srvId,'DestroyAll')
```

Finally, close the server

```
AlgoFlexClient(srvId,'Quit')
AlgoFlexClient('CloseServCon',srvId)
```

Closing connection to AlgoFlex server: localhost (port 4242)

2.3 A real-time processing example

```
% File: GliderGain.m
%
% This script creates an algorithm to playback an audio-file through a Gain
% block into an ASIO output. The coefficient-glider in the Gain block is
% then enabled, and audible.
%
% Demonstrates:
% - AudioIOWiz, a function to help configure audio-i/o blocks
% - Saving/restoring (sub-)algorithms via XML
% - Real-time processing
% - Coefficient-gliders

% Esben Skovenborg, Sept. 2008

% ---------------------------------------------------------------------------
```

Connect to an AlgoFlex server

```
srvId = AlgoFlexClient('OpenServCon','localhost',4242);
```

```
Now connected to the AlgoFlex server: localhost (port 4242)
```

Create and configure an audio-I/O block (stereo)

```
% Use the wizard to configure your audio-I/O. You only need to do this once to
% generate the two XML files: diagFilename and parmsFilename.
if ~exist('diagFilename','var')
   [diagFilename,parmsFilename] = AudioIOWiz(srvId, fullfile(pwd,'MyAudioIO'))

% Alternatively reconstruct and configure the audio-I/O block via the XML files
else
   AlgoFlexClient(srvId,'XMLLoadDiagramFile',diagFilename);
   AlgoFlexClient(srvId,'XMLLoadParametersFile',parmsFilename);
end
```

Create a Gain block and a FilePlayer block, and ask them what they do

```
AlgoFlexClient(srvId,'Create','FilePlayer',0,2,'MyFilePlayer');
AlgoFlexClient(srvId,'Create','Gain',2,2,'MyGain');

AlgoFlexClient(srvId,'Help','FilePlayer')
AlgoFlexClient(srvId,'Help','Gain')
```

```
ans =
BLOCK LIBRARY: FilePlayer
The Audio-file Player will play the n-channel audio samples supplied in a file.
Various file types are supported, via the libsnd library.
The available playback modes are: whole | segment | standby | whole* | segment*
Asterisk is infinite loop.
Parameters are
  * InputFileName :
      the name of the file to open.
  * PlayMode :
      whole   : plays the whole file.
      segment : plays a part of the file. See SegmentBorders.
      standby : plays silence.
      whole*, segment* : loops the whole file or the segment.
  * SegmentBorders :
      use this to give the bounds of the segment. The sample on the 2nd bound
      is excluded, so that a region like [0 1] can loop on 1 second exactly. If
      the In or Out bound doesn't exactly fall on a sample, it is rounded to
      the nearest. The bounds validity is checked before Start.
      You can use the value 'Inf' to play to the file's end.
  * BufferSize : number of multichannel samples to hold in the buffer.
  * FileSize : the size of the audio file, in sample frames.
      This read-only parameter is set when the InputFileName is specified.
Transmitter parameter:
  * CurrentPosition : A value in seconds, continuously updated (1 Hz)
```

```
ans =
BLOCK LIBRARY: Gain
The Gain block will amplify or attenuate the signal in all channels with a
constant gain.

Data-parameters:
* GainDb - sets the gain, in dB.

Transmitter-parameters:
* ChannelClip - can optionally be used to detect clipping [deprecated].
```

Connect the blocks

```
AlgoFlexClient(srvId,'GetBlocks')  % optionally check what block-instances
currently exist
AlgoFlexClient(srvId,'ConnectAudio','MyFilePlayer',[1 2],'MyGain',[1 2]);
AlgoFlexClient(srvId,'ConnectAudio','MyGain',[1 2],'AudioIO',[1 2]);
```

```
ans =
    'AudioIO'    'MyFilePlayer'    'MyGain'
```

Set the execution-sequence and sample-rate

```
fs = 44100;  % must be the same as in the audio-file
AlgoFlexClient(srvId,'SetSampleRate',fs)
AlgoFlexClient(srvId,'SetExeSeq',{'MyFilePlayer','MyGain','AudioIO'});
```

Configure MyFilePlayer to play a certain audio-file

```
audiofilename = 'C:\Audio\SexDrugsRainbows-03.wav'   % <--- use your own audio
file
AlgoFlexClient(srvId,'SetData','MyFilePlayer','InputFileName',audiofilename)
AlgoFlexClient(srvId,'SetData','MyFilePlayer','PlayMode','whole*')  % play whole
file, looped
```

```
audiofilename =
C:\Audio\SexDrugsRainbows-03.wav
```

Check the state of MyFilePlayer

```
AlgoFlexClient(srvId,'BlockInfo','MyFilePlayer')
```

```
ans =
BLOCK: FilePlayer, version 2008-02-08.
       ID= 2, NAME= MyFilePlayer, nIn= 0, nOut= 2, ExeRateFactor= 1
-----------------------------------------------------------------------
Data-parameters for block 'MyFilePlayer':
                 BufferSize: [1] = 512
                   FileSize: [1] = 8424864
              InputFileName: [1] = 'C:\Audio\SexDrugsRainbows-03.wav'
                   PlayMode: [1] = 'whole*'
             SegmentBorders: (null)  *

Transmitter-parameters for block 'MyFilePlayer':
            CurrentPosition: (unconnected)

Probes:
       (none)

Input/output ports:
       (none)

Audio-connections for block 'MyFilePlayer':
Inputs:
       (none)
Outputs:
       chan. 1 --> bl. 3 ch-1;
       chan. 2 --> bl. 3 ch-2;
```

Now run the algorithm

- using a non-blocking, time-unlimited start command.

```
AlgoFlexClient(srvId,'MultiStart');  % non-blocking, time-unlimited
```

Try setting different parameter-values while listening to the algorithm's output

```
% Set a new amplification-values, and listen to it glide...
AlgoFlexClient(srvId,'SetData','MyGain','GainDb', -9)
% ...
AlgoFlexClient(srvId,'SetData','MyGain','GainDb', +3)
```

Now set some non-default glider-properties

- and repeat the above step, to hear the gliding.

```
AlgoFlexClient(srvId,'SetData','MyGain','GLD_GainFactor', {'Linear','3','1e-6'})
% linear mode, 3 secs
AlgoFlexClient(srvId,'BlockInfo','MyGain')  % check the state
```

```
ans =
BLOCK: Gain, version 2008-09-10.
       ID= 3, NAME= MyGain, nIn= 2, nOut= 2, ExeRateFactor= 1
-----------------------------------------------------------------
Data-parameters for block 'MyGain':
             GLD_GainFactor: [3] = 'Linear','3','1e-6'
                     GainDb: [1] = 3

Transmitter-parameters for block 'MyGain':
       (none)

Probes:
       (none)

Input/output ports:
       (none)

Audio-connections for block 'MyGain':
Inputs:
       chan. 1 <-- bl. 2 ch-1
       chan. 2 <-- bl. 2 ch-2
Outputs:
       chan. 1 --> bl. 1 ch-1;
       chan. 2 --> bl. 1 ch-2;
```

Stop the algorithm

(which would otherwise keep running forever :)

```
AlgoFlexClient(srvId,'Stop')
```

You can also control the parameters and Start/Stop, via GUI components in the AFGUI application

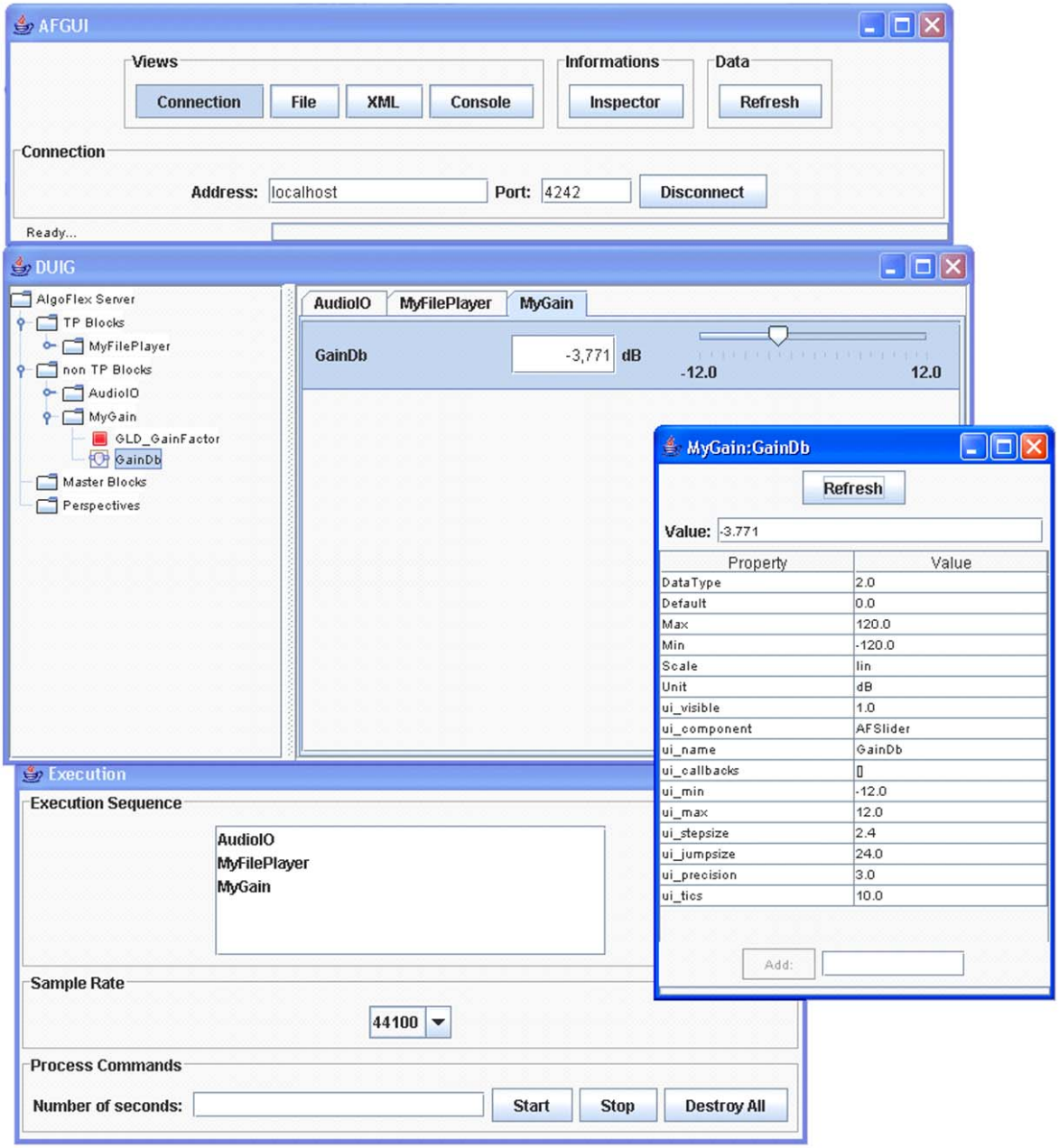
```
AlgoFlexClient('CloseServCon',srvId)  % first disconnect the MATLAB client

% --> Now start the AFGUI application, connect to the running server, and switch
to the DUIG pane!
```

```
Closing connection to AlgoFlex server: localhost (port 4242)
```

2.4 Running an Algorithm from AFGUI

The following screenshot shows the AFGUI application. The application connected to the server with the algorithm created in the previous section (GainGlider). Using AFGUI, the gain-parameter can be adjusted in real-time, the server can be started/stopped, the Inspector can show the state of blocks, etc.

The AFGUI program can also be used on a PC without a MATLAB installation, when given an algorithm in its XML form.



AFGUI is typically used for the following tasks:

- Loading the algorithm

- Loading the parameters
- Running the algorithm on the server
- Adjusting parameter-values, -ranges and mappings
- Saving a modified preset

Section 4 describes how to auto-generate a GUI, set new parameter mappings, etc.

2.5 Batch processing vs. real-time processing

Note that the only difference between batch-processing and real-time simulation is the including of some **sync-block**, in the latter case. Batch-processing may happen faster or slower than real-time depending on the complexity of the algorithm and the CPU-power available. Generally, time-limited execution is used in batch-processing (otherwise it can be difficult to know when to stop the server).

Most of the steps performed in the FilterTestScript demo can in fact be performed automatically, using the MATLAB function **AlgoFlexEval** (see Special Topics).

The following sync blocks currently exist:

- DummySync – syncs to the pc's timer
- AsioStream – real-time audio i/o (syncs to audio device)
- AudioStream - real-time audio i/o (syncs to audio device)
- TcpSource and TcpDestination – syncs one algorithm to another, across a network

2.6 Execution-sequence, upsampling and downsampling

```
% File: ExeSeq-demo.m
%
% This script demonstrates how the execution-sequence depends on the
% execution-rates of the individual blocks.

% Esben Skovenborg, Sept. 2008

% ---------------------------------------------------------------------------
```

Connect to an AlgoFlex server

```
ServerIpName = 'localhost';
srvId = AlgoFlexClient('OpenServCon',ServerIpName,4242);
```

```
Now connected to the AlgoFlex server: localhost (port 4242)
```

Create some block-instances to experiment with

```
AlgoFlexClient(srvId,'Create', 'Bypass', 0, 0, 'A');
AlgoFlexClient(srvId,'Create', 'Bypass', 0, 0, 'B');
AlgoFlexClient(srvId,'Create', 'Bypass', 0, 0, 'C');
AlgoFlexClient(srvId,'Create', 'Bypass', 0, 0, 'D');
```

Set the execution-sequence

```
AlgoFlexClient(srvId,'SetExeRateFactor','A',1);  % default
AlgoFlexClient(srvId,'SetExeRateFactor','B',1);  % default
AlgoFlexClient(srvId,'SetExeRateFactor','C',1);  % default
AlgoFlexClient(srvId,'SetExeRateFactor','D',1);  % default

% The SetExeSeq command returns the "expanded execution-sequence":
expandedExeSeq = AlgoFlexClient(srvId,'SetExeSeq',{'A' 'B' 'C' 'D'})
```

```
expandedExeSeq =
    'A'    'B'    'C'    'D'    '[TERM]'
```

Set different exe-rates for some blocks: Upsampled relative to base-rate

```
AlgoFlexClient(srvId,'SetExeRateFactor','B',2);  % upsampled
AlgoFlexClient(srvId,'SetExeRateFactor','C',2);  % upsampled

expandedExeSeq = AlgoFlexClient(srvId,'SetExeSeq',{'A' 'B' 'C' 'D'})
```

```
expandedExeSeq =
    'A'    'B'    'C'    'D'    'B'    'C'    '[TERM]'
```

Set different exe-rates for some blocks: Downsampled relative to base-rate

```
AlgoFlexClient(srvId,'SetExeRateFactor','B',1/2);  % downsampled
AlgoFlexClient(srvId,'SetExeRateFactor','C',1/2);  % downsampled

expandedExeSeq = AlgoFlexClient(srvId,'SetExeSeq',{'A' 'B' 'C' 'D'})
```

```
expandedExeSeq =
    'A'    'B'    'C'    'D'    '[SAMP]'    'A'    'D'    '[TERM]'
```

Set different exe-rates for some blocks: Mixing downsampling-factors

```
AlgoFlexClient(srvId,'SetExeRateFactor','B',1/2);  % downsampled, factor 2
AlgoFlexClient(srvId,'SetExeRateFactor','C',1/3);  % downsampled, factor 3

expandedExeSeq = AlgoFlexClient(srvId,'SetExeSeq',{'A' 'B' 'C' 'D'})
```

```
expandedExeSeq =
    'A'    'B'    'C'    'D'    '[SAMP]'    'A'    'D'    '[SAMP]'    'A'    'B'    'D'
'[SAMP]'    'A'    'C'    'D'    '[SAMP]'    'A'    'B'    'D'    '[SAMP]'    'A'    'D'
'[TERM]'
```

3 Block-Author's Guide

This section is a guide for *block-authors*, introducing the block API and various *best practices*.

3.1 Creating a new AlgoFlex block

The following procedure is the fast-track to start developing a new processing block:

1. Install the VC++ 8 compiler (i.e. Visual Studio 2005 Express/Pro with SP1, and Windows Platform SDK)
2. Install the AlgoFlex Block SDK via the AlgoFlex Installer (svn://svn/AlgoFlex/trunk/Installer/bin/AlgoFlexInstall.exe)
3. From MATLAB, call the function `mkblock.m` (located in AlgoFlex\AlgoFlexBlocks) to create a new project for your new block, based on one of the standard blocks as 'template' (e.g. the simple Gain block is good to use as template). This will create both the C++ source files and a corresponding Visual Studio project file, based on the selected 'template' block.
4. The new block can then be built, using the newly created project (the .vcproj file); of course you might need to edit the code, to get the block to do what you want :)

Before starting to implement a new block, is it generally a good idea to consider what parameters it should provide. What is the simplest set of parameters which is sufficient to configure the block, for all its intended purposes? Try to choose appropriate names for the parameters, as these are difficult to change later when the block is in use. Finally, note that the block's data-parameters should collectively be able to *reconstruct* the block, for instance when an algorithm is loaded via XML.

To learn about the methods of AlgoFlex block interface, have a look at the API documentation for the two main classes **AlgoBase** and **AFData** – the AlgoFlex Installer places a shortcut on your desktop (`AlgoFlex/doc/doxygen/html/annotated.html`)

You might also want to browse the commented source code of some fundamental blocks – it's sometimes easier to learn-by-example:

- AlgoFlex/AlgoFlexBlocks/Gain
- AlgoFlex/AlgoFlexBlocks/Delay
- AlgoFlex/AlgoFlexBlocks/GainMatrix
- AlgoFlex/AlgoFlexBlocks/Oscil
- AlgoFlex/AlgoFlexBlocks/ParmProdSum

3.2 Calling-sequence of AlgoBase methods

The AlgoFlex 'engine' is the part of the AlgoFlex server which handles the scheduling of the processing blocks, the parameter-callbacks and their dependencies, etc. The AlgoFlex engine calls a set of methods in every AlgoBlock, which are inherited from its base-class AlgoBase. The purpose and calling sequence of these methods is as follows:

	Method	When	What	Who
1	**AlgoBlock()**	At the construction of a new block instance.	Initialize member variables. Declare data- and transmitter-parameters and dependencies. Register probes? Register gliders? Set parameter-ranges and defaults.	Required for all blocks.
2	**Prepare()**	Called just before IsReady(), to allow the block to perform any last-minute tasks.	Prepare() is mainly needed for blocks which need to sync two or more servers, such as the TcpSource and TcpDestination blocks.	This method is optional for a block to implement.
3+	**SamplerateUpdate()**	Called one or more times, when the engine wants to start processing, *and* also called when the system's sample-rate is changed.	Blocks which only support certain sample-rates can use this callback to possibly reject the sample-rate (as the old ConfirmSamplerate-method).	Blocks that don't care what their sample-rate is don't need to implement this method. Blocks that declare parameters as dependent on SamplerateUpdate may or may not need to implement this method.
4	**IsReady()**	Called one or more times, when the engine wants to start procesing.	Check that the block is ready to SignalProcess. This method should generally not have any side-effects.	Implementation of this method is required for all blocks.
5	**Start()**	Called once, immediately before processing starts.	Initialize 'state' variables, e.g. sample-buffers and filter-states.	Optional for a block to implement. Recommended, for blocks with a 'state'.
6	**WaitForNextSample()**	Called once per sample.		Only implemented by blocks which can provide "sync".
7	**SignalProcess()**	Called once per sample.	Read input-samples, process samples, write output-samples.	Required for all blocks.
8	**Stop()**	Called once, immediately after processing stops.		Optional for a block to implement.
9	**~AlgoBlock()**	Called when a block instance is deleted.	Deallocate memory and other dynamically allocated resources.	Optional for a block to implement. Required for blocks that use dynamically allocated memory.

Please see the API documentation (or the file AlgoBase.h) for further information of these – and other – base-class methods.

3.3 Block I/O-dimension

By *i/o-dimension* we refer to the number of input- and output-channels for a block. The i/o-dimension of a block is always set via the client's Create command, which constructs a new AlgoBlock object. Each AlgoBlock type specifies the acceptable range of input and output

channels, as constants in its constructor initialiser list. A block may want to have the same i/o-dimension for all instances of the block, i.e., it cannot "re-size on creation". This is achieved by setting the minimum and maximum number of channels to the same value.

For instance, a mixer may be instantiated with any number of channels, as specified by the client. The block can learn its own i/o-dimension using the base-class methods **GetNoInputChans()** and **GetNoOutputChans()**.

Suppose a block's i/o-dimension depends on (say) the number of channels in a file which would not be known before the block actually opened the file. In this case, the block must handle a possible mismatch between its (client-specified) i/o-dimension and the data in the file, possibly by alerting the user by throwing an exception from a parameter-update callback, or by simply sending out zeros on any unused channels.

3.4 Parameters

The AlgoFlex blocks can have two kinds of parameters:

- **Data-parameter**, which has a state, and thus holds an AFData object. Data-parameters are used to exchange parameter data with the client program, via the SetData and GetData client commands. Data-parameters are also used to receive event-driven data from other blocks. A data-parameter can have one or more **parameter properties**.
- **Transmitter-parameter**, which has *no state*, and is used to transmit parameter-data to other blocks. In other words, they are event-driven "output parameters". The client may also ‘subscribe’ to one or more transmitter-parameters (e.g. meter data).

An AlgoFlex block must declare the names of parameters which it will use to exchange parameter data with the client program and/or other blocks. The block calls the methods **RegisterDataParm()** and **RegisterTransmitterParm()**, normally in its constructor, to declare its parameters. The block should not allocate any memory for a parameter – the engine will allocate and deliver AFData objects to the callback function associated with the data-parameter. The valid range, and a default value can be set using **SetDefaultParmValues**().

All parameter values are passed and stored in **AFData** objects. These objects could for example contain a scalar, a string, a string array, or a complex 3-dimensional matrix. Note that both data- and transmitter parameters use AFData objects as *data containers*. Indexing in AFData objects containing multiple elements starts at 0, as usual in the C/C++.

A data-parameter can be given a value in different ways:

- via a SetData command from an AlgoFlex client program
- via a connected transmitter-parameter, that transmits a data object
- via an XML parameter document

The block does not know which of the above methods was been used.

The actual time of the parameter-update and -callback is managed by the engine, but is not synched to a particular audio sample (time-stamped). The sequence of the SetData() calls are controlled by the client. In case the SetData() calls are based on loading an XML parameter document, the sequence is determined by the UpdateOrder parameter property.

When declaring a data-parameter, it may optionally be linked to a class variable of the pointer-type corresponding to its data. This is done by using the 2nd argument for RegisterDataParm(). When a member variable is thus linked to a data-parameter, the current parameter value can

always be read (from the block code) through the member variable. Alternatively, GetData() must be used to get the parameter value.

When a data-parameter has not been assigned any value yet, GetData() will return NULL instead of an AFData object. This can be used by the block to test whether a parameter has been **initialized**. After initialization, a data-parameter will never obtain the NULL-value again.

When the client (or another block) modifies a data-parameter's value, the corresponding AFData object is automatically updated. The block may implement one or more **parameter-update callback functions**, which must also be registered in the parameter's RegisterDataParm(). Several parameters may share the same callback function. For parameters that don't need to *react* to parameter-updates, no callback function is necessary. A callback will typically compute and update coefficients of the block, which are used in the block's SignalProcess method.

Please see the Gain, Delay, GainMatrix, Oscil, and ParmProdSum blocks, for some examples.

3.4.1 Update of parameters while processing

Typically, a subset of the data-parameters in the blocks of an algorithm, are **user-parameters**. The user will like to update these data-parameters in real time. This in turn typically updates one or more coefficients (class variables) used in the signal processing. It is often crucial that a group of variables are updated together, e.g. to avoid filter instability.

The design of AlgoFlex is such that a parameter callback function is executed completely before the **SignalProcess()** method is called. This means that coefficients can safely be updated from a callback function. Furthermore, a **SignalProcess()** method is never interrupted by any parameter callback function. So without any further handling by the block programmer, the system guarantees that the group of variables is set as a group and that it happens between two calls of the **SignalProcess()** method.

GetData() and **SetData()** may be called from the client program, and from the block itself, at *anytime*. Thus the parameter-update callback functions may also be called by the engine at anytime. The callback functions are *thread safe* - by using an efficient mutex mechanism.

3.5 Sample-rate

The sample-rate of a block is determined by the engine, via a SetSampleRate command from the client program. That is, a block can *not* request its own sample rate. Its actual sample rate is a combination of the overall sample rate which the engine uses, and any up/down-sampling (exe-rate factor) for the individual block. The up/down-sampling of individual blocks is set via client commands to the server, or by loading an algorithm from its XML representation.

The block can find its current 'local' sample-rate using **GetSampleRate()** or (equivalently) ***m_fs**. If the engine hasn't got a sample-rate yet, these calls return -1.0.

When processing data from an audio file, the sample rate may need to be changed. This is the responsibility of the client-side application. Likewise, the engine sampling-rate needs to match that of an audio-i/o device in the algorithm.

3.6 Parameter- and samplerate-dependency

When setting a data-parameter, one or more underlying variables (e.g. filter coefficients) may be updated. In many cases, setting one data-parameter has a consequence on variables set also by another data-parameter. Or the value of one data-parameter may influence the interpretation of another data-parameter. One very common dependency is that of the sample rate: Coefficients

of digital filters depend on its value, so coefficients cannot be calculated before the sample rate has been set.

A set of mechanisms in AlgoFlex enables control of such dependencies. They are based on parameter-update callback functions, and on instructions to the engine about the dependencies.

CONCEPTS:

- **callback** (or **callback function**): A method in an AlgoBlock, which is associated with a certain data-parameter, via the RegisterDataParm() call. The callback is then called by the engine whenever the data-parameter is updated. The callbacks may throw an exception if the new parameter value is illegal (in which case the engine reinstates the 'old' value, and passes the error on to the client). Callback methods must have the signature:
```
void MyCallback(const string& parmName, const AFData& dataObject)
throw(const char*);
```
- **linked parameter-pointer**: A member-variable, which is a const-pointer to the current value of some data-parameter. The parameter-pointer is associated with its data-parameter in the RegisterDataParm() call. After that, the engine will update the pointer whenever the value of the parameter changes. Initially, when the data-parameter is uninitialized, the pointer will be NULL.
- **parameter-group**: A set of data-parameters associated with the same callback. For example, the *ratio* and *threshold* parameters of a compressor block would probably be in the same parameter-group, and *attack* and *release* would probably be in another. A parameter-group may consist of a single data-parameter.

The AlgoBase method **RegisterCallbackDependency**() can be used by blocks to declare that one parameter-group depends on another. This means that the callback of the dependent parameter(s) will automatically be called, when the parameter that they depend on is updated. Furthermore, a callback will not be called before all parameters that it depends on have been initialized, i.e., have non-NULL values. This is a bit abstract, so let's have an example. The following code could be in some blocks constructor:

```
RegisterDataParm("A1", &m_A1, &AlgoBlock::AUpdate);
RegisterDataParm("A2", &m_A2, &AlgoBlock::AUpdate);
RegisterDataParm("B", &m_B, &AlgoBlock::BUpdate);
RegisterCallbackDependency(&AlgoBlock::BUpdate, &AlgoBlock:: AUpdate);
```

The above declarations state that the parameter-group AUpdate consists of the data-parameters A1 and A2, and BUpdate consists of B. Furthermore, that the parameter-group BUpdate *depends on* AUpdate.

The effect of the above declarations, are:

a) AUpdate() will not be called before both A1 and A2 have been initialized;
equivalently: both the pointers m_A1 and m_A2 point to a valid AFData object (i.e. not NULL).

b) BUpdate() will not be called before B has been initialized *and* A1 and A2 have been initialized.

c) If, after initialization, A1 or A2 is updated, the engine automatically calls AUpdate(), *and* subsequently calls BUpdate().

The RegisterCallbackDependency() can be used to establish both one-to-many and many-to-one relations. However, *cyclic* dependencies are of course not allowed.

Let's look at the **Delay** block (in AlgoFlex\AlgoFlexBlocks\Delay). This block's constructor contains the statements:

```
RegisterDataParm("DelayTime", &m_delayTime, &AlgoBlock::DelayTimeUpdate);
RegisterCallbackDependency(&AlgoBlock::DelayTimeUpdate, &AlgoBlock::SamplerateUpdate);
```

So DelayTimeUpdate now depends on SamplerateUpdate. The **SamplerateUpdate** corresponds to a special 'system' parameter-group, which contains the local sample-rate of the block. The dependency means that the engine will not call the DelayTimeUpdate() callback before the block gets its first real sample-rate, and that the engine will automatically call DelayTimeUpdate() whenever the sample-rate is changed.

Furthermore this sample-rate 'system' parameter is linked to the pointer m_fs (in AlgoBase). In other words m_fs can be used the same way as m_A1 and the other parameter-pointers above. That is why *m_fs will return the current sample-rate, in your block.

Therefore, the DelayTime-callback could look like this:

```
void AlgoBlock::DelayTimeUpdate(const string& parmName, const AFData& dataObject)
throw(const char*)
{
   m_delayLenSamps = (int)(*m_delayTime * *m_fs);
   ...
}
```

3.7 Coefficient-probes

A 'probe' – or coefficient probe – allows the client to read the current value of certain private variables inside a given block instance. This can be useful for test/verification purposes. A block's callback-functions compute and update internal coefficients in response to data-parameter updates. Probes can then be used to verify the correct behavior of these callbacks. A block declares a probe via the **SetupProbe**() call. The block will then automatically offer its probe to the engine, which again can read the probe via a client request (e.g. from MATLAB).

Note that, unlike the RegisterDataParm() family of methods, the SetupProbe may be called multiple times with the same probe. If a coefficient-block is reallocated, SetupProbe() should be called subsequently – in the same callback – to notify the engine that the coefficients in question have a new address.

```
void AlgoBase::SetupProbe(const string& probeName, const double* probeVarPtr, int
probeLen = 1, int stride = 1);
- probeName: the name of the probe.
- probeVarPtr: a pointer to the probe coefficient (or the first of a series).
- probeLen: the number of integers to read (default: 1, i.e. a scalar).
- stride: the 'step-size' between elements (default 1); stride != 1 probes non-
contiguous data.
```

EXAMPLE BLOCKS:

- AlgoFlex\AlgoFlexBlocks\Gain

3.8 Gliders

The AlgoFlex engine has built-in support for *gliders*, i.e. special coefficients which, when assigned a new value, glide towards their target value during a number of samples. Various

glider-modes have been implemented: Step, Linear, Exponential, Spline, … (see the **AFGlider** class in the API for details).

The script AlgoFlex\Client\TestScripts\GliderGain.m demonstrates how to enable a glider, by setting a non-default glider-mode from the client.

A glider in a block has **Step mode** as the default glider-mode. Step mode means that the glider jumps directly to its target value, without any gliding, i.e. the glider-coefficient behaves equivalently to a coefficient of type double.

The gliders in general-purpose blocks, such as GainMatrix and IirFilter, are never given a non-default glider-mode from the inside (i.e. from the block code). Because such blocks can be used in many different contexts, it is preferable to leave it up to the user of the block to decide which glider-mode and -time to use, if any. In contrast, for blocks with dedicated high-level parameters ("user-parameters"), it may be possible to choose a suitable glider-mode which can then be set from inside the block, such that certain gliders are in effect as soon as the block is created.

The gliding of gliders can be **disabled** globally in a running AlgoFlex server, for purposes of testing or verification. Disable gliding globally by sending the server this command prior to any block-creation:

```
AlgoFlexClient(srvId,'SetSystemParameter','GlidersEnable',0)
```

EXAMPLE BLOCKS:

- AlgoFlex\AlgoFlexBlocks\Gain – a glider for a single coefficient
- AlgoFlex\AlgoFlexBlocks\GainMatrix – gliders for a 2D-marrix of coefficients, treated as a group (same properties, scheduling)
- AlgoFlex\AlgoFlexBlocks\IirFilter – gliders for a 2D-matrix of coefficients, the dimensions of which can change dynamically on runtime.

3.9 The real-time code

The **SignalProcess()** method contains the code that will execute every sample, possibly in real-time, so special considerations apply. In general, variables which are only used from within SignalProcess() and which don't need to retain their value between samples should be declared as *local variables* of the SignalProcess() method. Using class-member variables for this can lead to slower processing (and makes the code harder to maintain).

The code in SignalProcess() should never -

- allocate/deallocate memory, or resize dynamic types such as vector
- use any operation which might throw an exception without a surrounding try/catch section
- use blocking i/o operations (sync-blocks can use the WaitForNextSample() method)
- assume a fixed (hard-coded) sample-rate

For multi-channel blocks, the following pattern is recommended (due to certain 'magic' in the Native Processing Framework):

PATTERN:

```
void AlgoBlock::SignalProcess() throw()
{
    int chan;
    double y;
```

```
        for (chan = 0; chan < GetNoInputChans(); chan++)
        {
            y = GetInputSample(chan);
            // transform y as desired
            PutOutputSample(chan, y);
        }
    }
```

3.10 Synchronisation

Some special AlgoFlex blocks can provide synchronization, by having contact with external clock-sources. The most common example is the blocks implementing real-time audio i/o. These blocks are called **sync-blocks**. A sync-block simply sets the "sync-flag" in its constructor-initialize list to true, and should implement the base-class method **WaitForNextSample()**. Non-sync blocks should *not* implement WaitForNextSample().

Even though a block cannot change its own sample rate it may be practical to be able to handle a changing input sample rate, e.g. from a digital interface. This can be done, by the client-side application, by executing a sequence of Stop(), SetSampleRate(), Start() commands. (See also Samplerate, above)

3.11 Memory management

It is OK for a block to use the heap for dynamically allocating data, i.e. via **new**. However, memory or objects should never be allocated or de-allocated in the SignalProcess-function. Data structures that depend (only) on the number of audio input/output channels should generally be allocated in the block's constructor. Data structures that depend on the value of data-parameters, should be allocated in a registered callback-function (or a private function called by one), corresponding to the parameter.

Embedded DSP systems traditionally use staticly allocated memory, corresponding to the "worst case". For instance, a delay-line will have a hard-coded maximum size, the memory for which will be staticly allocated. This static allocation principle doesn't fit well with a reusable, flexible AlgoFlex block. In most cases, it would be both difficult and restrictive to assume a maximum size of a block's data-structures. Furthermore, the overhead associated with using dynamically allocated memory does not hurt the real-time performance, because the allocation happens at construction-time (primarily) or in parameter-callbacks (secondarily).

3.12 Frequency-domain signal processing

If a block does frequency-domain signal processing, or other frame-based processing, the block itself should allocate a frame-buffer, collect the samples, and perform its computation whenever the buffer is full. Such a block will normally have latency proportional to the length of the frame-buffer. It is the responsibility of the algorithm-author to take this into account, for instance by inserting latency-compensation delays in parallel signal paths.

Note that frequency-domain signal processing can be used independently of AlgoFlex's chunk-processing feature. Chunk-processing is enabled globally in order to improve performance, at the cost of a higher latency. Regardless of whether chunk-processing is enabled, the individual block only gets one sample at the time.

3.13 Designing for reusability

When designing a new algorithm, you consider how the algorithm can be *broken down* into processing-blocks. For many common operations you can use the standard blocks, but you may also need to write a couple of new blocks. The question is then, how comprehensive should the new block be? At one extreme: code the entire algorithm as a single block and simply connect that to an audio-i/o block; at the other extreme, use a block for every basic operation (as in Simulink). When making this decision, the **reusability** of the new block is an important aspect to consider. A simple block, that does some well-defined task well, is often preferable over a more complex block that tries to do too much.

Reusable blocks –

- are continually **tested** and **optimized**
- enables **faster prototyping** of new algorithms
- **increase stability** (because the bugs are usually in *the code you just wrote*, not in the block that has been in many other algorithms)
- yield algorithms that are **easier to test**, because it's straightforward to simulate subsets of the algorithm independently

The computation-overhead of making an audio-connection between two blocks, as opposed to coding the 2 blocks as 1, is essentially no more than an assignment per channel.

The task of maintaining the audio- and parameter-connections between many smaller blocks can be considerable. This issue is being addressed with a new AlgoFlex feature: **meta-blocks**. A meta-block is a "virtual" block which consists of several real blocks that are interconnected. The meta-block can be instantiated and connected to other blocks, just like a real block. Each meta-block is defined by an XML file, similar to the diagram- and parameter-XML already implemented in the server. [Meta-blocks are work-in-progress.]

3.14 API Documentation and Block help

The classes AlgoBase and AFData provide interfaces that all AlgoFlex blocks – present and future – will use. Therefore these classes are documented with the Doxygen system, so that the semantics of all the public methods are clearly presented – even without referring to the header files.

The individual AlgoFlex block is documented, by describing its function, its audio connections and parameters, as a multi-line string which is then available to the client/server on runtime. The AlgoFlex blocks Gain.cpp and Delay.cpp can be used as examples.

3.15 Platform-neutral coding

Whenever something cannot be implemented in a platform-neutral manner, two equivalent versions of the code block should be provided: a Windows version and a POSIX (i.e. Linux) version.

PATTERN:

```
#ifdef _WIN32
```

```
    // the Windows way
#else
    // the Linux/POSIX way
#endif
```

3.15.1 Portability, language and libraries

In general, an AlgoFlex block may use any function or library that is part of the C++ Standard Library [ISO, 2003]. Platform-specific functions, such as those part of the WIN32 API or part of .NET framework should not be used, as the block would then be unusable under Linux and Mac OS.

The extensions to the C language, standardized in the C99 standard, are not yet part of the C++ standard, and are not yet commonly available on the relevant platforms. Therefore, the C99 extensions should not be used at this point. See the DummySync block, for a simple block that does require platform-specific calls, yet supports several platforms.

The AlgoFlex SDK provides some header files to make cross-platform coding easier:

- AFNumeric.h – dealing with 64-bit integers and non-finite floating-point objects etc.
- AFLinuxMacros.h – macros to provide Linux equivalents of common (but non-standard) Windows functions
- AFThreadMacros.h – macros for platform-neutral handling of thread, critical section, mutex, etc.

It is best to avoid defining constants and macros with common names, like TEST, EPS, MIN, DEBUG, PRINT, etc. The Native Processing Framework *inlines* all the blocks in an algorithm into a single module, so if several blocks have (say) a `#define TEST ...`, problems arise. If your block needs one or more constants, it is preferable to define these inside the AlgoBlock class as data members which are const static.

Moreover, we recommend using the C++ standard library functions min() and max(), as in this pattern (rather than defining your own):

```
#include <algorithm>       // for std::min and std::max

...

z = std::min(x,y);
```

4 GUI Generation, Interactive Control, and the AFGUI Application

[http://wiki/bin/view/TCDesign/AlgoFlexGUI?skin=print.pattern]

The user's manual for the automatic graphical user interface (GUI) tool for AlgoFlex (AFGUI).

4.1 Introduction

This page describes the behaviour of a graphical user interface program for AlgoFlex – AFGUI.

Common for all the GUIs is that they are constructed using a number of **components** put together in overall structures such as **windows** or **tabbed panes** or similar collecting mechanisms.

A GUI may be built either manually, automatically or as a mixture of both.

For full control, the GUI can be built manually from a client such as Matlab using appropriate commands. The Matlab user interface tool GUIDE may even be used – using the user interface elements described here. Every aspect of layout and behaviour of the components may be controlled in this way. As everything is under manual control the blocks do not need to specify any of the parameter properties which are used as guidelines for building user interfaces. Examples of such parameter properties would be "Min" and "Max". Please remember that full manual control means also lots of manual configuration.

For automatic generation of a usable GUI, certain information is needed for making meaningful choices of parameter labels, scale endpoints etc. Each block can provide this information in the form of parameter properties. By making the (small) effort of specifying meaningful parameter properties when a block is constructed, the subsequent generation of a graphical user interface for the block is eased considerably. The parameter properties may also be set using AFGUI.

Technical information of the current AFGUI is found at the AfGuiTechnical page.

An earlier version of the automatic GUI generator is decribed in the page AlgoFlexDynamicUIGeneratorSpecifications.

4.2 The AFGUI Application - Basic Features

AFGUI (AlgoFlex Graphical User Interface) is the main program used to construct stand-alone graphical user interfaces for AlgoFlex. It is a Java program and therefore requires an appropriate Java runtime environment on the machine used for the user interface. In AlgoFlex terms AFGUI is a Client, it will run on any platform with the appropriate Java installed, and it connects to the AlgoFlex Server through a network socket, remote or local. This means that it is possible to separate the uneven processor load caused by the user interface actions from the more even load of the signal processing. But of course, both Client and Server may also run on the same machine.

The behaviour and properties of the AFGUI application described here reflect the target state of the present development cycle. The design goals have primarily been set by EsbenS and SoerenNielsen – and with BertrandBouzigues as key programmer. Being work in progress, some features may not behave exactly as described here. You may want to consult the AfGuiRequests page to check whether a feature is still lacking functionality.

The system requirements for AlgoFlex itself, including AFGUI, are described in the AlgoFlexInstall page.

4.2.1 Start of the program

The AFGUI program may be started by double-clicking its icon, or from the command line by writing:

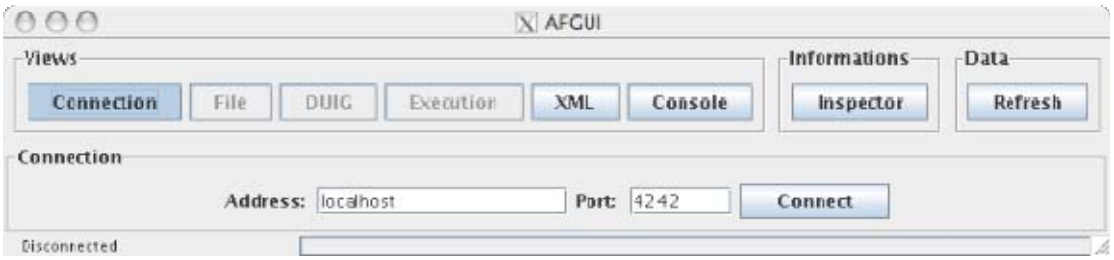
```
java -jar afgui.jar
```

in the directory containing `afgui.jar` (something like `AlgoFlex/Client/AFGUI/bin`). Alternatively, from any directory write:

```
java -jar <path_to_afgui.jar>/afgui.jar
```

A shotcut to `afgui.jar` may be put a convenient place in order to avoid writing long path names.

The AFGUI startup screen will be similar to the one shown below.



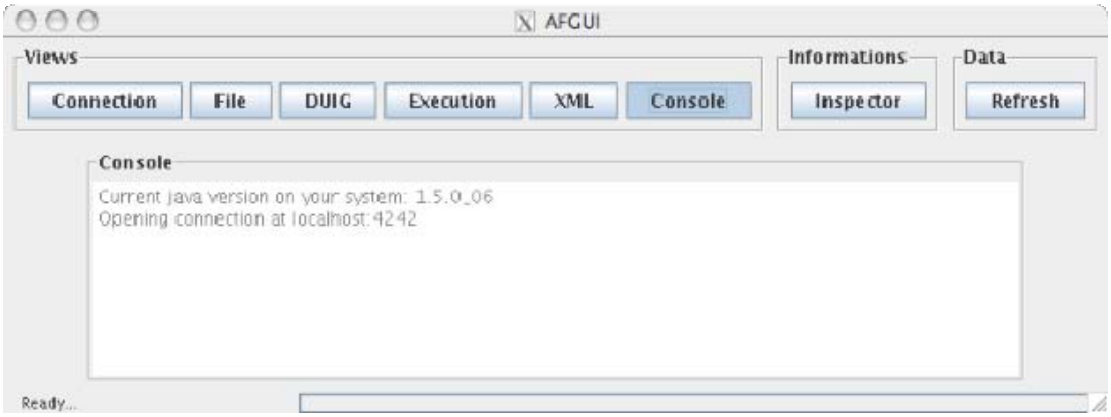
The window is divided into an upper part, which is always visible, and a lower part which is changed according to the function chosen in the **Block controls** field.

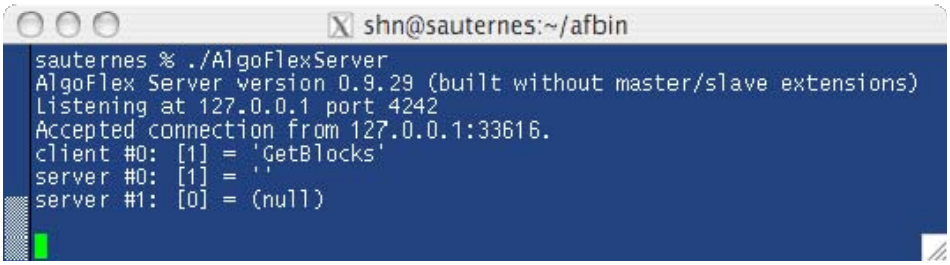
The first thing to do is to connect to the Server, using the **Connection** view. As most of the AlgoFlex functionality is implemented on the server side – even though it may look like a client side function – it is mandatory that an AlgoFlex server is running in order to use AFGUI. The server does not need to be started before the GUI, but it must be started before the **Connect** button is pressed with the appropriate settings in the Connection area of the window: Address and Port.

The AlgoFlex server is started in a command window by the command `AlgoFlexServer`. If the directory where the server executable is stored is in the search path, the command can just be typed. Otherwise, the current path must be set (cd ... or .../AlgoFlexServer). For convenience, a batch file can be used to start the server.

If the AFGUI program is running on the same computer as the server, the *address* `localhost` can be used. Either a host name or an IP address may be used. The default *port number* where the server is listening is 4242, but this can be set to another value, if required, at the startup of the *server*.

At any time the status messages of the system can be seen by pressing the **Console** button. Examples of console and server outputs are shown below:





In AlgoFlex it is possible to disconnect from the server and connect again any number of times. So for running different jobs it is **not** necessary to stop and restart the server. In case of a server running on a remote machine this can be quite convenient.

4.2.1.1 Display in separate windows

The row of buttons in the **Views** field change the view in the lower half of the AFGUI window if they are clicked upon with the mouse. That is, a left-click. Right-clicking the samme button displays the information in a separate window instead. This can be convenient for debugging. When a separate window is used for display, the corresponding button disappears from the **Views** field.

By closing the separate window the corresponding button re-appears in the **Views** field.

4.2.2 Handling block diagram and parameter settings

The settings consist of two parts:

1. The Diagram part, which contains description of the blocks used and the connections between them.
2. The Parameter (Data) part, which contains the values of the parameters, and typically also information of the scale range and type, and even hints to the preferred type of user interface component.

By separating block diagram and parameters it is easy to save and recall a number of different parameter settings – independently of the block diagram.

The format of the settings is XML – and formally described by two XML Schemas, `blockDiagram.xsd` and `blockParameters.xsd`. For normal use of AFGUI it should not be necessary to know the details of the XML format, as described by the Schemas. Some additional XML information is contained in the page AlgoFlexXML.

4.2.2.1 Diagrams

The following mechanisms exist for creating a Diagram:

- MATLAB. In many cases it is convenient to create a set of blocks and their interconnections from the Matlab client (`AlgoFlexClient.m`). The settings created in this way may be stored in XML format and afterwards be loaded by the AFGUI program.
- AFGUI. The AFGUI program allows manual construction of a Diagram from scratch if one is not already available. This is done by means of a built-in XML editor which knows the XML Schemas and thus only allows legal XML constructions. The **XML** button activates this editor.
- Hard-core XML hackers may want to create their XML Diagram files themselves. Text editors with automatic validation of the XML code (based on the Schemas) exist for that purpose, but in fact any text editor may be used.

4.2.2.2 Parameters

The settings of the parameter values and other properties can also be made from the Matlab environment. Alternatively, they can be set using AFGUI. For that purpose, an Inspector exists. Often, however, the blocks themselves contain this information. This is the preferred situation.

4.2.2.3 File handling

Settings are loaded and saved through the "view" reached by pressing the **File** button.

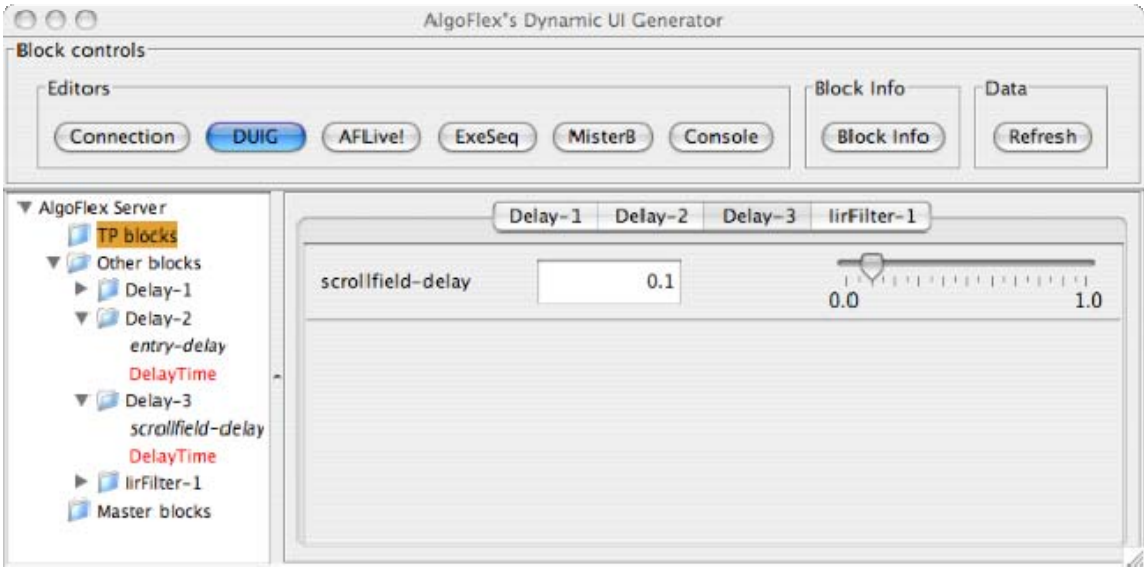
A pair of test settings is available under `AlgoFlex/Client/TestScripts/DUIG`.

The XML files describing the Diagram and the Parameters can have any names (spaces?) – but it is convenient to use the extensions `.diag.xml` and `.parm.xml` for the two types of XML files.

4.2.3 Generating a graphical user interface

When settings for diagram and parameters have been loaded or created, a GUI can be generated automatically by pressing the **DUIG** button (Dynamic User Interface Generator).

Parameters belonging to each block are shown in the same tabbed pane, in alphabetical order (for now). An example is shown below.



If a slider component is not shown, although a (horisontal) slider was specified as UI component, it is likely to be because the window is too narrow. Try dragging the window wider. The length of the slider scales with the window size.

A tabbed pane (i.e. the area containing the parameters for a block) may be converted into a separate window by right-clicking on its tab (headline). To get the tab back into the main window just close the separated window. The tabs will stay in their original sequence. BUG: They don't always at present.

If the sequence of the tabs needs to be changed, it is a simple drag and drop process: Press and hold the (left) mouse button on the tab to be moved, move the mouse pointer to the new position of the tab, and release the mouse button.

4.2.4 Tree view of blocks and parameters

In the **DUIG** view the left-hand side of the window is occupied by a tree view of the blocks and their parameters. As usual with such visualisations the branches of the tree can be unfolded (and folded in again) by pressing the little triangles (arrows) next to the block names.

The tree is hierarchical with this structure:

- Level-1 Category
- Level-2 Blocks (block instances for that category)
- Level-3 Block parameters (data-parameters and transmit-parameters for each block instance)
- Level-4 Parameter connections (to/from other data-parameters and AF-components)

In **Level 1** of the tree view, three different categories of blocks are displayed:

(blocks-with-transmit-parms (1 node), all other blocks (1 node), and individual master-components (1 node for each))

- TP blocks. These are blocks which generate data in the form of transmit parameters (TP), such as metering information. These blocks will typically also process signals. There is one node for all TP blocks.
- Other blocks. All signal processing blocks, which do not generate transmit parameters. There is one node for all ordinary blocks.
- Master blocks. The master blocks and their associated mapping functions. There is one node for each master block. This may change in the near future, see AfGuiRequests.

In **Level 2**, the blocks are listed. There is one branch for each block. The name of each block is as specified in the Diagram file.

In **Level 3**, the parameters of each block are listed. Two strings are associated with each parameter: The Name (parmName) and the Label (ui_name). If a parameter does not have any user interface component associated with it (yet) its name is written in red. Otherwise, the string shown in the tree is the Label for that parameter. A small icon displays which kind of user interface component is selected.

In **Level 4**, connections to and from parameters are listed.

In the blocks receiving parameter data the connection is described by a string of the form <transmitting block>:<parameter name>.

In the blocks transmitting parameter data, either from a TP blocks or from a Master block, the output parameter name is written in blue (ParmY).

4.2.5 The execution view

In order to run an AlgoFlex application (i.e. a collection of blocks) some further information must be given to the system. These are reached through the **Execution** view.

4.2.5.1 Execution Sequence

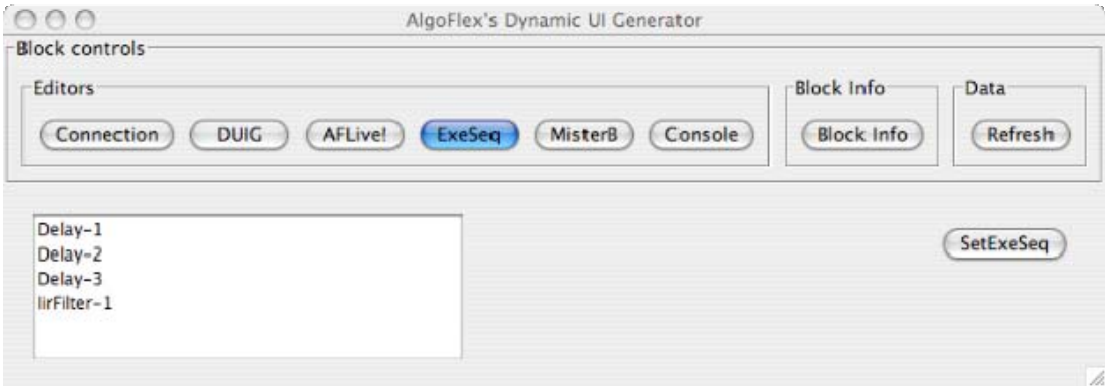
First the mass murderers, then the single murderers, and last the rest of those we really don't like

In terms of AlgoFlex processing blocks, the execution sequence determines in which sequence the individual blocks are executed. This only matters in cases where feedback loops are constructed between the blocks – or in cases where the processing latency has to be minimised. The AFGUI program automatically generates an execution sequence in case none is specified (in the XML diagram file).

The execution sequence is from top to bottom in the list. A block may be moved by first selecting it with a single mouse click, and then dragging it to its desired position (mouse movement with button pressed). The block is inserted below the one it is dropped upon (mouse button release).

Notice that only signal processing blocks appear in the list. Ancillary blocks, such as those involved in the Master Components, are actually part of the execution sequence, but as it is not necessary to edit their sequence, they are not shown in the list.

An example of an execution sequence is shown below.



The execution sequence is set automatically whenever it has been modified.

4.2.5.2 Sample Rate

In AlgoFlex, setting the sample rate is mandatory. In AFGUI it is set by selecting one of the rates in the list. Notice than even though the list already shows a rate of 44100 Hz at the first **Execution** view, the rate needs to be set. This may be seen as a bug.

4.2.5.3 Process Commands

For processing audio in real time, commands for start and stop are needed. Furthermore, the desired (maximum) processing time needs to be set before starting.

In order for a complex set-up involving Master Components and Parameter Mappers to work the execution sequence must be set, by entering the **Execution** view. Furthermore the system must have processed at least one sample. Or in more technical terms, the Start() command of each block must have been executed.

This is easily achieved by setting some sample rate and letting the application run briefly, e.g. for one second.

4.2.6 Inspector

It may of great convenience to check the present settings of a block or of a parameter. This is done by selecting the block in the **DUIG** view and pressing the **Inspector** button. The result is shown in a separate floating window.

The Inspector window can be kept open while selecting other blocks or parameters in the main window. The contents of the Inspector window will be updated automatically according to the selection.

In the Inspector window it is possible to modify almost any aspect of a parameter setting:

- The actual value of the parameter can be set.
- The properties of the parameter itself can be modified, such as Default, Unit etc. The names of these properties start with a capital letter.
- The properties relating to the display of the parameter in the present context can be modified. The names of these properties start with `ui_`.

Selection of a field in the Inspector can be done by:

- Double-clicking the field. A blinking cursor is placed at the end of the value and can be moved by the left/rigth arrow keys or by the mouse. The Backspace key will empty the field.
- Moving up and down with the arrow keys. The (invisible) text insertion point is at the end of the field. The Backspace key will delete just the last character.

Basically any value, text or number, can be entered with the Inspector. This does not mean, however, that the value entered will be meaningful. It is the responsibility of the user to ensure that the values entered make sense in the context. Only a minimum of diagnostics messages is provided (output of server in debug mode).

The new value is activated when:

- The Enter key is pressed.
- Another field is selected, e.g. by an pressing an up/down arrow key or by clicking another field.

4.2.7 Refresh

Sometimes the information in the user interface and the one actually present in the server gets out of sync. It should not, but ... Anyway, a **Refresh** button is provided in the **Data** field to update the GUI with the data as stored on the server.

4.3 Advanced features

In this section features of AFGUI for generating a more advanced user interface are described.

4.3.1 Meters

Meters are a special kind of UI component as they only receive (and display) data. They are typically controlled by a transmit parameter from a block. The meter component will typically display a bar whose length corresponds to the value of the transmitter parameter.

Alternative views are: ...

4.3.2 Perspectives

Sometimes it is convenient to collect selected parameters from one or more blocks into a single tab such they can be operated from one place with a minimum of distracting UI elements. AFGUI has a mechanism for that called Perspectives.

A Perspective is created in the tree view by right-clicking the *Perspectives* branch, and selecting *New Perspective*. A new tab with the name `Unnamed Perspective 1` is created. This name can be changed in the **Inspector**.

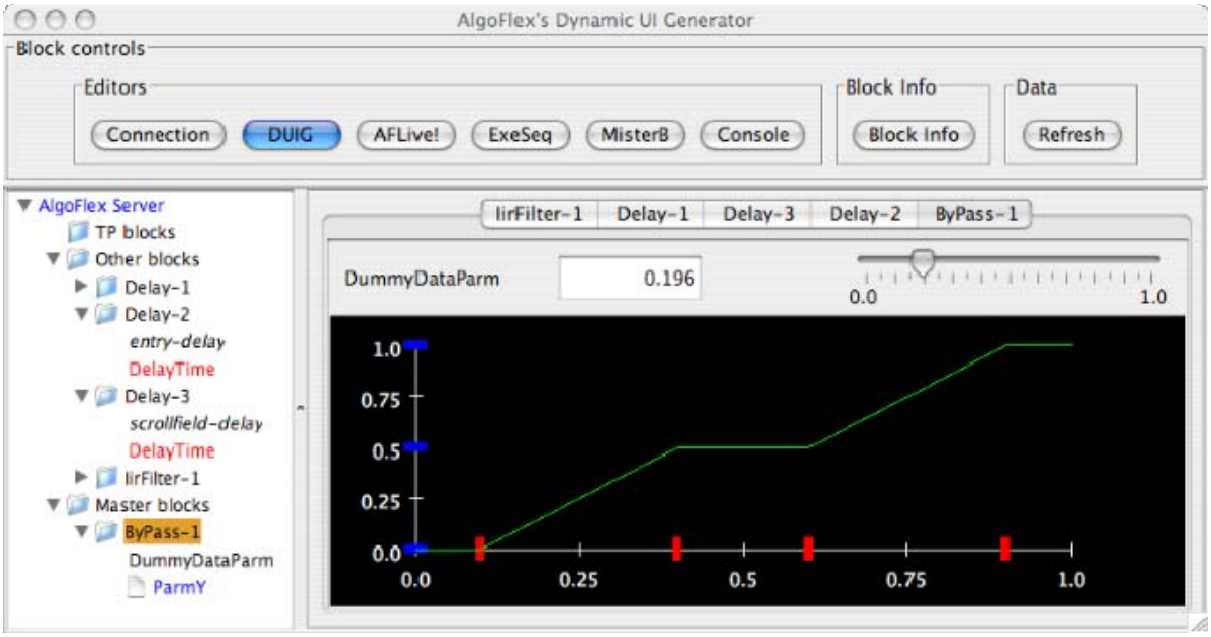
To add a parameter to a *perspective*, right-click on the parameter in the tree, and select *Add to perspective*. A list of possible *perspectives* appear. For now with the names *Bypass-1*, *Bypass-2* etc. The numerical index corresponds to the index of the *Unnamed Perspective #* name. Ultimately, the (possibly changed) names of the *perspectives* should appear in the list.

The *Perspective* mechanism works an ordinary parameters as well as master parameters/components, see the following section.

4.3.3 Master components / -blocks

Master components are used to control one or more underlying parameters through a mapping function. A master component may be part of the settings stored in the XML files – or it may be created and edited interactively.

A new master components is created by right-clicking the **Master blocks** branch in the tree view, and selecting **New Master Block**. At present, each master component is shown in a separate tabbed pane together with a mapping function. For example:



A master component may be connected to several underlying parameters using individual mapping functions. To connect a master component with a parameter drag and drop the **ParmY** parameter of the appropriate mapping function belonging to a master block onto the receiving block parameter.

4.3.3.1 Mapping function

The mapping between the master components and and underlying parameter consists (at present) of a piecewice linear function called `ThreeStepLinear`. The four corner points on the input (X) axis may be moved using the red rectangles. The corresponding output (Y) values may be moved using the blue rectangles. The changes take effect immediately.

A master components may be connected to several parameters through individual mapping functions.

The size of the mapping function graph scales with the window size.

In the future, several different mapping functions can be selected interactively. At present only the `ThreeStepLinear` function is available for interactive editing.

Furthermore, pre- and post-mappings will be available in the future. At least logarithmic and exponential transformations will be provided. This facilitates the use of a simple basic mapping function operating in a domain which for example is perceptually optimal.

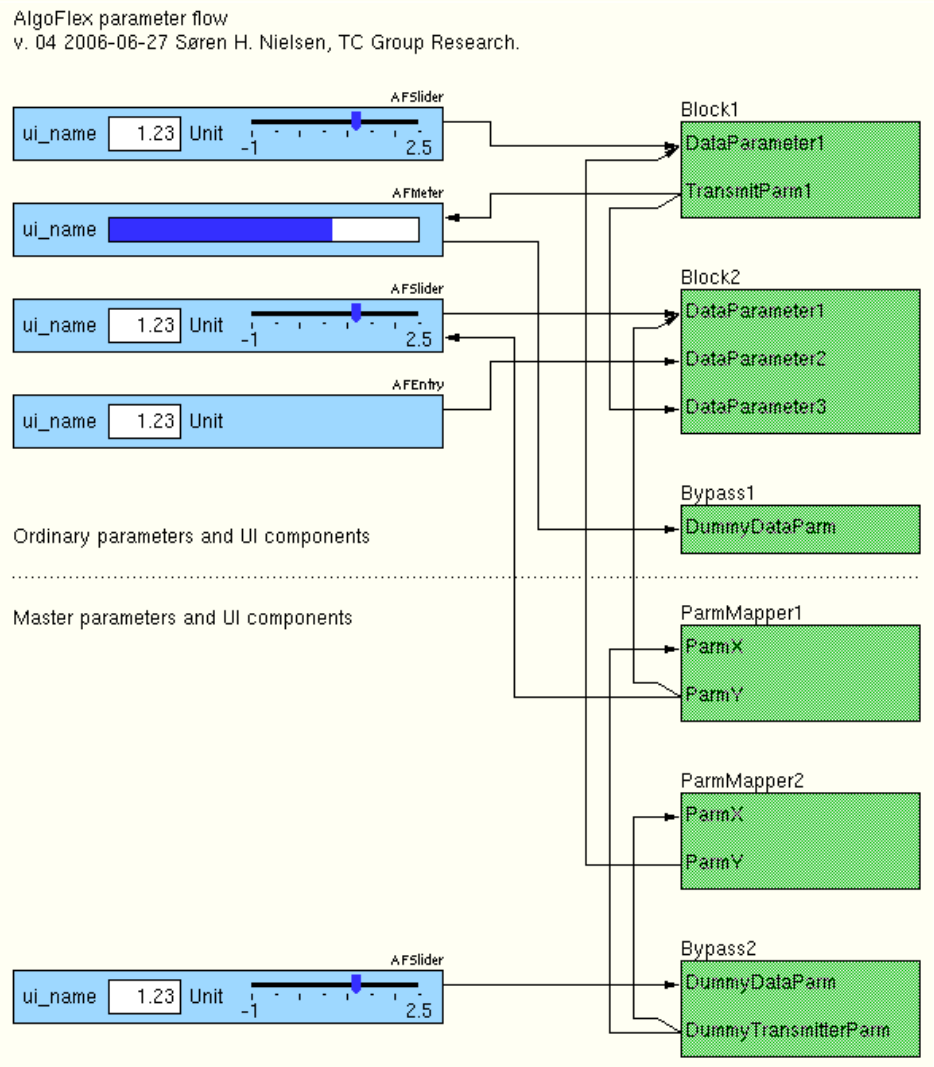
4.3.4 Associating UI components with parameters

Each normal parameter can be controlled by at most one UI component – plus optionally through the master component mechanism. The type of UI component used can be changed with a right-click pop-up menu for each parameter. The UI component may even be deleted through the pop-up menu menu.

The following components are available: `AFEntryField`, `AFSlider`, `AFScrollField`, and `AFButton`. For more details about the different components, see AfGuiTechnical#TypesOfComponents.

If a parameter is connected to a master component (see below) the state/value changes according to the setting of the master component.

An overall diagram of some possible parameter connections between UI components and blocks, and directly between blocks, is shown in the figure below:



The individual UI components and special blocks are explained in the AfGuiTechnical page.

5 AF GUI Components

The graphical components, that the AFGUI application uses, can be used from MATLAB scripts or from Java applications (i.e. without AFGUI). The AFComponents, as they are called, are built as JavaBeans, meaning they are independent software-components which can be used without any other dependencies than a standard Java runtime. The AFComponents typically present one or more graphical controls, and furthermore know how to communicate with an AlgoFlex server, for instance sending parameters to a given block.

Use the AFGUI application –

- when you need a GUI for your algorithm without writing a single line of code
- when you only need sliders to adjust parameter-values, meters, and other standard components
- when you don't need "application logic"

Use the AFComponents (without AFGUI) –

- when you need to combine the AFComponents with MATLAB graphics, such as plots
- when you need to program application logic (can be performed quickly with MATLAB Guide applications)
- when you need detailed control over the layout of the components (e.g. using Eclipse GUI builder tools)

Currently, the following AFComponents exist: AFSlider, AFMeter, AFButton, AFConsolePanel, AFEntry, AFComboBox, AFTextField.

The documentation for the AF GUI components is work-in-progress, but lives here:
http://wiki/bin/view/TCDesign/AFComponents
http://wiki/bin/view/TCDesign/AfGuiTechnical

5.1 The AFComponent Java interface

The individual AFComponents implement a set of Java interfaces. New components could be developed, and used immediately with the existing applications (incl. AFGUI), as long as they implemented the same interfaces.

EXAMPLES and API:

- AlgoFlex\Client\AFGUI\algoflex\swing\afcomponent
- package algoflex.swing.afcomponent

5.2 Using AFComponents in Matlab GUI programs

TODO: A screenshot from one of the examples, with the essential MATLAB patterns.

The following MATLAB functions can be used to create and layout the components: create_af_button.m, create_af_meter.m, create_af_slider.m, make_simple_gui.m, make_scalable_gui.m.

EXAMPLES:

- AlgoFlex\Client\DemoScripts\AFComponents

5.3 Using AFComponents from GUI-builder tools

"***JavaBeans*** *are reusable software components for Java that can be manipulated visually in a builder tool. Practically, they are classes written in the Java programming language conforming to a particular convention. They are used to encapsulate many objects into a single object (the bean), so that they can be passed around as a single bean object instead of as multiple individual objects.*" [Wikipedia]

Because the AFComponents are also JavaBeans they can be used within Java development tools such as Eclipse or NetBeans. See, for example, the Jigloo SWT/Swing GUI Builder for Ecliupse; or the Swing GUI Builder for NetBeans.

LINKS:
- http://en.wikipedia.org/wiki/JavaBeans
- http://www.eclipse.org/
    - http://www.cloudgarden.com/jigloo/
- http://www.netbeans.org/
    - http://www.netbeans.org/features/java/swing.html

6 Using AlgoFlex XML

CONCEPTS:

- Diagram XML: block instances, execution-rates, audio- and parameter-connections
- Parameters XML: values of data-parameters and their properties

The AlgoFlex server can save an algorithm, with its block-connections, parameters, exe-sequence and all, as a pair of platform-neutral XML files. The algorithm can the later be reconstructed, possible on another AlgoFlex server on another machine. Transforming the algorithm into XML may be called serialization, XML data binding, or marshalling; reconstructing the algorithm from XML may be called XML parsing or unmarshalling.

These server commands can be used to save/load XML documents: XMLLoadDiagram, XMLSaveDiagram, XMLLoadParameters, XMLSaveParameters, XMLLoadDiagramFile, XMLLoadParametersFile (see the complete description of these commands in the appendix).

The Schema of the AlgoFlex XML format can be found in the Appendix. The Schema is the grammar that unambiguously specifies what constitute a valid diagram XML or parameter XML document.

TIPS:

- Visual Studio contains a decent XML editor, with syntax check and color highlighting etc.
- The parameters XML doesn't need to contain all parameters of all the blocks of the algorithm; *partial presets* can also be saved.
- The sampling-rate is *not* stored in the XML; this is by design, as the XML-files should in general work with more than one specific sampling-rate.
- An XML document (i.e. a multi-line string) can easily be saved/loaded from MATLAB using the functions SaveTextFile and LoadTextFile.

6.1 GainGlider example

The following shows the XML representation of the algorithm from GainGlider example (in the Tutorial section). You'll recognize the 3 blocks of this algorithm: MyFilePlayer, MyGain, AudioIO. The parameters for AudioIO correspond to the particular sound-card used; if the algorithm is to be 'shared' between machines with different sound-cards, the AudioIO parameters could be saved in a separate parameter XML file, which could then exist in multiple versions.

6.1.1 The diagram XML for GainGlider

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<AlgoFlexXmlDoc author="ESK@tcgroup.tc" created="2008-09-12T15:09:53"
serverVersion="0.12.3" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="AlgoFlexXML.xsd">

  <diagramDoc>

    <partialExecutionSequence>
      <bl>MyFilePlayer</bl>
      <bl>MyGain</bl>
      <bl>AudioIO</bl>
    </partialExecutionSequence>

    <blockInstance>
      <blockName>AudioIO</blockName>
```

```
      <libName>AsioStream</libName>
      <IODimension nIn="2" nOut="2"/>
      <executionRate>1</executionRate>
    </blockInstance>

    <blockInstance>
      <blockName>MyFilePlayer</blockName>
      <libName>FilePlayer</libName>
      <IODimension nIn="0" nOut="2"/>
      <executionRate>1</executionRate>
      <connection dstBlock="MyGain" srcBlock="MyFilePlayer">
        <audioConnections>
          <audioConn dstChannel="1" srcChannel="1"/>
          <audioConn dstChannel="2" srcChannel="2"/>
        </audioConnections>
      </connection>
    </blockInstance>

    <blockInstance>
      <blockName>MyGain</blockName>
      <libName>Gain</libName>
      <IODimension nIn="2" nOut="2"/>
      <executionRate>1</executionRate>
      <connection dstBlock="AudioIO" srcBlock="MyGain">
        <audioConnections>
          <audioConn dstChannel="1" srcChannel="1"/>
          <audioConn dstChannel="2" srcChannel="2"/>
        </audioConnections>
      </connection>
    </blockInstance>

  </diagramDoc>

</AlgoFlexXmlDoc>
```

6.1.2 The parameter XML for GainGlider

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<AlgoFlexXmlDoc author="ESK@tcgroup.tc" created="2008-09-12T15:09:13"
serverVersion="0.12.3" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="AlgoFlexXML.xsd">

  <parametersDoc>

    <blockParameters>
      <blockName>AudioIO</blockName>
      <parameter>
        <parmName>AsioInputIndex</parmName>
        <data>
          <dimension>
            <dim>2</dim>
          </dimension>
          <real>
            <va>1</va>
            <va>2</va>
          </real>
        </data>
      </parameter>
      <parameter>
        <parmName>AsioOutputIndex</parmName>
        <data>
          <dimension>
            <dim>2</dim>
          </dimension>
          <real>
            <va>1</va>
            <va>2</va>
          </real>
        </data>
```

```
  </parameter>
  <parameter>
    <parmName>BufferSize</parmName>
    <real>1024</real>
    <property name="DataType" type="real" value="2"/>
    <property name="UpdateOrder" type="real" value="1"/>
  </parameter>
  <parameter>
    <parmName>DriverName</parmName>
    <string>ASIO Hammerfall DSP</string>
    <property name="DataType" type="real" value="1"/>
    <property name="UpdateOrder" type="real" value="1"/>
  </parameter>
  <parameter>
    <parmName>InputChannelInfo</parmName>
    <data>
      <dimension>
        <dim>2</dim>
      </dimension>
      <string>
        <va>Inp_01 &lt;--&gt; Chan_00: HDSP9652 ADAT 1</va>
        <va>Inp_02 &lt;--&gt; Chan_01: HDSP9652 ADAT 2</va>
      </string>
    </data>
    <property name="UpdateOrder" type="real" value="-1"/>
  </parameter>
  <parameter>
    <parmName>OutputChannelInfo</parmName>
    <data>
      <dimension>
        <dim>2</dim>
      </dimension>
      <string>
        <va>Out_01 &lt;--&gt; Chan_00: HDSP9652 ADAT 1</va>
        <va>Out_02 &lt;--&gt; Chan_01: HDSP9652 ADAT 2</va>
      </string>
    </data>
    <property name="UpdateOrder" type="real" value="-1"/>
  </parameter>
</blockParameters>

<blockParameters>
  <blockName>MyFilePlayer</blockName>
  <parameter>
    <parmName>BufferSize</parmName>
    <real>512</real>
    <property name="DataType" type="real" value="2"/>
  </parameter>
  <parameter>
    <parmName>FileSize</parmName>
    <real>8424864</real>
    <property name="UpdateOrder" type="real" value="-1"/>
  </parameter>
  <parameter>
    <parmName>InputFileName</parmName>
    <string>C:\Audio\SexDrugsRainbows-03.wav</string>
    <property name="DataType" type="real" value="1"/>
    <property name="UpdateOrder" type="real" value="1"/>
  </parameter>
  <parameter>
    <parmName>PlayMode</parmName>
    <string>whole*</string>
    <property name="DataType" type="real" value="1"/>
  </parameter>
</blockParameters>

<blockParameters>
  <blockName>MyGain</blockName>
  <parameter>
```

```
        <parmName>GLD_GainFactor</parmName>
        <data>
          <dimension>
            <dim>3</dim>
          </dimension>
          <string>
            <va>Linear</va>
            <va>3</va>
            <va>1e-6</va>
          </string>
        </data>
        <property name="IsGlider" type="real" value="1"/>
        <property name="UpdateOrder" type="real" value="10001"/>
      </parameter>
      <parameter>
        <parmName>GainDb</parmName>
        <real>-9</real>
        <property name="DataType" type="real" value="2"/>
        <property name="Default" type="real" value="0"/>
        <property name="Max" type="real" value="120"/>
        <property name="Min" type="real" value="-120"/>
        <property name="Scale" type="string" value="lin"/>
        <property name="Unit" type="string" value="dB"/>
      </parameter>
    </blockParameters>
  </parametersDoc>

</AlgoFlexXmlDoc>
```

6.2 XML format compatibility

The support for saving and loading algorithms in XML was originally implemented in AlgoFlex by a French computer science student, as part of his internship at TC. This initial implementation was completed within the scope of the project, and it allowed us to gain valuable experience and insight into what we wanted and didn't want in the AlgoFlex XML format. However, both the original format and the way it was implemented presented a number of severely limiting factors. So we eventually decided to fix these fundamental problems by designing and implementing a new AlgoFlex XML format. This 'new format' was completed in January 2008, and the AlgoFlex XML format hasn't changed at all since then.

Prior to the implementation of the new format, certain projects had created AlgoFlex XML in the old format. While some projects contain scripts that can easily re-create the XML in the new format, other XML documents reflected "manual tuning". Therefore a procedure exists to *translate* an AlgoFlex XML document from the old format into the new format – a one-time process.

For over 6 months, the AlgoFlex server modules for both the old and the new XML format were maintained. Thus, a server from this period can be used to load the old-format XML and then save the equivalent new-format XML. The table outlines this transitional period, with corresponding SVN revision IDs.

Date	SVN Revision	AlgoFlex version	What happened
18-09-2008	26939	1.0.0	The version 1.0 release, with Manual and all.
08-09-2008	26890	0.12.0	Removing support for the old XML format.

04-03-2008	26425	0.10.8	New XML format now working well – meaning only minor issues have been found and fixed after this version.
29-01-2008	26240	0.10.1	Both the new and old XML-parser and -serializer co-exist. Now using the new XML format by default.
23-01-2008	26226	n/a	The AlgoFlex XML Schema, for the 'new' format, completed.

Procedure for translating a diagram/parameter-XML file, old format into the new format:

1) Pull AlgoFlex – code for both server and blocks – from SVN, using a revision where both the new and the old format are supported (see table above), and where your own (i.e. non-standard) blocks are in a state corresponding to the time your XML documents were originally saved.

2) Build the AlgoFlex libraries, server and the required blocks (binaries). It's probably best to use debug-mode.

3) Start the AlgoFlex server, and connect from MATLAB as usual.

4) Load the old-format XML diagram and parameters using the "post-fixed" version of the XML-commands, such as XmlLoadDiagram_ (note the underscore).

4) Save the new format XML diagram and parameters, using the normal version of the XML-commands. These XML documents can now be loaded from a current AlgoFlex.

7 Special Topics

This section contains concise descriptions with examples of the more 'advanced' subjects.

7.1 AlgoFlexEval

An AlgoFlex block, or a chain of AlgoFlex blocks, can be evaluated as a function via MATLAB using the AlgoFlexEval function. The input to the block(s) is given as function arguments, and the function returns the output of selected channels of one or more of the involved blocks. This corresponds to automatically inserting MatrixPlayer and MatrixRecorder blocks and performing a batch-processing on the server. Using AlgoFlexEval can be very convenient when developing or testing a new block.

```
>> help AlgoFlexEval

  outputData =
AlgoFlexEval(srvId,algoExeSeq,parmsSettings,inputData,inputRouting,outputRouting,fs
                           [,nOutputSamples[,nRunSamples[,outputExeRateFactor]]])

  AlgoFlexEval can be used to 'evaluate' one or more connected AlgoFlex blocks
  as a function, i.e., specify a data matrix as input, then let the block(s)
  process the data, and return the resulting output.

  The AlgoFlex (server) commands to create and connect a MatrixPlayer and
  MatrixRecorder, set the exe-sequence, start/stop processing, etc. are issued
  automatically by this function. The block(s) to be evaluated must be created
  and inter-connected, prior to calling this function.
  The MatrixPlayer and MatrixRecorder are both in 'Repeat' modes, so that only
  the last (say) 100 samples can be obtained.

  INPUT ARGUMENTS:
    srvId          - the AlgoFlex server ID
    algoExeSeq     - the exe-sequence of the block instances constituting the
                     algorithm to evaluate
    parmsSettings  - a cell array of the form: {blockID1,parmName1,parmValue1;
                                                blockID2,parmName2,parmValue2; ...}
    inputData      - a matrix with dimension [nInputSamples,nInputChannels]
    inputRouting   - a matrix with dimension [nInputChannels,2] of the form:
                     [inputBlockID_ch1 inputBlockChan_ch1;
                      inputBlockID_ch2 inputBlockChan_ch2;
                      ...]
                     - OR ALTERNATIVELY -
                     {inputBlockName_ch1 inputBlockChan_ch1;
                      inputBlockName_ch2 inputBlockChan_ch2;
                      ...}
    outputRouting  - a matrix with dimension [nOutputChannels,2], analogous
                     to inputRouting
    fs             - the sample rate
    nOutputSamples - (optional, default: nInputSamples*outputExeRateFactor)
    nRunSamples    - (optional, default:
max(nInputSamples,nOutputSamples/outputExeRateFactor))
    outputExeRateFactor - (optional, default: 1)

  OUTPUT ARGUMENTS:
    outputData     - the result of the evaluation, i.e., the processed samples,
                     a matrix with dimension [nOutputSamples,nOutputChannels]

  EXAMPLE:
    srvId = AlgoFlexClient('OpenServCon','localhost',4242);

    idG = AlgoFlexClient(srvId,'Create','Gain',2,2);
    outputData = AlgoFlexEval(srvId, [idG], {idG,'GainDb',-3}, ones(100,2), [idG 1;
idG 2], [idG 1; idG 2], 44100);
```

```
    disp(outputData(1:10,:))  % 2 channels, gain -3 dB

    AlgoFlexClient(srvId,'Create','Delay',1,1,'MyDelay');
    AlgoFlexClient(srvId,'Create','Gain',1,1,'MyGain');
    AlgoFlexClient(srvId,'ConnectAudio','MyDelay',1,'MyGain',1);
    parms = {'MyDelay','DelayTime',0.001; 'MyGain','GainDb',6};
    x = [1;zeros(1000,1)];
    y = AlgoFlexEval(srvId, {'MyDelay','MyGain'}, parms, x, {'MyDelay',1},
{'MyGain',1}, 44100);
    disp(y(1:50)')  % delay 1 ms and gain 6 dB

    % If you have many channels routed to/from the same input/output-block, this
    % pattern can be used to set up the inputRouting or outputRouting argument:
    inputRouting = cell(nIn,2);
    inputRouting(:,1) = {'MyInputBlock'};
    inputRouting(:,2) = num2cell(1:nIn);

  See also: AlgoFlexClient, AlgoFlexEvals


>> help AlgoFlexEvals

  outputData =
AlgoFlexEvals(srvId,libName,parmsSettings,inputData,outputChans,fs,dispBlockInfo)

  AlgoFlexEvals can be used to 'evaluate' any AlgoFlex block as a
  function, i.e., specify a data matrix as input, then let the block process the
  data, and finally return its output.

  The AlgoFlex (server) commands to create the AlgoFlex block (-instance) to
  evaluate, create and connect a MatrixPlayer and MatrixRecorder, set
  exe-sequence etc. are issued automatically by this function.

  INPUT ARGUMENTS:
    srvId         - the AlgoFlex server ID
    libName       - the name of the block library (not block instance)
    parmsSettings - a cell array of the form {parmName1,parmValue1;
                                              parmName2,parmValue2; ...}
    inputData     - a matrix with dimension [nSamples,nInputChannels]
    outputChans   - the number of 'channels' in the output
    fs            - the sample rate
    dispBlockInfo - whether to display a BlockInfo of the evaluated block
                    (default: false)

  OUTPUT ARGUMENTS:
    outputData    - the result of the evaluation, i.e., the processed samples

  EXAMPLE:
    srvId = AlgoFlexClient('OpenServCon','localhost',4242);

    outputData = AlgoFlexEvals(srvId, 'Gain', {'GainDb',-3}, ones(10,2), 2, 44100);
    disp(outputData)

    outputData = AlgoFlexEvals(srvId, 'Delay', {'DelayTime',0.1}, [1;zeros(10000,1)],
1, 44100);
    find(outputData)

  See also: AlgoFlexClient, AlgoFlexEval
```

7.2 Client-side callbacks

The server can be instructed to send parameter-events, transmitted from a given block, back to the client via the TCP protocol. Any block with a transmitter-parameter can be used to generate client-side callbacks -- the block doesn't even know that this is happening.
This feature is useful, for instance for transmitting meter-data back to AFGUI or MATLAB to display graphically.

At the client-side, there are two alternative ways of handling the incoming transmitted events:

a) A Java class, implementing a certain interface, can receive the transmitted events. This is the method employed by AFGUI in order to get input for its meter component.
b) An .m-function can receive the transmitted events, by using a special feature of MATLAB. Note that, because MATLAB has a single-threaded program event loop, you should not use a blocking Start command, to start the server, and the .m-function used as callback should always return, relatively quickly.

If the rate of client-side callbacks is high (say >100 Hz), the MATLAB event loop may not be able to handle the incoming events in time, and the client system can become unstable. The server and TCP-communication, however, can handle event rates at 1000 Hz or higher.

EXAMPLE:

- AlgoFlex/Client/DemoScripts/ClientCallback/ClientCallbackTest.m

7.3 Parameter-only blocks

Certain blocks have only parameter-inputs and outputs, and have cannot accept any audio-connections. Recall that the communication between parameters is completely event-driven. This means that the parameter-only blocks and their parameter-connections are completely "free" unless events are being transmitted. Data-parameters can be considered "parameter inputs", and transmitter-parameters can be considered "parameter outputs".

When processing parameters, you may find these blocks useful, alone or in combinations:

- ParmMapper
- ParmCombiner
- ParmProdSum
- Sequencer

(See the description of these standard blocks in Appendix.)

7.3.1 Blocks to interface 'audio'-connections and parameter-events

Two blocks can interface 'audio'-connections and parameter-events:

- Parms2Audio: parameter-input, audio-output
- Meter: audio-input, parameter-output (periodic)

(See the description of these standard blocks in Appendix.)

7.4 AFData semantics

The AFData class is used as a flexible data-container. Each value of a data-parameter is an AFData object, and each transmitted *event* from a transmitter-parameter is also an AFData object. The AFData class is modeled loosely from the traditional MATLAB data types.

Every AFData object has a **type** and a **dimension**. The 'type' is either AF_STRING, AF_REAL, AF_COMPLEX, or AF_DATA. The latter is a recursive data type, so it allows values of different types to be combined in the same object; the equivalent in MATLAB is *cell arrays*. The 'dimension' is itself a vector of 1 or more integer numbers, specifying the 'shape' of the object. If the dimension has only one number it means that the object is 1-dimensional, i.e. a scalar or a vector. If the dimension has 2 numbers it means the object is a 2D matrix; for example, the dimension [2,3] corresponds to a matrix with 2 rows and 3 columns. This way AFData objects can be used to contain many different kinds of data, for instance, a three-dimensional complex matrix or a string array. By now the sharp reader will have noticed the main difference between AFData objects and the basic MATLAB data types: AlgoFlex does *not* distinguish between a "row vector" and a "column vector", for one-dimensional data types (MATLAB does).

7.4.1 Recursive AFData object types

An AFData object of type AF_DATA is a recursive data type, i.e. the object contains other data objects. Recursive data objects can be sent over the TCP protocol, and can be stored/retrieved via XML parameter documents, just like any other AFData object. The recursive AFData objects have **value semantics** (like the other AFData types), so for instance when the object is copied or assigned, a *deep copy* is being made.

When recursive AFData objects are used for a data-parameter in a block, it is normally best to choose the simplest possible data-type which can represent the useful values for that parameter. Avoid the temptation to invent complex, deep data structures, where different values for the parameter may have (very) different types or formats, as this can lead to unnecessarily complex code to parse/construct/maintain the data objects.

The following example demonstrates a recursive AFData object consisting of a string and a value:

```
AFData* stringAndValue = new AFData( 2, AF_DATA );  // (string, real)
stringAndValue->SetAt( AFData("MyString"), 0 );
stringAndValue->SetAt( AFData(123.456), 1 );

...

assert( stringAndValue.GetAt<AFData>(0).GetType() == AF_STRING &&
        stringAndValue.GetAt<AFData>(0).GetNumItem() == 1 );
string str = (string) stringAndValue.GetAt<AFData>(0);
double val = (double) stringAndValue.GetAt<AFData>(1);
```

7.4.2 Empty AFData objects, and non-initialised data-parameters

The following statements are equivalent, for AFData object x:

```
a) x.GetNumItem() == 0
b) x.GetDims()[0] == 0
c) x.IsEmpty() == true
```

Data-parameters which are not yet assigned their first value (i.e., AFData-pointers registered with a data-parameter, are NULL) are in a special state. A data-parameter will never enter this non-initialized state again, once it has been assigned a value. Note that this non-initialized state is *not* the same as an empty object. An empty object has a type and a dimension (i.e. 0), and should generally be considered a value like any other (think of the null-vector in math).

7.5 How to use non-scalar data-types in a block

Patterns … TODO

See the public methods of the AFData class, in Appendix.

7.6 Debug and Release mode binaries

Two versions of all the libraries and executable code exists, based on the two build configurations: *Debug mode* and *Release mode*. Typically the Release mode executables are considerably faster. In debug mode, a number of assert() statements in the engine are active; they perform some checks at run-time that can detect various problems. Using a debug-mode block with an otherwise release-mode AlgoFlex server should work in principle, but may cause unexpected behaviour or instability due to the mixed run-time libraries. In general, **use the debug-build during development, and the release-build for simulation**.

7.7 How to debug of a new block

If you have problems, try the following procedure:

1. Run the AlgoFlex server (Debug mode) in your debugger, or *attach* to an already running server process
2. Connect to the server with a client
3. Make the server load the blocks (DLL) to be debugged, either -
    a. Create an instance of the new block on the server, i.e. the client sends a Create <blockLibName> command
    b. Ask the server for the block's help text, i.e., the client sends a Help <blockLibName> command. This doesn't even run the block's constructor, so it's the 'minimal' way of loading a troublesome block.
4. Now you can set a breakpoint the source code for the block under investigation…

7.8 Performance profiling (using the server's Profiling run-time mode)

Some background info: [http://wiki/bin/view/TCDesign/AlgoFlexPerformanceTimestamps]

EXAMPLE:

- AlgoFlex/client/DemoScripts/ProfilerTestScript.m

7.9 The Native Processing Framework

The Native Processing Framework takes a complete algorithm – including its blocks, connections, parameter-values, exe-sequence – and automatically generates C++ code which implements the equivalent algorithm. In the auto-generated code, the algorithm is hard-coded, i.e. the methods from the relevant blocks are inlined into a staticly linked module (rather than individual DLLs). Because the code-generation is automatic, the risk if introducing new errors is much smaller than if the algorithm had to be re-implemented manually.

The AlgoFlex modules to support the auto-generated code are located in AlgoFlex\NativeAlgorithmFramework. In particular AFAlgorithmInterface.h describes the interface between the auto-generated algorithm code, and the application layer. The "application" of could be GPF, in case the algorithm is employed in a VST plug-in. A template for new *native processing* projects, including examples: AlgoFlex\NativeAlgorithmFramework\ProjectTemplate

CONCEPTS:

- **SignalProcess inlining**: the SignalProcess methods of the different blocks in the algorithm are inlined, according to the exe-sequence, so that function-call overhead is eliminated
- **mode**: a collection of data-parameter values, such as a (partial) preset, automatically serialized and encoded as static data in a C++ module
- **MemoryStream block**: a dummy i/o block, connecting the algorithm to the application; inserted instead of (say) the AsioStream block, prior to code-generation
- **GenerateCode command**: the command used to generate the modes and the algorithm C++ code, given a complete AlgoFlex algorithm

Note that *modes* can be layered: For instance the bottom layer contains all the parameters (e.g. coefficients) that are independent of sample-rate and user preset. The second layer would then contain the parameters that depend (only) on the sample-rate – one mode would be generated per supported sample-rate, so that the application can switch between them. The top layer would contain the parameters that constitute a traditional "preset".

7.10 Chunk-processing

Normally, the AlgoFlex server processes one sample at the time, by each block, according to the exe-sequence and the local exe-rates. Chunk-processing refers to the situation where each block processes several samples at the time, in order to improve CPU-cache performance and register usage etc.

Although chunk-processing is primarily a feature of the Native Processing Framework, it is also supported by the stand-alone server.

EXAMPLES:

- AlgoFlex\Client\DemoScripts\FilterTestScript_chunkproc.m
- AlgoFlex\NativeAlgorithmFramework\ProjectTemplate

7.11 Generating a block-diagram

The AlgoFlex server can generate a graphical representation of the blocks and their inter-connections in an algorithm. Specifically, the server generates a .dot file containing a description of the block-diagram, which it then uses the external program GraphViz to interpret and render into a corresponding graphics file. Prior to using this AlgoFlex feature, you must install GraphViz, such the program is on the system PATH. On Windows, it may also be necessary to install GhostScript, to get the ps2pdf program.

LINKS:

- http://www.graphviz.org/
- http://pages.cs.wisc.edu/~ghost/

The server's GenerateGraph command has the following syntax (copied from the AlgoFlexCommands TWiki page, of course):

GenerateGraph(shortcuts, hiddenblocks, what, size, format, filename)

```
Generates a visual representation of the algorithm, showing the audio and
parameter connections between the blocks and colorizing the blocks by their
respective execution rates. The shortcuts parameter contains a list of named
connection points which can be used to replace long connection lines. If not
used, pass an empty list {}.
The hiddenblocks parameter contains a list of blocks that you don't want to
have in the graph. An example of such could be a block with a very large
number of parameters that would tend to clutter the graph with wires. Any
wires connected to blocks that are hidden will be replaced by shortcuts to
the outputs of the hidden block, but the hidden block itself will not be
shown in the graph..
'what' is a string that speficies the types of wires generated. If the string
contains the substring 'Audio', audio-connections will be visualized. If the
string contains the substring 'Parms', data parameter connections will be
shown.
The default size is 10 (inches) and the default format is Encapsulated
PostScript 2 ('eps'). If a file name is specified, output will be written
into that, otherwise a temporary file will be used.
```

GenerateGraph({}, {})

```
Generate graph showing all blocks, audio and data parameter wires.
```

GenerateGraph({}, {'CoeffComput'}, 'Parms')

```
Generate graph showing all blocks except 'CoeffComput' and show only
parameter connections.
```

GenerateGraph({'FEED_DELAY_L' 3 'LER1' 'DELAY1_L' 2 'RRT32' }, {}, 'Parms')

```
Generate graph showing all blocks, but replace the wires starting at block
FEED_DELAY_L's output 3 and DELAY1_L's output 2 by shortcuts with the labels
'LER1' and 'RRT32', and show only audio connections.
```

EXAMPLE:

- AlgoFlexClient(srvId,'GenerateGraph',{},{},'Audio+Parms',50,'png','C:\tmp\MyDiagram')

7.12 Simulink/RTW integration

A Simulink model can be semi-automatically transformed into C/C++ code that constitute a new AlgoFlex block. Certain (reasonable) limitations apply to the Simulink model, in order for the code-generation to work: only discrete-time processing, no dynamic memory-allocation, …

Furthermore, Simulink with relevant Blocksets, the Real-Time Workshop and Embedded Coder are required. Unfortunately these MathWorks products suffer from rather expensive licenses.

The principles and tools for performing this transformation are described in:
AlgoFlex\SimulinkCodeGen\Doc\QuickStart.pdf

An excerpt from the AlgoFlex/RTW follows:

Introduction

This document is a quick-start guide to AlgoFlex-block-generation with Simulink and the Real-Time Workshop. The goal is to get you introduced into how to work with the RTW-AlgoFlex SDK by means of demo-models and stepby-step examples. While Simulink provides the possibility to create block diagrams that represent your audio-effect, the Real-Time Workshop is an extension that translates the model to C (or C++) code. In addition, we have implemented C++-classes that interact with the generated code and interface it to the AlgoFlexenvironment. In the following, we'll jump into the design-process quickly and try to omit treating the details wherever it's possible. In chapter 1 the first-time setup is described and after that, in chapter 2, we'll jump into matters directly by investigating some example models that demonstrate all important features of the SDK. Chapter 3, finally, tries to show how to solve some common problems. Have fun... and, if additional questions arise, take a look into the appendix or MATLAB's online-help.

Features

• Automatic generation of C/C++ code for new AlgoFlex blocks
• Compiled AlgoFlex blocks do not require MATLAB/Simulink or any Simulink runtime library
• Sample-based and/or frame-based processing
• No extra buffering (zero-latency processing)
• AlgoFlex data-parameters supported (scalar, vector, matrix)
• Event-based internal parameter (e.g. filter coefficients) calculation, based on parameter updates
• Default, minimum, and maximum parameter properties can be defined
• AlgoFlex transmitter-parameters supported (scalar, vector, matrix)
• Multi-rate processing supported – automatic scheduling
• Multi-channel processing
• Referenced models supported (modular/hierarchical Simulink models)
• Demo models – showing the most important features – included

Who did it?

The AlgoFlex/RTW SDK was developed in 2006-07 by Matthias Brandt in collaboration with Esben Skovenborg, for TC Group|Research.

7.13 Distributed processing

It is possible to run an algorithm *distributed*, that is, on several CPUs. This may be desirable, if the algorithm is particularly CPU-hungry. Most – if not all – existing AlgoFlex algorithms have been able to fit onto a single CPU, so the distributed processing feature of AlgoFlex has not been used a lot.

Distributed processing on AlgoFlex is described in: AlgoFlex/doc/BertrandBOUZIGUES-Internshipreport.pdf .

7.13.1 TCP-blocks

Before running a distributed algorithm, you must determine which blocks to run on the same CPU – in other words, where to divide up the algorithm. A pair of TcpSource/TcpDestination-blocks are inserted at each 'cut'. An AlgoFlex server is then started on each of the CPUs involved, and the different sections of the algorithm are loaded on the different servers.

Note that, for real-time simulation, this requires buffers (in the TCP blocks and I/O blocks, if any) that are relatively large.

BLOCKS:

- TcpSource – transmit data to another AlgoFlex server
- TcpDestination – receive data from another AlgoFlex server

7.13.2 AlgoFlex Master-server mode

[Copied with minor editing from: http://wiki/bin/view/TCDesign/AlgoFlexMasterServer]

What is it ?

The MasterServer makes the user able to distribute blocks manually on different servers without worrying about TCP blocks configurations. The user is connected only to one server, the master server which is able to open connections to the other servers and control them.

How to use it ?

Blocks Name

In order to identify blocks in a MasterServer context, every block name uses the pattern :

```
BlockName@ServerName
```

The block name can be chosen by the user or by the server by default. The server name is set by default as the hostname of the machine.

Commands

There a 4 kinds of commands in a MasterServer context :

the MasterSever commands that have been created just for the MasterServer (InitMasterServer, OpenConnection..)

the commands that need to be executed on all servers (slave + master) (SetSampleRate..)

the commands that need to be executed on a single server (BlockInfo, CreateBlock..)

the commands that need to be executed on some servers (ConnectAudio..)

For the last 3 kinds of commands, the user doesn't need to specify on what server the command has to be executed; the master server has just to check the block name.

Differences in Matlab scripts

First, the user needs to init the master server mode using the command : **"InitMasterServer"**

Then, it can opens as many connections as he wants to the other servers using the command :
"ConnectToAFS", "IP", port

It is possible to get the list of the slave servers using the command : **"GetConnections"**

For creating blocks, the user can specify or not a server or a block name :

"Create", "MastrixPlayer", 0, 1 will create a block MatrixPlayer on the MasterServer itself, the name will be chose by the server (ie: MatrixPlayer-1@MasterServerName)

"Create", "MastrixPlayer", 0, 1, "MyBlock" will create a block MatrixPlayer on the MasterServer itself, the name will be MyBlock@MasterServerName

"Create", "MastrixPlayer", 0, 1, "MyBlock@myServer" will create a block MatrixPlayer on the server called "myServer", the name will be "MyBlock@myServer".

"Create", "MastrixPlayer", 0, 1, "@myServer" will create a block MatrixPlayer on the server called "myServer", the name will be chosen by the server (ie: MatrixPlayer-1@myServer)

The other commands are identical.

7.14 Command-line arguments for the server

The AlgoFlex server has the following command-line arguments:

```
C:\AlgoFlex\bin\win32_x86\Debug>AlgoFlexServer.exe -?

AlgoFlex Server version 0.12.4 (Debug) (without master/slave extensions)
USAGE:
AlgoFlexServer [-s<server-port>] [-e<client/server-echo>] [-p<process-priority>]
               [-c<command-file>]
where options are:
   <server-port> the default port is 4242
   <client/server-echo> 0 or 1, whether to print client/server communication
   <process-priority> is 0 = normal, 1 = high, 2 = real-time
   <command-file> a text file, interpreted as a list of commands
```

7.14.1 Command-files

The AlgoFlex server can run one or more commands, as the first thing, after the process starts (i.e., also before listening for a client to connect). This is useful in connection with “batch-processing” tasks. All server-commands can be used. Note, in particular, the commands related to XML…

The syntax of the command-files is exemplified below:

```
% File: AlgoFlex/Server/CmdFileParser/Test-CmdFile.txt
% An example to test the AlgoFlex server's command-file parser

% No args
Echo()

% Real and string argument types
Echo(42)
Echo(42,43)
Echo('Hello, World!')

% Test various characters
Echo('/\?!"#¤%&-_|()[]{}=æøå')

% Real- and string-array (1D vector, really)
Echo([42,43])
Echo({'Hello','World'})

% Empty arrays of real both types
Echo([])
Echo({})

%This(is-bad syntax(

% OK with extra white-space
Echo ( [ 123.456 , -1.23e45 ] )

% An unknown command can be handled
XYZ()

% Finally, terminate the server-process
Quit()
```

7.15 The order of things

Often a new algorithm is initially developed in the form of a MATLAB script which will create and connect blocks, set their parameter values, etc. Issuing these common AlgoFlex commands in a certain order leads to the most robust performance.

The following pattern is recommended for (MATLAB) scripts creating an algorithm on an AlgoFlex server:

1. Connect to the server, and store the server-id
2. Create block instances
   Set exe-rate-factors (of blocks which are upsampled or downsampled)
   Set special data-parameter values, for blocks with parameters which configure the blocks available parameters (e.g. the CombinerRatio parameter of ParmCombiner blocks)
3. (Load diagram-XML for sub-algorithm, if any)
4. Make audio-connections
   Make parameter-connections
5. (Load parameter-XML for sub-algorithm, if any)
6. Set data-parameter values
7. Set execution-sequence
   Set sample-rate
8. (Check state of blocks using BlockInfo, ReadProbe and GetData commands)
9. Simulate algorithm using Start/Stop commands or AlgoFlexEval functions

Note that, when loading an algorithm from a pair of diagram- and parameter-XML files, the server automatically takes care of setting up things in the appropriate order. In particular, the **UpdateOrder**, which is an optional data-parameter property, together with parameter-dependency information, makes this possible.

8 Frequently Asked Questions

8.1 Installation and Build

1) Q: Where do I find the latest AlgoFlex Installer?
   A: On TC's SVN server: svn://svn/AlgoFlex/trunk/Installer/bin/AlgoFlexInstall.exe

2) Q: Why doesn't the AlgoFlex Installer have a un-install feature?
   A: Uninstalling is equivalent to deleting the files under the top-level AlgoFlex directory, and removing the desktop shortcuts. The installer does not store any files anywhere else, nor any data in the Registry (Windows).

3) Q: When I update my AlgoFlex, using a newer AlgoFlex Installer, can I "install over" my existing AlgoFlex?
   A: Yes, generally this will work fine. The only issue is if a file has been moved/renamed, in which case you would get 2 copies of the file, and could thus risk using the old one rather than the new one.

4) Q: I updated my AlgoFlex using SVN Update (instead of the AlgoFlex Installer). Now my blocks don't work anymore – what to do?
   A: You probably need to rebuild the link-libraries (AlgoBaseLib), the server, and then the blocks you use – in that order. The link-libraries and server are easily built using the AlgoFlex\Server\AlgoFlexServer.sln solution (Windows).
   If you used the AlgoFlex Installer instead, you would only need to rebuild your own blocks.

5) Q: I copied or installed the debug-mode AlgoFlex server on to some PC, but it gives me an error ("Your application is not installed correctly" or similar) when starting; the release-mode server works fine, however.
   A: The debug-mode server requires that Visual Studio (Express/Pro) is installed on the machine. Microsoft license says.

6) Q: Can I use the newer Visual Studio 2008 instead of Visual Studio 2005?
   A: It seems that neither the (XML) project files nor the (binary) C++ link-libraries are compatible between these two versions. So in order to avoid double-maintenance, only one version is supported – and at the moment that is Visual Studio 2005 SP1.

7) Q: What's the deal with the binaries under AlgoFlex/bin/win32_x86/… in SVN?
   A: In the past we've been using this SVN folder as a quick'n'dirty way of distributing the AlgoFlex binaries (for Windows). However, you should most likely use the AlgoFlex Installer instead of relying on these binaries. The files under AlgoFlex/bin might disappear from SVN at some point.

8) Q: Why are the binaries under AlgoFlex/bin/win32_x86/… in SVN write-protected?
   A: The binaries are write-protected because there is an SVN 'lock' on them; this is to indicate that they are the 'official' binaries, and that you should not commit your own binaries to SVN. You can safely delete your local copy of the binaries, or reset their read-only bit, in order to build your own, of course.

8.2 Blocks and Development

9) Q: Why does a block have both a *help-text* and a *BlockInfo*?
   A: The BlockInfo shows the current parameter values of a specific block instance – the block's "state" so to speak. As opposed to the block-library's static help-text.

10) Q: Why are there both an IirFilter block and an IirFilt class?
A: The IirFilter block implements a multi-channel IIR filter, of a given order, for which coefficients are supplied as block parameters by the client. Typically the IirFilter block is used for filtering that does not depend on user-parameters. The IirFilt class provides an optimized and tested IIR filter engine and design-methods that can be used by any AlgoFlex block (with all the normal advantages of code reuse :).

11) Q: How do I debug/test my block with transmitter-parameters?
A: Use a Sequencer block for recording the data-objects.
A: Use Client-side callbacks, such that an .m function is called, whenever the transmitter-parameter sends an event – see AlgoFlex\Client\TestScripts\ClientCallbackTest.m.
A: Connect the transmitter-parameter to a Parms2Audio block, which then is connected to a MatrixRecorder, or simply take a look at the BlockInfo of the Parms2Audio block.

12) Q: What do the throw () and throw (const char*) in some of the AlgoBase methods mean?
A: The throw () in a method declaration means that the method in question may not throw any exception of any type. The throw (const char*) in a method declaration means that the method may throw a (C-style) string, should it want to do so.

13) Q: My algorithm has a block which doesn't have a SignalProcess (i.e. parameter-handling only) – should I remove this block from the exection-sequence?
A: No – *all blocks, that are part of the algorithm must also be somewhere in the execution-sequence*. However, the actual position of such a block in the exe-sequence doesn't matter.

14) Q: Why do I get the error "The API version of server and block differs" (or similar)?
A: The AlgoFlex server that you are running has been built using one version of the block API, and one or more blocks have been built using *another* version of the API. Using this combination could potentially result in any number of strange errors; hence the check by the server. If you have updated your AlgoFlex using a new AlgoFlex Installer, you may have to recompile your own blocks (i.e. the blocks that are not included in the Installer as binaries).

15) Q: Why are the blocks using 64-bit doubles everywhere? My platform X doesn't support that…
A: AlgoFlex is primarily a virtual prototyping system, meaning *flexible algorithms* and *quick implementation*. On modern Intel/AMD CPUs operations on 64-bit doubles are generally as fast as on 32-bit floats. Furthermore, a given target platform may only support integer-arithmetic, or quantization-issues may require re-implementation of the final signal-processing code anyway.

16) Q: Why do I get this error from the server, when I try to load an XML parameters- or diagram-document:
```
ERROR: :0:0 warning: An exception occurred! Type:RuntimeException,
Message:Warning: The primary document entity could not be opened.
Id=AlgoFlexXML.xsd
```
A: This is the server XML parser's way of telling you that it could not find the file AlgoFlexXML.xsd. The parser needs this file, containing the XML Schema, in order to validate your XML document. Usually this problem is cause by starting the server with a *working directory* other than the one in which the server executable is located (which is usually a bad idea).

17) Q: How do I start a server from my MATLAB script, in such a way that the shell window closes automatically when I tell the server to Quit?
A: The following pattern works well. You can also supply 'start' with various arguments, such as *run minimized*.
```
afserverexe = fullfile(ServerDir,'AlgoFlexServer.exe');
```

```
if exist(afserverexe,'file')
   status = system(['start ' afserverexe]);
else
   disp('Not starting new AlgoFlex Server (connecting to an already running).')
end
srvId = AlgoFlexClient('OpenServCon',ServerHostname,ServerPort);

...

AlgoFlexClient(srvId,'Quit');
```

8.3 Performance

18) Q: I get drop-outs during my real-time processing – what can I do?
A: Use the Release build of server and blocks (not the Debug build)
A: Kill non-essential processes and services on the PC, on which the AlgoFlex server is running.
A: Make sure that the individual blocks don't have an "audio-rate" execution-rate (i.e. ExeRateFactor =1) if their "local sample-rates" are in fact lower. For instance, certain side-chain blocks run at FS/4 (i.e. ExeRateFactor =1/4).
A: Increase the buffer-sizes of your audio-I/O device and corresponding block (e.g. AsioStream or AudioStream).
A: Increase the priority of the server – see the command-line arguments of the server program.
A: Use chunk-processing, a chunk-size of (say) 32 samples is for many algorithms more efficient than the (default) sample-by-sample processing. Can be difficult, though, if your algorithm contains feedbacks.
A: Run the GUI program, for parameter-control etc., on a separate PC (or CPU). The client/server communication is based on TCP anyway, so there's nothing special to do.
A: Is there a bottleneck in (some block in) your algorithm, that could be optimized? It is recommended using a profiler (e.g. the Intel VTune profiler) to make an empirical analysis, before performing any optimization that might hurt the portability, readability, or maintainability of the code.
A: Get a faster PC (a CPU with 2 or more *cores* is recommended) ☺

19) Q: Why are my debug-build server and blocks *slow*?
A: The debug-mode modules contain a number of range- and consistency-checks, and assert() statements that are disabled in the release-build. Furthermore, the release-build is of course optimized by the compiler. The guideline is: *Use the debug-build during development, and the release-build for simulation*.

20) Q: Why does the CPU-load of my AlgoFlex server rise considerably, when the input to my algorithm changes from the real signal to digital silence (i.e. zeros)? The algorithm contains a recursive structure (such as an IIR filter) which converges towards zero, for zero input.
A: When the recursive structure converges towards zero, it will often pass through a so-called *sub-normal* numerical range – numbers which are so close to 0 that they can not be represented exactly in double precision. In certain older CPUs, operations with such numbers are considerably more expensive than for normal numbers. An easy fix is to insert a NoiseInjector block, with an appropriately low noise-level (say –200 dB), just before to the recursive structure in the algorithm.
A: If your algorithm contains a division which will lead to a division-by-zero for 0's input, the FPU might throw an exception. It is best to avoid division-by-zero and other numerical errors (e.g. sqrt, log) in the SignalProcess-methods.

8.4 Miscellaneous

21) Q: Do I need MATLAB in order to run AlgoFlex?
A: No – you could connect to the AlgoFlex server using another client program – such as the AFGUI. However, it is convenient to use the MATLAB client while developing a new algorithm.

22) Q: Is there a graphical diagram-editor tool (like in Simulink or whatever)?
A: No – not yet :) We have a student-project definition for developing such a tool, but we haven't yet found a student with the right profile.

23) Q: Is AlgoFlex supported for Mac OS X?
A: The Windows implementation of AlgoFlex serves as the reference implementation. Except for certain system-dependent blocks, we try hard to develop everything such that it is Linux (POSIX) compatible. As OS X itself supports POSIX, Mac is implicitly supported.

24) Q: How do I build a VST plugin from my AlgoFlex algorithm?
A: You use the Native Processing Framework (in AlgoFlex) to compile your algorithm, with its blocks and parameter-sets, into a link library. You then use that in a GPF (*General Plugin Framework*) project. GPF was originally developed at TC Works but has been adopted by TC Electronic. GPF is independent from AlgoFlex, and is beyond the scope of this document.

25) Q: Different users are allowed to use AlgoFlex servers running on my machine (by connecting to it with their clients); what consequences does that have on the security of my machine?
A: In principle an AlgoFlex server is able to do anything that a process is allowed to do, for the particular user account under which the process is running. In other words, an AlgoFlex server process, as such, should *not* be considered "safe" – consider, for example, what a server could do with a FileRecorder block, if controlled by a malicious AlgoFlex client.

26) Q: Where is the MD3 compressor, the VSS reverb, and all the other TC Electronic algorithms?
A: The AlgoFlex system for algorithm development and virtual prototyping is one thing; the algorithms that are developed on AlgoFlex is another. This manual only describes the former. (The exchange of IP, such as algorithms, may be a matter of one manager talking to another…)

27) Q: Why don't I have access to the folders TCBlocks and Applications on the AlgoFlex SVN repository?
A: It has been decided that not every AlgoFlex user should automatically have access to all algorithms/blocks developed by TC-E and TC-G Research. Therefore, root-level access to these two folders has been blocked, and access is granted to specific sub-folders, on demand (i.e. *just ask*). Note that you *can* have access to a certain folder, in SVN, without having access to its parent folder – simply specify the full path when you do SVN Checkout.

28) Q: Is it true that AlgoFlex is also the name of some kind of drug in Hungary?
A: Yes. We haven't been able to get hold of any yet, so we don't know what it does :)

29) Q: How about running AlgoFlex on a real-time Linux OS, such as RTLinux or eCos, on some PC-style hardware?
A: Such a combination would provide a very powerful and flexible platform. A pilot study indicated that the dominating task, in such a project, would probably be to fit the software to the somewhat artificial "hard real-time plus the rest" model; for instance, standard drivers

(e.g. for soundcards) can generally not be used in such a real-time OS.
As an easier alternative, is might be sufficient to use a ‘normal’ (non-RTOS) Linux, but built with a kernel customized for real-time operation (e.g. by obtaining pre-emptive scheduling also in kernel mode). MontaVista Linux is a prominent example of this approach. Anyhow, experiments have shown that even on a desktop PC running Windows XP, it is possible to obtain a stable real-time processing with a <5 ms roundtrip latency, simply by following a few simple guidelines (see the Performance FAQ).

9 Bugs, Feature-requests, Mailing-list and TWiki

By collecting and comparing feedback from AlgoFlex users, with their experiences and wishes, we can focus the future development. Only a couple of people are regularly involved in developing AlgoFlex, and usually it is not their first priority. Therefore, new features are generally added to AlgoFlex when they would contribute significantly to some research/development-project (or would significantly hurt the project, if not added). However, in order for us to continually prioritize our effort, it is important that bug reports and feature request are entered into TestTrack.

9.1 AlgoFlex-Users mailing list

There is a mailing list, with the address: AlgoFlex-Users@tcelectronic.com. On AlgoFlex-Users all AlgoFlex-related questions and comments are welcome – especially those that may be of general interest. If you'd like to subscribe to the list, please contact: IT@TCGroup.tc (or EsbenS@TCGroup.tc). Unfortunately, there is currently no way of seeing past messages – no mailing-list archive.

9.2 Found a bug?

If you've found a bug, you basically have 3 options:

1) **Fix it**

… test it, and commit the change to SVN. Good karma for you and a better AlgoFlex for all of us. Feel free to consult with one of the AlgoFlex architects first, to get a second opinion on your patch.

2) **Report it**

Submit a minimalistic MATLAB script and/or a set of XML-diagram and -parameter files to **reproduce** the problem. Good karma for you – and a good chance that the bug will be fixed at some point. TestTrack makes it easy to organize the reports.

3) **Forget it**

Your contribution would be appreciated. (Risk of bad karma.)

9.3 TestTrack

The TestTrack system is used for collecting bug reports and feature requests for the AlgoFlex system. The different components of the AlgoFlex system have been registered in TestTrack: Server, MATLAB client, Standard blocks, TC blocks, Documentation, etc.

You can access AlgoFlex on TestTrack via the friendly web interface: http://testtrack/ttweb

Log in using your TC domain name/password, and choose the "AlgoFlex" project. To get access to the AlgoFlex project on TestTrack, you must first to contact a TestTrack admin: strni@TCElectronic.com.

By entering the bugs and feature requests that you encounter, we will be able to focus the continual development of AlgoFlex. Also, you can check on TestTrack whether some unexpected behavior is a known problem, in which case there may be a **work-around**.

9.4 AlgoFlex on TC TWiki

The TC TWiki server contains a number of pages concerning AlgoFlex-related topics; in fact some of the sections in this Manual were derived from TWiki pages. You need to have access to

the TC Group domain to use the TWiki server. If you haven't used the TWiki before, you must first contact a TWiki admin: SHN@TCElectronic.com.

Be warned, however, that many of the pages on TWiki contain information which is incomplete, out-of-date, misleading, or just wishful thinking ;-)

If you find information in this Manual that contradicts information on TWiki, generally the Manual is right.

These two pages are good places to start on TC Twiki:

- AlgoFlex portal: http://wiki/bin/view/TCDesign/AlgoFlex
- Reading Guide - finding your way in the AlgoFlex documentation: http://wiki/bin/view/TCDesign/AlgoFlexReadingGuide

10 AlgoFlex Installation and System Requirements

The following is included directly from the AlgoFlex Wiki :
http://wiki/bin/view/TCDesign/AlgoFlexInstall?skin=plain

This document describes how to install and setup AlgoFlex.

UNLESS YOU NEED TO MODIFY THE ALGOFLEX ENGINE OR SERVER, YOU SHOULD PROBABLY JUST RUN THE *ALGOFLEX INSTALLER*:
svn://svn/AlgoFlex/trunk/Installer/bin/AlgoFlexInstall.exe

THE INSTALLER CAN ALSO INSTALL THE *ALGOFLEX SDK* WITH LINK-LIBRARIES, DOXYGEN DOCS, MATLAB CLIENT FUNCTIONS, AND EVERYTHING :-). OF COURSE THE SYSTEM REQUIREMENTS (BELOW) STILL APPLY.

10.1 SVN and VSS

All AlgoFlex related source code is now located on TC-E's SVN server, in the repository named **AlgoFlex**. All references to VSS, in this page and others, should be read as SVN, that is -

svn://svn/AlgoFlex/trunk/...

10.2 Installing the AlgoFlex binaries

If you only need to use algorithms consisting of the pre-build blocks, the easiest way to start using AlgoFlex is to install the AlgoFlex binaries (for Windows). This installation includes the server program and the standard processing blocks.

1. From SVN: Checkout, from svn://svn/AlgoFlex/trunk/bin/win32_x86
2. Run AlgoFlexServer.exe from AlgoFlex/bin/win32_x86/Debug or AlgoFlex/bin/win32_x86/Release (depending on whether you need diagnostic information or speed, respectively).

10.2.1 System requirements for AlgoFlex binaries

- Windows 2000(SP4) or XP(SP1)
- The server executable depends on certain dll files from the Visual C++ runtime. *If you receive an error such as "DLL not found" or "Application not installed correctly", when launching AlgoFlexServer.exe, do the following:*
  - Run the installer vcredist_x86.exe, located in AlgoFlex\bin\win32_x86\Redist. You can now run the Release-mode binaries, on a 'new' system (i.e., without Visual Studio installed).

- Unzip the file Microsoft.VC80.DebugCRT.zip, located in AlgoFlex\bin\win32_x86\Redist, and move the resulting directory to your folder containing the server executable. You can now run the Debug-mode binaries, on a system without the corresponding runtime libraries.

Links:

- Microsoft Visual C++ 2005 SP1 Redistributable Package

10.3 Setting up the MATLAB client

Set up MATLAB to work with the AlgoFlex client function.

1. From SVN: Get latest, recursive, from /AlgoFlex/Client
2. To add the AlgoFlex client to the MATLAB path:
   ```
   >> addpath('C:\projects\Tools\AlgoFlex\Client')
   ```
3. To add the Java library, needed by the client to MATLAB's Java path:
   ```
   >> javaaddpath('C:\projects\Tools\AlgoFlex\Client\afjavalib.jar')
   ```
4. These examples assume that your AlgoFlex root directory is C:\projects\Tools\AlgoFlex. **We recommend that the above two lines of code are executed by Matlab at startup** -- for instance, include them in the startup.m file in your Matlab/toolbox/local directory. (Hint: If you do not yet have the startup.m file, then have a look at the startupsav.m file).
5. Now you can use the client program from MATLAB, for example: `help AlgoFlexClient`

10.3.1 System requirements for Matlab client

- Matlab Rel.14/SP2 (= version 7.0.4). This Matlab version, or newer, is required in order to have the degree of Java integration needed by the AlgoFlex client function.

When transferring large amounts of data between the client and the server, it may be necessary to increase the heap space for the Java VM. See the attached HeapspacefortheJavaVM.pdf for details.

10.4 Setting up the Java client (AFGUI)

1. From SVN: Get latest, recursive, from /AlgoFlex/Client
2. Install Sun's Java runtime 1.5.0 or newer. Download and follow the instructions from here: http://java.sun.com/j2se/1.5.0/download.jsp
3. Run the Java client application, from the directory AlgoFlex/Client/AFGUI/bin:
   ```
   java -jar afgui.jar
   ```
4. Usage, see AlgoFlexGUI (on TWiki)

10.4.1 System requirements for Java client

- Any platform which supports Java version 1.5, formally called "Java 2 Platform Standard Edition 5.0" or J2SE 5.0.

10.5 Installing and Building AlgoFlex from source

To install the Visual C++ 8 Express (which is free), with Service Pack 1:

1. Download and install the Visual C++ 2005 Express Edition: http://www.microsoft.com/express/2005/download/default.aspx

2. Download and install the Visual Studio 2005, Express Edition, Service Pack 1 (as it so far normally isn't included in the above): http://msdn2.microsoft.com/en-us/vstudio/bb265237.aspx/
3. Download and install Windows platform SDK: http://www.microsoft.com/express/2005/platformsdk/default.aspx/
4. Make sure you have set up the Platform SDK lib/include directory paths in your Visual C++. In the menu: Tools -> Options -> Projects & Solutions -> VC++ Directories. http://www.microsoft.com/express/2005/platformsdk/default.aspx/

The VC++ 6 compiler (from 1998!) is no longer supported by the AlgoFlex projects - please use the Visual C++ 8 (= Visual Studio 2005 (Express)) instead. According to Microsoft these two compilers can co-exist, if required.

---

*An **AlgoFlex SDK**, for use with the VC++ 8 is available via the AlgoFlex Installer. With the SDK, building a new AlgoFlex block will be very simple, and does not involve compiling the server or runtime-libraries. When not using the Installer, the following steps are required.*

Building the AlgoFlex system from the source code, using VC++ 8 (Visual C++ 2005):

1. From SVN: Get latest, recursive, the entire AlgoFlex project: /AlgoFlex
2. Check the file Common.vsprops (located in the AlgoFlex root directory), to make sure that the AlgoFlexRoot macro is set to the path of your local AlgoFlex root directory. This should be the only place, in the AlgoFlex projects, where an absolute path is specified.
3. Build the AlgoBaseLib and AFComLib libraries (used by each block, and by the server):
    - Open the solution: AlgoFlex\Server\AlgoFlexServer.sln
    - Build All: this will build the AlgoFlex libraries and the server application.
4. (Optionally) Build the standard blocks - each block has a corresponding project in the solution: AlgoFlex\AlgoFlexBlocks\AlgoFlexBlocks.sln
5. (Neither the MATLAB client nor the Java client programs require re-compilation.)

Under Unix, the above build steps can be performed by calling a top-level make script, located in the AlgoFlex root directory: run init, then make.

Note that two versions of all the libraries and executable code exists, based on the 2 build configurations: *Debug Mode* and *Release Mode*. Typically the Release mode executables are considerably faster. In debug mode, a number of assert statements in the engine are active; they perform some checks at run-time that can detect various problems. Using a debug-mode block with an otherwise release-mode AlgoFlex server should work in principle, but may lead to instability due to the mixed run-time libraries.

See also the notes in the directory AlgoFlex/doc/install, describing how the AlgoFlex source tree is structured.

10.5.1 Creating a new AlgoFlex block

1. First, install the VC++ 8 compiler, and build the AlgoFlex libraries (as described above).
2. From Matlab, call the function `mkblock.m` (located in AlgoFlex\AlgoFlexBlocks) to create a new project for your new block, based on one of the standard blocks as 'template' (e.g. use the simple Gain block as template). This will create both the source files and a corresponding project file, based on the selected 'template' block.
3. The new block can then be built, using the newly created project (the .vcproj file)
4. *For further info, see _Block-Writer's Guide* in the AlgoFlex Manual._

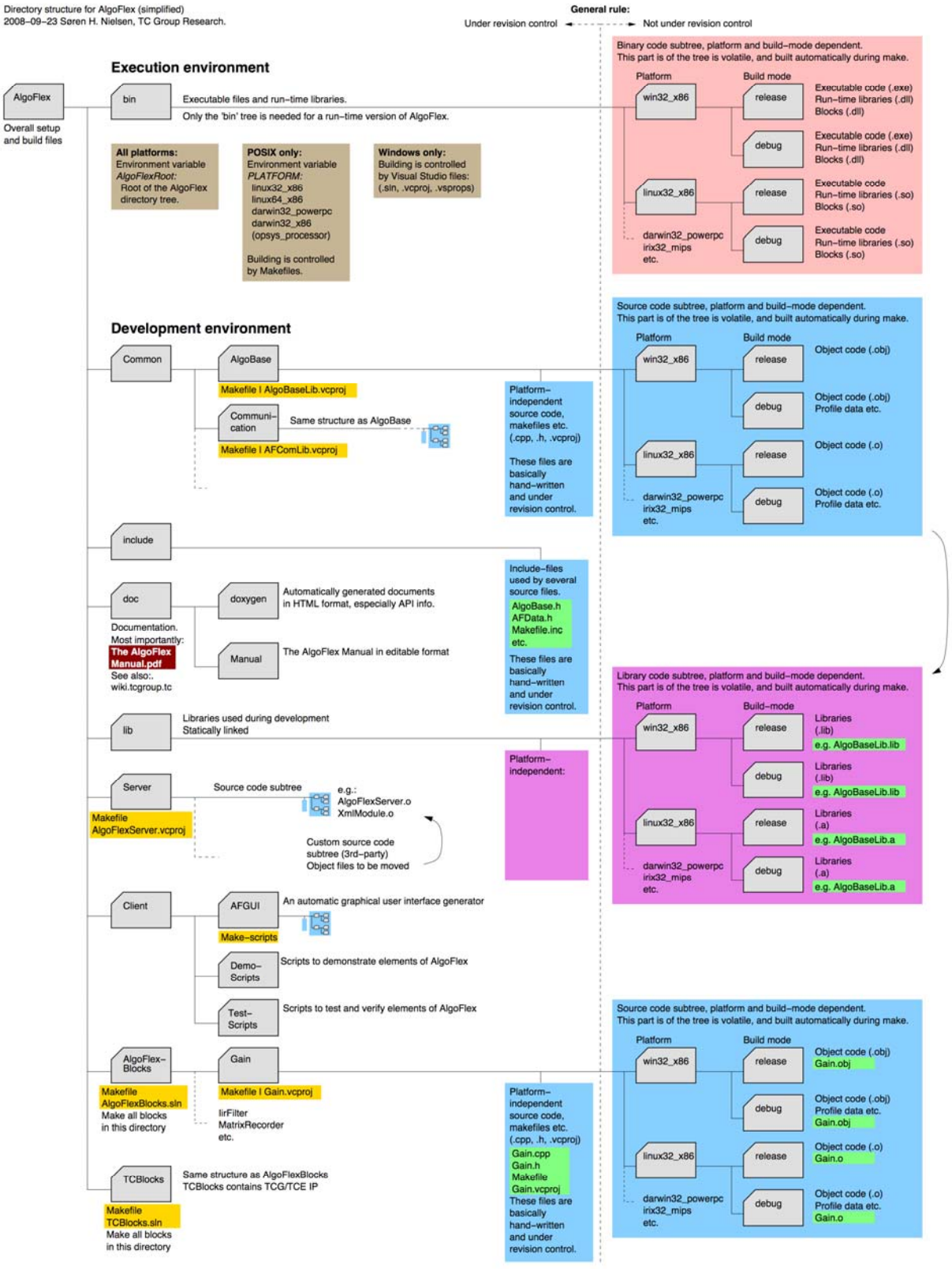
10.5.2 System requirements for building from source

In order to build the AlgoFlex system, and write new blocks, the following system components are required:

- Windows:
  - Windows 2000(SP4) or XP(SP1)
  - Visual C++ 8.0 SP1 (part of Visual C++ 2005 Express/Pro)
- Linux:
  - GCC 4.0 (3.x is no longer supported)
  - Sun Java 1.5.x - only tested in 64-bit mode (Intel EM64T/AMD64)
  - GNU Make 3.8 - standard on newer systems
  - The GNU debugger is recommended, together with the DataDisplay Debugger, ddd.
  - libjack.so should be visible, e.g. in /usr/local/lib with LD_LIBRARY_PATH set accordingly.
- Mac OS X
  - OS 10.4 "Tiger" (for Java 1.5 support).
  - GCC 4.0
  - GNU Make 3.8 - standard on OS X
  - Support for Intel CPU, made by Jan Kuhr
  - No support for PowerPC

10.5.3 Directory structure

Although the following diagram is simplified, it illustrates the structure of the Windows / Linux specific files.



11 License etc.

The majority AlgoFlex has been developed from scratch at TC Group Research / TC Electronic. The parts of AlgoFlex that are not “home made”, are based on open source software – for instance, third party libraries for audio or MIDI device I/O, or for reading/writing audio files. Note that *open source*, in this context, does generally *not* mean GPL licensed, but simply that we have access to the source code, and the possibility of modifying it should we need to.

--- APPENDIX ---

12 AlgoFlex Client/Server Commands

[http://wiki/bin/view/TCDesign/AlgoFlexCommands?skin=plain]

12.1 Blocks handling

Command	Arguments	Description
Create	[blockID,blockName] = Create(libName,nIn,nOut)	Loads the specified library, and creates a new instance (object) of the processing block. LibName must be the filename of a dll, on the library path. Ex: libName = "IirFilt", which might return blockID = 2, blockName = "IirFilt_1". nIn and nOut specify desired the number of audio inputs and outputs.
	[blockID,blockName] = Create(libName,nIn, nOut,blockName)	As above, but with a user-specified (rather than auto-generated) block name. The block name must be of the same form as C/C++ identifier (e.g. variable and function names).
Destroy	Destroy(blockID/blockName)	Destroys the block instance. (First the block's audio and parameter connections must be unconnected). The server automatically unloads the dll corresponding to block (so that the file is no longer locked and may be replaced), when a block's InstanceCount == 0.
DestroyAll	DestroyAll()	Destroys all block instances, together with their audio- and parameter-connections. Start from scratch.
GetBlocks	[blockName1,blockName2,...] = GetBlocks()	Get a list of all existing block-instances.
GetBlockName	blockName = GetBlockName(blockID)	Returns the name of the block instance corresponding to the blockID.
GetBlockID	blockID = GetBlockID(blockName)	Returns the blockID corresponding to the name.
GetBlockLibName	[libName] = GetBlockLibName(blockID/blockName)	Get the block's libName, i.e., the block type that the given block is an instance of.

12.2 Audio connections

Command	Arguments	Description
ConnectAudio	ConnectAudio(srcBlockID, srcPort, dstBlockID,dstPort)	A source/destination port can be specified either as a port name (string), as returned by GetAudioInputPorts / GetAudioOutputPorts, or as a vector with one or more integers, in the range [1..numChannels]. The number of channels, in the source and destination, must be the same. Ex: ConnectAudio(mySrcBlockID,"StereoOutput-1", myDstBlockID,[1 2])
	ConnectAudio(srcBlockName, srcPort, dstBlockName,dstPort)	
UnconnectAudio	UnconnectAudio(srcBlockID, srcPort, dstBlockID,dstPort)	Remove the specified audio connection.
	UnconnectAudio(srcBlockName, srcPort, dstBlockName,dstPort)	
GetIODim	[nIn,nOut] = GetIODim(blockID/blockName)	Returns the total number of audio input/output channels of the specified block. *To do: return the port names corresponding to any channel-sets defined in the block.*
GetInputConnection	[blockName,chan] = GetInputConnection(blockID/blockName, channel)	Returns the input-connection...
GetOutputConnections	[blockName1,chan1,...] = GetOutputConnections(	Returns the output-connections...

	blockID/blockName, channel)	

12.3 Parameter connections

Command	Arguments	Description
GetDataParmNames	[parmName_list] = GetDataParmNames(blockID/blockName)	Returns the list of available data-parameters for the specified block. Data-parameters may be used with setData/getData and at the recepient end of a parameter connection.
GetTransmitterParmNames	[parmName_list] = GetTransmitterParms(blockID/blockName)	Returns the list of available transmitter-parameters for the specified block. Transmitter-parameters may be used at the transmitter end of a parameter connection.
ConnectParm	ConnectParm(srcBlockID ,transmitterParm, dstBlockID,dataParm)	Make a parameter connection between the specified transmitter parameter and data parameter.
UnconnectParm	UnconnectParm(srcBlockID, transmitterParm, dstBlockID,dataParm)	Remove the parameter connection between the specified transmitter parameter and data parameter.
GetParmConnections-FromSource	[[blockname1, parm1], ...] = GetParmConnectionsFromSource(srcBlockID, transmitterParm)	
GetParmConnections-ToDestination	[[blockname1, parm1], ...] = GetParmConnectionsToDestination(dstBlockID, dataParm)	

12.4 Parameter values and properties

Command	Arguments	Description
SetData	SetData(blockID/blockName ,parmName, data)	Transfer the data to the specified data-parameter. Data may be scalar, string, or vector.
GetData	data = GetData(blockID/blockName, parmName)	Reads the data from the specified parameter.
SetDataParmProperty	SetDataParmProperty(blockID/blockName, parmName, propertyName ,data)	Assign the data to the specified property of the specified data-parameter.
GetDataParmProperty	[data] = GetDataParmProperty(blockID/blockName, parmName, propertyName)	
GetDataParmNames	[parmName1,parmName2,...] = GetDataParmNames(blockID/blockName)	Returns a list of names of registered data-parameters.
GetDataParmPropertyNames	[propName1,propName2,...] = GetDataParmPropertyNames(blockID/blockName, parmName)	Returns a list of names of existing properties of the specified data-parameter.

12.5 Save/Load parameters and algorithms

Command	Arguments	Description
XMLLoadDiagram	XMLLoadDiagram(XMLdoc)	Create new blocks with the names, input/output-dimensions, audio connections and parameter connections as described in the XML document. The XML document is passed as one long string.
	XMLLoadDiagram(XMLdoc, [blockName1,blockName2,...])	As above, but creating only the block(s) specified and their inter-connections. If a block with the specified name does not exist in the XML

		document, an error will be thrown and the loading process will abort.
XMLLoadParameters	XMLLoadParameters(XMLdoc)	Replace the data-parameter values of one or more parameters of one or more existing blocks with those described in the XML document. The names of blocks in the XML documents must match the names of existing (created) blocks else the loading process will abort.
	XMLLoadParameters(XMLdoc, [blockName1,blockName2,...])	As above, but loading only the data parameters for the specified block(s).
	XMLLoadParameters(XMLdoc, [[blockName1,parmName1], [blockName2,parmName2], ...])	As above, but loading only the specified parameter(s). The *N* pairs of (blockName,parmName) are specified as a *Nx2* string array.
XMLSaveDiagram	[string] = XMLSaveDiagram()	Return a string that contains the XML description of the current block diagram.
	[string] = XMLSaveDiagram([blockName1, blockName2,...])	As above but the string will contain the XML description of the specified blocks. The only audio/parameter links that will be saved are those of the specified blocks. If a block with the specified name does not exists it will throw an error and abort the saving process.
XMLSaveParameters	[string] = XMLSaveParameter()	Return a string that contains the XML description of the parameters of the blocks on the server.
	[string] = XMLSaveParameter([blockName1, blockName2,...])	As above but the string will contain the XML description of the parameters of the specified blocks. If a specified block does not exist it will throw an error and abort the saving process.
	[string] = XMLSaveParameter([[blockName1, parameterName1], [blockName2, parameter2],...])	As above but the string will contain the XML description of the specified parameter(s). The *N* pairs of (blockName,parmName) are specified as a *Nx2* string array.
XMLLoadDiagramFile	XMLLoadDiagramFile(fileName)	Loads a diagram directly from a file. Note that the specified path must be valid from from the server's point of view.
	XMLLoadDiagramFile(fileName,blockList)	
XMLLoadParametersFile	XMLLoadParametersFile(fileName)	Loads parameters directly from a file. Note that the specified path must be valid from from the server's point of view.
	XMLLoadParametersFile(fileName, blockList)	

12.6 Execution

Command	Arguments	Description
SetSampleRate	SetSampleRate(fs)	The sample-rate must be specified by the client, at any time, but before calling start.
Prepare	Prepare()	Prepare the blocks for start - this is a blocking call. Prepare must be called for all servers, before start is called.
Start	Start()	Start the audio and parameter processing (non-blocking), after checking that all blocks in the execution sequence are ready.
	Start(duration)	Run for a certain duration (seconds).
	Start(duration, isBlocking)	As above, but specifying whether the command is blocking.
Stop	Stop()	Stop the audio and parameter processing.
SetExeSeq	[blockID_list] = SetExeSeq([blockID1,blockID2,...,blockIDN])	Specify the execution sequence of the blocks. Each created block must be in the sequence 0 or 1 times. Returns the execution-schedule

		constructed by the engine, as a sequence of blockIDs. The execution-schedule depends on the calls to SetExeSeq and SetExeRateFactor.
GetExeSeq	[blockID-list] = GetExeSeq('blockID')	Get the current execution sequence.
	[blockName-list] = GetExeSeq('blockName')	
SetExeRateFactor	SetExeRateFactor(blockID/blockName, factor)	After calling SetExeSeq, one or more blocks may be set to execute with a rate that is fs*n or fs/n, for n=1,2,3..., where fs is the audio sampling rate. Ex.: factor = 2, or factor = 1/2.
GetExeRateFactor	factor = GetExeRateFactor(blockID/blockName)	Get the block's local execution rate factor (as set by SetExeRateFactor).
DetermineExecutionOrder	exeorder = DetermineExecutionOrder(startblockID)	Suggest an execution order by starting at the block with ID startblockID and traversing the graph recursively by it's audio and parameter links. If startblockID is not specified, the traversal will start at the MemoryStream.
	exeorder = DetermineExecutionOrder()	

12.7 Code generation

Command	Arguments	Description
GenerateCode	GenerateCode(CodeType, Name, OutputFileName, ...)	Generates source code for running the algorithm on a native platform. CodeType = 'Algorithm Class C++' or 'Inline Mode, C++'. Common for all CodeTypes are the parameters Name and OutputFileName, which must be specified. On top of that each CodeType has it's own list of optional parameters.
	GenerateCode('Algorithm Class C++', Name, OutputFileName, OptimisationStrategy)	Generates a .cpp and a .h file in which the algorithm currently instantiated on the server is implemented as a C++ class. OptimisationStrategy = * 'Baseline' (simple strategy, based on inlining each block's SignalProcess exactly once, according to the execution-order), * 'OtherStrategy' ... *TODO*
	GenerateCode('Inline Mode C++', Name, OutputFileName)	Generates an inline file with a mode consisting of parameter data for all the blocks in the current exesequence.
	GenerateCode('Inline Mode C++', Name, OutputFileName, [blockName1, ..., blockNameN], Exclude)	A list can be specified of blocks who's parameters are to be included in the mode. Alternatively, if the Exclude flag is set to true the specified blocks will be excluded from the mode.
	GenerateCode('Inline Mode C++', Name, OutputFileName, [[blockName1,parmName1], ..., [blockNameN,parmNameN]], Exclude)	Alternatively a list can be supplied of specific parameters to be included in the mode. The *N* pairs of (blockName,parmName) are specified as a *Nx2* string array. As above, if the Exclude flag is set to true the specified parameters will be excluded from the mode.

12.8 Help and verification

Command	Arguments	Description
Help	Help()	General server help text, with list of the most common server commands.
	Help(serverCommand)	Help text about a given server command. [same as above, for now]
	Help(libName)	Help text about a block library (i.e., a processing block type).

BlockInfo	BlockInfo(blockID/blockName)	Get information about the specified processing block, i.e., its state, parameter values and connections.
	BlockInfo(blockID/blockName, verbosity)	
GetProbeNames	[probe1,probe2,...] = GetProbeNames(blockID/blockName)	Return a list of the registered probe names.
ReadProbe	[data] = ReadProbe(blockID/blockName, probeName)	Return the current value of the specified probe.
GenerateGraph	GenerateGraph(shortcuts, hiddenblocks, what, size, format, filename)	Generates a visual representation of the algorithm, showing the audio and parameter connections between the blocks and colorizing the blocks by their respective execution rates. The shortcuts parameter contains a list of named connection points which can be used to replace long connection lines. If not used, pass an empty list {}. The hiddenblocks parameter contains a list of blocks that you don't want to have in the graph. An example of such could be a block with a very large number of parameters that would tend to clutter the graph with wires. Any wires connected to blocks that are hidden will be replaced by shortcuts to the outputs of the hidden block, but the hidden block itself will not be shown in the graph. For an example of its use, see the RevNative.m file in the Rev4 project. 'what' is a string that speficies the types of wires generated. If the string contains the substring 'Audio', audio-connections will be visualized. If the string contains the substring 'Parms', data parameter connections will be shown. The default size is 10 (inches) and the default format is Encapsulated PostScript 2 ('eps'). If a file name is specified, output will be written into that, otherwise a temporary file will be used.
	GenerateGraph({}, {})	Generate graph showing all blocks, audio and data parameter wires.
	GenerateGraph({}, {'CoeffComput'}, 'Parms')	Generate graph showing all blocks except 'CoeffComput' and show only parameter connections.
	GenerateGraph({'FEED_DELAY_L' 3 'LER1' 'DELAY1_L' 2 'RRT32' }, {}, 'Parms')	Generate graph showing all blocks, but replace the wires starting at block FEED_DELAY_L's output 3 and DELAY1_L's output 2 by shortcuts with the labels 'LER1' and 'RRT32', and show only audio connections.

12.9 Misc.

Command	Arguments	Description
Echo	data = Echo(data)	Transmits one argument consisting of data of the permitted types; the engine then returns the same data. Used for testing the data coding and engine communication.
	[data1,data2,...] = Echo(data1,data2,...)	
Profile	[profilingData...] = Profile(command)	command = 'ON', 'OFF', 'RESULTS'
SetSystemParameter	SetSystemParameter(sysParmName, data)	Set an internal "engine" parameter. Generally, this should only be done on a fresh server (i.e. with no blocks instantiated).
GetSystemParameter	data = GetSystemParameter(sysParmName)	...
Quit	Quit()	The client/server-connection is closed, and then the server process terminates (after releasing its allocated resources).

12.10 Master-server (distributed processing)

Command	Arguments	Description
GetMachineName	serverHostname = GetMachineName()	...
InitMasterServer	InitMasterServer()	...
IsSlaveReady	IsSlaveReady()	...
SlaveStart	SlaveStart()	Syntax as Start() command

12.11 Client commands (AlgoFlexClient for MATLAB)

Command	Arguments	Description
OpenServCon	serverID = AlgoFlexClient('OpenServCon', serverIpAddr, serverPort)	The client program opens a TCP/IP socket connection to the server. The serverIpAddr can be either a host-name or a numeric IP address. The serverPort is the port number, on which the server listens.
CloseServCon	AlgoFlexClient('CloseServCon', serverID)	The client/server connection (established by OpenServCon) is closed. Why is the syntax different in this command? Other AlgoFlexClient commands use srvId as the first parameter. 2008-04-11 Main.SoerenNielsen
MultiStart	AlgoFlexClient([serverID1,serverID2,...], 'MultiStart')	Sends the Prepare command to one or more servers, followed by the Start command.
RegisterClientCallback	AlgoFlexClient(serverID, 'RegisterClientCallback', blockID, transmitterParm, callback)	Registers a client callback with the server, where callback is either - a) the name of a Matlab function taking one input argument, or b) a Java object which implements the ParameterCallback interface.
AlgoFlexEval	outputData = AlgoFlexEval(srvId, algoExeSeq, parmsSettings, inputData, inputRouting, outputRouting, fs[,nOutputSamples [,nRunSamples [,outputExeRateFactor]]])	AlgoFlexEval can be used to 'evaluate' one or more connected AlgoFlex blocks as a function, i.e., specify a data matrix as input, then let the block(s) process the data, and return the resulting output. See `help AlgoFlexEval` (MATLAB) for details.
SetDatas	SetDatas(serverID, blockID1,parmName1,value1, ..., blockID_n,parmName_n,value_n)	SetDatas is a simple convenience function to replace multiple, similar 'SetData' commands issued via AlgoFlexClient.
	SetDatas(serverID, blockID ,parmName1,value1, ..., parmName_n,value_n)	
	SetDatas(serverID, parmsSettings)	
GetDatas	parmsSettings = GetDatas(serverID, blockID)	GetDatas returns the current values of all data-parameters of the specified block(s). GetDatas is a simple convenience function to replace multiple, similar 'GetData' commands issued via AlgoFlexClient. The cell array, returned by GetDatas, is compatible with the SetDatas function. ParmsSettings: a cell array of the form: {blockName1,parmName1,parmValue1; blockName2,parmName2,parmValue2; ...}
	parmsSettings = GetDatas(serverID, [blockID1 blockID2 ...])	
	parmsSettings = GetDatas(serverID, {blockName1 blockName2 ...})	

"Client commands" either mean commands that only affect the client (not the server), or commands that are 'macros' which map into a common sequence of server commands (such as MultiStart).

12.12 Not implemented / dropped

Command	Arguments	Description
SetLibraryPath	[libName_list,libPath_list] = SetLibraryPath(libPath)	Specify the directory path on which the processing block dll's are found. If multiple dll's (in different directories) have same name, the first instance is used. The specified file-systems must be visible from the engine process (i.e., no local client files). Ex: libPath = "C:\AlgoFlex;H:\NewBlocks\dll".
GetParmConnections	[blockID_list] = GetParmConnections(blockID,parmName)	Returns the list of blocks (if any) that are connected to the specified data/transmitter parameter in the specified block.
GetAudioConnections	[blockID_list] = GetAudioConnections(blockID/blockName,port)	Returns the list of blocks (if any) that are connected to the specified port in the specified block.

-- EsbenS - 09 Dec 2005 (Created)

13 Standard Blocks

[http://wiki/bin/view/TCDesign/AlgoFlexStandardBlocks?skin=plain]

The following directory shows the help text of each of the standard AlgoFlex blocks.

- The Standard Blocks in AlgoFlex
  - AllpassSparse
  - AsioStream
  - AudioStream
  - Bypass
  - CallbackTestBlock
  - ChannelCombiner
  - Delay
  - DialButtons
  - DownFir
  - DummySync
  - FastConv
  - FilePlayer
  - FileRecorder
  - Gain
  - GainMatrix
  - GainMatrixSparse
  - GainSum
  - Hex32FileReader
  - Hex32FileWriter
  - IirFilter
  - MatrixPlayer
  - MatrixRecorder
  - MemoryStream
  - Meter
  - MidiControl
  - MultitapDelay
  - NoiseInjector
  - Oscil
  - ParmCombiner
  - ParmMapper
  - ParmMorpher
  - ParmProdSum
  - Parms2Audio
  - Prod
  - RIAAFilter
  - ReverbFilter
  - RmsWindow
  - Sequencer
  - ShapeQuantizer
  - SinGen
  - TCPDestination
  - TCPSource
  - TextFileWriter
  - TwoTimeConst
  - UpFir
  - VssColorFilter

13.1 AllpassSparse

```
BLOCK LIBRARY: AllpassSparse
The AllpassSparse block implements a simple sparse allpass filter - the type
used in certain reverb algos.
The structure is similar to that of a 1st-order allpass filter, but the length of
the delay, in the feedback, is a tunable parameter.
At this point, the block only implements one filter section, processing one channel.

The block has the following data-parameters:
* G:
```

  The gain value.
* N:
  The delay time (and thereby the filter order), specified in seconds.
* Function:
  Implementation of the filter, 'Ordinary' (default) or 'Alternative'

13.2 AsioStream

BLOCK LIBRARY: AsioStream
The AsioStream block will input/output n-channel audio using the ASIO interface.
Parameters are:
* DriverList:
  A list of the available ASIO drivers.
* DriverName:
  The user assigns one of the names from DriverList to DriverName.
* BufferSize:
  The block's buffer size, in samples.
* AsioInputIndex, AsioOutputIndex:
  The user selects which input and output ASIO channels to connect to the
  blocks i/o channels. These parameters accept a vector of indices.
* InputChannelInfo, OutputChannelInfo:
  The block displays channel names to the user (read-only).

13.3 AudioStream

BLOCK LIBRARY: AudioStream
The AudioStream block allows you to use your sound interfaces in AlgoFlex.
It automatcaly detects your hardware and lets you choose which device to use.

The block's parameters are:
  For information only:
  * Capabilities   : shows a list of your devices and their characteristics,
  * RawCapabilities : the same, without the nice formating,
  * DeviceList   : shows the text names of the devices,
  To be set:
  * Device   : used to set which device you want to use,
  * CardInputs   : to set the card input corresponding to each block output,
  * CardOutputs   : to set the block input corresponding to each card output,
  * BufferSize   : size of one buffer (in samples, per channel),
  * NumberOfBuffers : how many buffers to use. For sound servers that only
      support one buffer (Jack, ASIO), the block does its own buffering.
The sample rate is set automaticaly.

13.4 Bypass

BLOCK LIBRARY: Bypass
The Bypass block will simply bypass the signal in all audio channels.

Data-parameter:
* DummyDataParm - will accept and store any parameter value of any type
  and dimension.

Transmit-parameter:
* DummyTransmitterParm - will re-transmit data-objects that are assigned
  to DummyDataParm.

13.5 CallbackTestBlock

BLOCK LIBRARY: CallbackTestBlock
This CallbackTestBlock block is used for testing the callback-semantics.
The block has the following data-parameters:
* A,B,C,D,E,F,G:

13.6 ChannelCombiner

BLOCK LIBRARY: ChannelCombiner
The ChannelCombiner block will for each sample combine the signal across the
audio-rate input channels (1..N) using a function set by the user.
The result is a single audio-rate signal output.

It is possible to select which of the input channels are to be included in the
generation of the output. In case of no active inputs, the output is set to zero.

```
Data parameters:

* Function: A string to select which combination function to use. Options:
    * Min         The output is equal to the smallest of the active inputs.
    * Max         The output is equal to the largest of the active inputs.
    * MinAbs      The output is equal to the smallest of absolute values of
                  the active inputs.
    * MaxAbs      The output is equal to the largest of absolute values of
                  the active inputs.
    * Sum         The output is equal to the sum of the active inputs.
    * MeanAll     The output is equal to the artihmetric mean of all inputs.
    * MeanActive  The output is equal to the artihmetric mean of the active inputs.

* ChannelWeight:
    A vector containing weighting factors, one for each input channel.
    Typically, ChannelWeight consists of zeros and ones, but also other
    weights may be applied.
    If an element of ChannelWeight if different from zero, it means that the
    associated input channel is active.
    The default value is a vector of ones.
```

13.7 Delay

```
BLOCK LIBRARY: Delay
This Delay block implements a simple multi-channel delay.
The block has the following data-parameters:
* DelayTime:
  The duration of the delay, for all channels, specified in seconds.
```

13.8 DialButtons

```
BLOCK LIBRARY: DialButtons
The Dial&Buttons block allows you to use serial port controlers with AlgoFlex.
Don't read the rest of this message, it's only old junk from the DialButtons.
It automatcaly detects your hardware and lets you choose which device to use.

Its commands include :
  * Capabilities    : shows a list of your devices and their characteristics,
  * RawCapabilities : the same, without the nice formating,
  * Devices         : shows the text names of the devices,
  * Device     : used to set which device you want to use,
  * InMapping  : not implemented, do the routing with Connect,
  * OutMapping : not implemented, do the routing with Connect,
  * SampleRate : used to set the sample rate of the sound interface,
  * BufferSize : size of one buffer (in samples, per channel),
  * NumberOfBuffers : how many buffers to use. For sound servers that only
      support one buffer (Jack, ASIO), the block does its own buffering.
```

13.9 DownFir

```
BLOCK LIBRARY: DownFir
The DownFir block implements implements FIR filtering, followed by sample
skipping (0th order hold). Only the outputs actually needed are computed.
The block has the following data-parameters:
* Coefficients:
  The FIR-filter coefficients as a real-valued vector.
* DownsamplingRate:
  The downsampling rate: 1,2,3,4,... (1 creating a normal FIR filter)
```

13.10 DummySync

```
BLOCK LIBRARY: DummySync
The DummySync block provides a simple way of synchronising an AlgoFlex server
to the system clock. It is useful for 'real-time' testing of an algorithm
without any audio I/O (device).
The block has neither inputs, outputs, nor parameters. To use the block,
simply include an instance in the algorithm's execution-sequence.
The synchronisation is based on the sample rate of the block together with
the system clock.
If a high-resolution synchronisation is required, please use one of the
audio-device blocks instead, such as AudioStream or AsioStream.
```

13.11 FastConv

```
BLOCK LIBRARY: FastConv
The FastConv block implements an N-input, M-output FastConvolution FIR filter matrix.
Thus, each output-channel is the sum of all the input-channels, individually filtered.
The filter matrix is duplicated in up to O = 4 layers, of which one may be active at
any time

The CompleteSetup parameter is an NxMxO sized MATLAB Cell Array with each element
being a
column vector impulse response, e.g. h{N,M,O}.
Empty and different-length impulse responses are allowed

The Layer parameter selects the active layer. If Layer has 2 elements the second
element
is the crossfade length in samples
```

13.12 FilePlayer

```
BLOCK LIBRARY: FilePlayer
The Audio-file Player will play the n-channel audio samples supplied in a file.
Various file types are supported, via the libsnd library.
The available playback modes are: whole | segment | standby | whole* | segment*
Asterisk is infinite loop.
Parameters are
  * InputFileName :
      the name of the file to open.
  * PlayMode :
      whole   : plays the whole file.
      segment : plays a part of the file. See SegmentBorders.
      standby : plays silence.
      whole*, segment* : loops the whole file or the segment.
  * SegmentBorders :
      use this to give the bounds of the segment. The sample on the 2nd bound
      is excluded, so that a region like [0 1] can loop on 1 second exactly. If
      the In or Out bound doesn't exactly fall on a sample, it is rounded to
      the nearest. The bounds validity is checked before Start.
      You can use the value 'Inf' to play to the file's end.
  * BufferSize : number of multichannel samples to hold in the buffer.
  * FileSize : the size of the audio file, in sample frames.
      This read-only parameter is set when the InputFileName is specified.
Transmitter parameter:
  * CurrentPosition : A value in seconds, continuously updated (1 Hz)
```

13.13 FileRecorder

```
BLOCK LIBRARY: FileRecorder
The Audio-file Recorder will write n-channel audio samples to an audio-file.
The file name is set via the OutputFile parameter. It is opened at Start and
closed at Stop. At each Start the file is overwritten, so don't start again
when the take is good.
Parameters are...
  * RecordMode
      standby  : to disable the FileWriter. The file will not be altered.
      roll_on  : the FileWriter records from Start to Stop. See 'Pause'.
      sched_end : record only during a certain amount of time
  * Pause : valid in roll_on mode only. Set to 'true' or 'false'.
            You can set it before Start.
  * OutputFileName : the name of the file to wrote to
  * Format : '[FILE_TYPE]/[BIT_DEPTH]/[SAMPLING_RATE]' or 'default'
      FILE_TYPE : wav | w64 | aiff | mat4 | mat5 | iff |
                  voc | ircam | nist | paris | raw
      BIT_DEPTH : <n>bit (ints) | <n>uint (unsigned ints) | <n>flt (floats)
                 integers depths are 8|16|24|32, float depths are 32|64
      SAMPLING_RATE : <f>hz
      default : is equivalent to 'wav/16bit/44100hz'
  * BufferSize : number of multichannel samples to hold in the buffer.
  * RecordLength : in sched_end mode, sets the recording duration (s).

Default values : roll_on, false, "", wav/16bit/44100hz, 2048, 0.
```

13.14 Gain

```
BLOCK LIBRARY: Gain
```

```
The Gain block will amplify or attenuate the signal in all channels with a
constant gain.

Data-parameters:
* GainDb - sets the gain, in dB.

Transmitter-parameters:
* ChannelClip - can optionally be used to detect clipping [deprecated].
```

13.15 GainMatrix

```
BLOCK LIBRARY: GainMatrix
The GainMatrix block implements an N-input, M-output mixer.
Thus, each output-channel is the weighted sum of all the input-channels.
The Matrix parameter sets the linear gain, as follows:
  [ [ g11 g21 ... gN1 ]
    [ g12 g22 ... gN2 ]
    [         ...     ]
    [ g1M g2M ... gNM ] ]
```

13.16 GainMatrixSparse

```
BLOCK LIBRARY: GainMatrixSparse
The GainMatrixSparse block implements an N-input, M-output mixer, with a sparse
mixing-matrix. Each output-channel is the weighted sum of some of or all of the
input-channels.
* Data-parameters:
  MatrixSparse defines how the N-inputs are summed and weighted to get the
  M-outputs. Both input- and output-indices, below, are 0-based.
  [ [ outpt0 inptIdx gainx]
    [ outpt0 inptIdy gainy]
    [          ...         ]
    [ outpt3 inptIdz gainz]
    [          ...         ]
    [ outptM inptIda gaina] ]
```

13.17 GainSum

```
BLOCK LIBRARY: GainSum
The GainSum will multiply two sample streams and add a constant gain in dB.
The resulting products are added to form a single output channel.
An additional gain factor is applied to the output.
A typical application is in (possibly muliband) dynamics processing.

Inputs:
The input channels are organised as pairs of samples to be multiplied,
  e.g. signal1 gain1 signal2 gain2.

Output:
A single channel containing the sum of products,
  e.g. (signal1*gain1 + signal2*gain2) * pow(10, GainDb*0.05)

Data-parameters:
* GainDb - sets an additional gain, in dB.
```

13.18 Hex32FileReader

```
BLOCK LIBRARY: Hex32FileReader
The Text-file Reader will read the n-channel data from a 32 bit hex ascii-file.
Number of columns in the data file = N
Number of output channels = M
M > N ==> zeros are sent to the remaining M-N output channels
M < N ==> last N-M columns are skipped in the data file
Ascii string seperators can be <space>, <tab>, ';' or ','
LSB corresponds to 2^-23 = 1.19209289550781e-7
Valid numerical range = -256.0 -> 255.999999880791
The file is opened and closed via the InputFile parameter:
Use a file-name as argument, for opening a new file, and use 'close' for closing.
```

13.19 Hex32FileWriter

```
BLOCK LIBRARY: Hex32FileWriter
The Text-file Writer will write the n-channel audio samples to a 32 bit hex ascii-
file.
```

```
LSB corresponds to 2^-23 = 1.19209289550781e-7
Valid numerical range = -256.0 -> 255.999999880791
The file is opened and closed via the OutputFile parameter:
Use a file-name as argument, for opening a new file, and use 'close' for closing.
```

13.20 IirFilter

```
BLOCK LIBRARY: IirFilter
The IirFilter block implements a cascade of second order IIR filters.
The block has the following data-parameters:
* Coefficients:
  To set the filter-coefficients, a 3D-matrix is used, having the dimensions:
  [nBiquads,6,nChannels].
* FlushStates:
  A boolean to specify whether the filters' state is reset when coefficients
  are updated, and at every Start.
```

13.21 MatrixPlayer

```
BLOCK LIBRARY: MatrixPlayer
The Matrix Player will play the n-channel audio samples supplied in a matrix.
The block has the following data-parameters:
* Samples
  The nSamples x nChan matrix.
* PlaybackMode
  The available playback modes are: single | repeat | standby.
```

13.22 MatrixRecorder

```
BLOCK LIBRARY: MatrixRecorder
The Matrix Recorder will record a segment of n-channel audio samples into a matrix.

The block has the following data-parameters:
* SamplesToRecord
  The number of samples to record.
* RecordMode
  The available recording modes are: single | repeat | standby.
* Samples
  After recording, you can obtain the recorded samples in an nSamples x nChans
  matrix, using a GetData('Samples') command.
  The Samples parameter is updated at a Stop command and if the block is set in
  standby mode.
```

13.23 MemoryStream

```
BLOCK LIBRARY: MemoryStream
The MemoryStream block is used as the I/O-block in algorithms for the Native
Processing Plugin Framework. This block provides buffers for a 'chunk' of the
algorithm's input- and output-samples.

Data-parameter:
* ChunkSize - the number of (multi-channel) samples to exchange and process at
  the time.
```

13.24 Meter

```
BLOCK LIBRARY: Meter
The Meter block will perform a minimum- or maximum-hold for each input channel.
Optionally, the absolute value of the signal will be used.

This block offers the following parameters:
* MeterMode:
  Set the mode as either MinHold, MaxHold, MinAbsHold, MaxAbsHold, Mean, RMS,
  or Last.
* MeterFrequency:
  The frequency (Hz) with which the meter values are transmitted.
* GlideMode:
  Whether the meter value is also a function of the previous meter value.
  Available modes: None, Linear (implemented so far).
* GlideParmUp, GlideParmDown:
  In Linear mode: GlideParmUp and GlideParmDown specify the maximum change per
  second, when the current value is moving up or down, respectively.

* MeterData:
```

```
  The transmitter-parameter which periodically transmits the updated meter
  values for all input channels. The meter-data for all channels is thus
  received by the client or another block.
```

13.25 MidiControl

```
BLOCK LIBRARY: MidiControl
The MidiControl block routes MIDI Control Change and System Exclusive messages
from MIDI to a TransmitParameter or from a DataParameter to MIDI.

The block has the following data-parameters:
* Devices:
  The available MIDI ports names.
* MidiIn & MidiOut:
  Set these to choose the device/port you want to use in each direction
  You can use 'none' if you want (for ex. if you only want input and no output).
* InputRoutes:
  Sets which MIDI event should be sent to data paramerters.
  ex : {'1cc/2ch/myparm1'; 'allcc/3ch/myparm2'; 'sysexin/myparm3'} will route
    Control Change message for controller 1 of channel 2 to myparm1,
    Control Change message for all controllers of channel 3 to myparm2,
    System Exclusive messages to myparm3.
  Special token : 'allcc' (all Control Change ctrls) and 'allch' (all channels).
* OutputRoutes:
  Sets which MIDI event should be sent to data paramerters.
  ex : {'myparm4/7cc/2ch'; 'myparm8/4cc/8ch'; 'myparm9/sysexout'} will route
    myparm4 to Control Change message for controller 7 of channel 2,
    myparm8 to Control Change message for controller 4 of channel 8,
    myparm9 to System Exclusive.
  The '14bit'(default 7) allows to give output range for CC controllers 0-31.
  This is only possible when the block is compiled without the TC_MIDI flag.

NOTE : The InputRoutes and OutputRoutes can be set while the engine is running.
```

13.26 MultitapDelay

```
BLOCK LIBRARY: MultitapDelay
This MultitapDelay block implements a multi-tap delay line.
The number of delay taps is set at block instantiation time as the number of inputs
minus one.
For performance reasons, there is _no_ check that the Delay Times are less than
MaxDelayTime, so
you must make sure the MaxDelayTime is sufficiently large.

Audio Rate Inputs:
* Signal (channel 1).
* Delay time (channels 2 to #Taps+1). Controls the delay time, in seconds, of the
output channel
  of the same number.

Audio Rate Outputs:
* Unprocessed signal output (channel 1).
* Delayed signal outputs (channels 2 to N+1).

Data-parameters:
* MaxDelayTime:
  The maximum delay-time in seconds. Changing this parameter (re-)allocates the delay-
line's buffer.
* InterpolationMethod:
  Nearest, Linear, Parabola, Sinc.

Tip: If you need the delay time(s) to be adjusted 'live', but to be set by parameters
(rather than
     sample-synchronous), see the Parms2Audio block.
```

13.27 NoiseInjector

```
BLOCK LIBRARY: NoiseInjector
The NoiseInjector block will simply add some noise to the signal in all audio
channels.
The noise on the different channels will differ but is not guaranteed to be
uncorrelated.
```

```
Data-parameter:
* NoiseGain - the gain of the added noise
* NoiseOffs - the offset of the added noise

Transmit-parameter:
  (none)
```

13.28 Oscil

```
BLOCK LIBRARY: Oscil
The Oscil block is a general oscillator. It can be used as an LFO or as
a simply wavetable synthesizer.
Each audio output corresponds to a separate 'phase-tap'.
The first audio input may be used to trigger the starting-phase. Positive edge
triggered.
The second audio input may be used for real-time frequency control (FM) in Hz

The output is scaled and offset in typical LFO style:
  1) As a default, the output moves between 0 and Ampl.
  2) The Ampl parameter indicates the peak-to-peak amplitude.
  3) The Offset parameter indicates a _deviation_ from the default offset of
     Ampl/2.

Data-parameters:
* Freq:
  The frequency in Hz.
* Ampl:
  The peak-to-peak amplitude (a multiplier on the basic waveform), for each output.
* Offset:
  An optional constant offset (added *after* Ampl is multiplied), for each output.
* StartPhase:
  A starting phase for each output. Each number is a normalised phase, with
  the interval [0; 1] corresponding to one period. Phases outside this interval
  will wrap around.
* Waveform:
  Sine, Triangle, Square, UpSaw, DownSaw, WaveTable.
* Wavetable:
  A vector of the samples of one period of a waveform.
  Used, when Waveform = 'WaveTable'
```

13.29 ParmCombiner

```
BLOCK LIBRARY: ParmCombiner
The ParmCombiner block will -
- split vector-parameters into multiple scalar transmitter-parameters,
  OR
- merge multiple scalar data-parameters into one vector transmitter-parameter.

The CombinerRatio parameter must be assigned a value that doesn't change in the
block's lifetime. This should happen as the 'next thing' after the block is created.
The InputParm- and OutputParm-parameters of the block cannot be used before
CombinerRatio has been assigned a value.

Data-parameters:
* CombinerRatio
  - to map 3 parameters to 1, set CombinerRatio = 3
  - to map 1 parameter to 3, set CombinerRatio = 1/3
* InputParm1          (in case CombinerRatio = 1/N)
* InputParm1,2,...,N  (in case CombinerRatio = N)

Transmit-parameters:
* OutputParm1,2,...,N (in case CombinerRatio = 1/N)
* OutputParm1         (in case CombinerRatio = N)
```

13.30 ParmMapper

```
BLOCK LIBRARY: ParmMapper
The ParmMapper block will transform parameter values by computing a selected
mapping function, with the specified coefficients.

Data-parameters:
```

```
* AvailableMappingTypes
  A string array: {MappingType1_name, MappingType1_description;
                   MappingType2_name, MappingType2_description; ...}
* MappingType
  One of the names listed in AvailableMappingTypes.
* MappingCoef
  The coefficient vector used in computing the mapping.
* InverseMapping
  Boolean to determine whether the inverse mapping is computed.
* ParmX
  The 'input' parameter value, or vector of values.

Transmit-parameters:
* ParmY
  The 'output' parameter value, or vector of values. A new event is transmitted,
  each time the ParmX receives a new value.
```

13.31 ParmMorpher

```
BLOCK LIBRARY: ParmMorpher
The ParmMorpher block will 'morph' 2 vector-parameters into a hybrid of the 2.

Data-parameters:
* A - a real-valued vector
* B - a real-valued vector
* X - a scalar to control the morphing in the range [0..1].
      X=0 -> AB=A, X=1 -> AB=B, X=0.5 -> AB=(A+B)/2.

Transmit-parameters:
* AB - a real-valued vector
```

13.32 ParmProdSum

```
BLOCK LIBRARY: ParmProdSum
The ParmProdSum block will compute the sum or product of two parameter
values, and transmit the result.

Data-parameters:
* OperatorType
  'Sum' (default) or 'Prod'.
* ParmX1
  The first 'input' parameter value.
* ParmX2
  The second 'input' parameter value.

Transmit-parameters:
* ParmY
  The 'output' parameter value. A new event is transmitted, each time
  a ParmX receives a new value.
```

13.33 Parms2Audio

```
BLOCK LIBRARY: Parms2Audio
The Parms2Audio block will convert scalar data-parameters into (constant)
audio signals.

When the block is instantiated, data-parameters are created, named
Chan1, Chan2, ..., ChanN (N = number of output-channels).
If the data-parameter Chan1 is given the value X, then the value X will be
continually sent on the block's audio output channel 1, etc.

(Hint: Need an Audio2Parms block instead? Have a look at the Meter block!)
```

13.34 Prod

```
BLOCK LIBRARY: Prod
The Prod block will output the product of the signals on two or more
input channels. Three different types of products can be calculated,
determined by the value of data-parameter Mode.
If the block is instantiated with 2 output-channels, the 2nd channel will
output the negated version of the 1st output's value.

* Mode:
```

```
     'Numerical': Inputs are simply multiplied (default mode).
     'Logical'  : Inputs are treated as logical values, i.e. (x>=1.0) ==> true,
                  and then AND'ed together as the output. The output is always 0 or 1.
     'Bitwise'  : Inputs are cast into unsigned integers, and their bitwise AND is
                  computed. The output is always a non-negative integer.

Hint: Need to multiply parameter-values? See the ParmProdSum block.
```

13.35 RIAAFilter

```
BLOCK LIBRARY: RIAAFilter
The RIAAFilter block applies the playback correction filter
used for vinyl records (LPs and singles).

Two different variations exist, according to the closely related
IEC 98 standards: With and without a subsonic cut-off filter.
The original RIAA standard and IEC 98 (1964) do not specify a subsonic
filter whereas the IEC 98 (1976) and later (1987) do.

The filter can be used with any number of audio channels.

The gain of the filter is unity at a specified frequency, normally 1 kHz.
If this frequency is set to zero, the original filter spec. is used - which
results in a gain of appr. 0.1 dB at 1 kHz.

This block has the following user parameters:
* SubsonicFilter  A boolean signalling whether the use the subsonic filter. (0|1)
* UnityGainFreq   Gain is unity at the specified frequency, normally 1 kHz.
```

13.36 ReverbFilter

```
BLOCK LIBRARY: ReverbFilter
The ReverbFilter block implements Reverb time filter used in Reverb algorithms
(Rev3,Rev4,RevNative)

The block has the following data-parameters:
* C0:
  Input filter coefficient
* C1:
  Input filter deleyed coefficient
* lpin:
  Lowpass input coefficient
* lpfb:
  Lowpass feedback coefficient
* lpout:
  Lowpass output coefficient
* hpin:
  Highpass input coefficient
* hpfb:
  Highpass feedback coefficient
* dir:
  Direct signal
```

13.37 RmsWindow

```
BLOCK LIBRARY: RmsWindow
The RmsWindow block calculates the RMS value over a sliding (time-)window.

The block has the following data-parameters:
* WindowLength:
  The length of the sliding window, specified in samples.
* Exponent:
  For example, Exponent = 2 (default) implies that RMS over the window is calculated.
* TimeResolution:
  The time resolution of the sliding window, specified as the number of
  samples per bin in the partial sums. TimeResolution = 1 is thus the finest
  resolution (default).
```

13.38 Sequencer

```
BLOCK LIBRARY: Sequencer
The Sequencer will play or record a sequence of time-stamped parameter events.
```

```
In playback-mode, the events are sent through the transmitter-parameter Playback.
In record-mode, the events are recorded from the data-parameter Record (with
'monitor').

Data-parameters:
* Mode
  'Playback' (default) or 'Record'
* Record
  The data-parameter acting as input for the events to be recorded.
* Sequence
  A data object with the format: {time_0 data_0; time_1 data_1; ...}
  where time is in seconds (from the last Start), and data can be any AFData object.
  The events must be sorted in time, so that: time_n <= time_n+1.
  Assigning an empty sequence, i.e. {}, is OK.
  Hint: In Matlab, a Sequence object is a 2D cell array.

Transmitter-parameters:
* Playback
  The sequenced events are transmitted from here, according to their
  time-stamps.
```

13.39 ShapeQuantizer

```
BLOCK LIBRARY: ShapeQuantizer
The ShapeQuantizer shapes input values between [-X:X] according to the waveform
described by wavetable and outputs an approximated high rated (this blocks execution
rate)
shaped waveform. I.e. the input waveform is shaped at a low rate and the output is
approaximated
by making liniar gliding at a high rate.
Data-parameters:
* OffsetValueIn:
  An optional constant input offset may be added to the input.
* ScaleIn:
  An optional scaling of the input signal. The scaling is carried out after adding
OffsetValueIn.
* OffsetValueOut:
  An optional constant output offset may be added to the output.
* ScaleOut:
  An optional scaling of the output signal. The scaling is carried out before adding
OffsetValueOut.
* Wavetable:
  A vector of the samples of one period of a waveform.
* DeltaDecimationRate:
  The shape functionality is optimized by having a decimation
  rate at which a high rated delta for linear gliding is calculated.
* InterpolationOff:
```

13.40 SinGen

```
BLOCK LIBRARY: SinGen
The SinGen block will generate a sine-wave.
The implemented synthesis-method is highly efficient, but does not (currently) deliver
a continuous output at frequency-updates.

Data-parameters:
* Freq - the frequency in Hz.
* Ampl - the amplitude of the sinusoid
* DC   - an optional constant offset
```

13.41 TCPDestination

```
BLOCK LIBRARY: TCPDestination
The TCPDestination block has the following data-parameters:
* Port
* RealPort
* Status
* Task
* NewParmTransmitter
* BufferSize
* MaxBufferSize
```

13.42 TCPSource

```
BLOCK LIBRARY: TCPSource
The TCPSource block has the following data-parameters:
* RemoteAddress
* RemotePort
* NewParmTransmitter
* Status
* Task
* PacketSize
* BufferSize
* MaxBufferSize
```

13.43 TextFileWriter

```
BLOCK LIBRARY: TextFileWriter
The Text-file Writer will write the n-channel audio samples to a text-file.
The file is opened and closed via the OutputFile parameter:
Use a file-name as argument, for opening a new file, and use 'close' for closing.
```

13.44 TwoTimeConst

```
BLOCK LIBRARY: TwoTimeConst
The TwoTimeConst block implements an assymetric envelope detector.
The filtering code was based on KBC's function of the same name.
The block contains the following data-parameters:
* TauAttack
  The time constant for a rising input signal.
* TauRelease
  The time constant for a falling input signal.
  Both time constants are e-based.
* FilterOrder (default: 1)
  For orders>1, the time-constants are 'shared' between the filter sections,
  resulting in total time-constants close to the set ones.
* Abs (default: false)
  Whether the input signal is abs'ed before filtering.
```

13.45 UpFir

```
BLOCK LIBRARY: UpFir
The UpFir block implements zero insertion followed by polyphase FIR filtering.
The block has the following data-parameters:
* Coefficients:
  The FIR-filter coefficients as a real-valued vector.
* UpsamplingRate:
  The upsampling rate: 1,2,3,4,...
```

13.46 VssColorFilter

```
BLOCK LIBRARY: VssColorFilter
The VssColorFilter block implements Reverb color filter used in Vss4 and Vss5

The block has the following data-parameters:
* Coefficients:

* The Coefficients should be structured like this:
* [a0, a1, a2, b0, b1, b2, c0, c1]

* See the block diagram included in the folder for this project
```

-- EsbenS - 12-Sep-2008

14 AlgoBase Class Reference

[From: AlgoFlex\doc\doxygen\html\class_algo_base.html]

The abstract base-class **AlgoBase**, from which any AlgoBlock inherits. More...

`#include <AlgoBase.h>`

List of all members.

14.1 Public Types

enum	**BlockStates** {
	IO_DIMENSION_OUT_OF_RANGE, **CONSTRUCTED**, **CONNECTED**, **PREPARED**,
	SAMPLERATE_CONFIRMED, **READY**, **STARTED**, **STOPPED**
	}
	Any block is always in one of these states. More...

enum	**UpdateOrder_t** {
	UPDATE_NEVER = -1, **UPDATE_FIRST** = 1, **UPDATE_SECOND** = 2, **UPDATE_THIRD** = 3,
	UPDATE_DEFAULT = 5000, **UPDATE_LAST** = 10000
	}
	'Special values' for the UpdateOrder parameter property. More...

14.2 Public Member Functions

virtual void	**Prepare** () throw () *Called, by the engine, prior to calling **IsReady()**.*
virtual const char *	**IsReady** ()=0 throw () *Called, by the engine, prior to calling ConfirmSampleRate() and **Start()**.*
virtual void	**SamplerateUpdate** (const string &parmName, const **AFData** &dataObject) throw (const char*) *Called, by the engine, whenever the sample-rate of the block is changed.*
virtual void	**Start** () throw () *Called, by the engine, after **IsReady()** but before the processing-loop which calls **SignalProcess()** is started.*
virtual void	**Stop** () throw () *Called, by the engine, after the the processing-loop is stopped, i.e., when **SignalProcess()** is no longer being called.*
virtual void	**WaitForNextSample** () throw () *Called, by the engine, only if this block is a Sync block.*
virtual void	**SignalProcess** ()=0 throw () *Called once per sample period (unless up/downsampled) by the engine.*
	AlgoBase (int NO_INPUT_CHANS_MIN_, int NO_INPUT_CHANS_MAX_, int NO_OUTPUT_CHANS_MIN_, int NO_OUTPUT_CHANS_MAX_, bool CAN_PROVIDE_SYNC_, const char *BLOCK_NAME_, const BlockCtorStruct *) *Constructor.*
virtual	**~AlgoBase** () *Destructor.*

int	**GetBlockID** () const *Returns the block's unique ID (on this server).*
const char *	**GetBlockInstanceName** () const *Returns the unique name of this block instance, such as "IirFilter-2".*
const char *	**GetLibName** () const *Returns the name of the library (i.e., the type of the processing block), such as "IirFilter".*
int	**GetNoInputChans** () const *Get this block's number of input channels.*
int	**GetNoOutputChans** () const *Get this block's number of output channels.*
double	**GetSampleRate** () const *The block's sample-rate, in Hz, *including* any up- or down-sampling.*
double	**GetExecutionRate** () const *The relative rate-of-execution.*
bool	**IsRunning** () const *Whether the block's **SignalProcess()** is currently in a running server's execution-loop.*
void	**SetData** (const string &parmName, const **AFData** *data) throw (const char*) *Called by the engine and/or by the processing block itself, to assign a data object to the specified data-parameter.*
void	**SetData** (const string &parmName, double data) throw (const char*) *Short-cut method, to spare the block-author allocating the **AFData** object when just assigning a double scalar.*
void	**SetData** (const string &parmName, const string &data) throw (const char*) *Short-cut method, to spare the block-author allocating the **AFData** object when just assigning a string.*
const **AFData** *	**GetData** (const string &parmName) const throw (const char*) *Returns the **AFData** object currently assigned to the specified data-parameter.*
void	**SetParmProperty** (const string &parmName, const string &property, const **AFData** *metadata) throw (const char*) *Set a given property of the specified data-parameter.*
void	**SetParmProperty** (const string &parmName, const string &property, double metadata) throw (const char*) *Set a given property of the specified data-parameter.*
void	**SetParmProperty** (const string &parmName, const string &property, const string &metadata) throw (const char*) *Set a given property of the specified data-parameter.*
void	**SetParmProperty** (const string &parmName, const string &property, const char *metadata) throw (const char*) *Set a given property of the specified data-parameter.*
const **AFData** *	**GetParmProperty** (const string &parmName, const string &property) const throw (const char*) *Returns the **AFData** object currently assigned to the specified property of the specified data-parameter.*
void	**GetDataParmNames** (vector< string > &strList) const *Constructs a vector containing the names of the data-parameters as strings.*
void	**GetTransmitterParmNames** (vector< string > &strList) const *Constructs a vector containing the names of the transmitter-parameters as strings.*
void	**GetPropertyNames** (const string &parmName, vector< string > &strList) const *Constructs a vector containing the names of all properties of the given data-parameter.*
void	**SetParmRange** (const string &parmName, double minValue, double maxValue, double defaultValue, const char *scaleType="lin", const char *unit="")

	A short-cut method for setting some parameter properties, common for scalar-valued parameters.
void	**SetParmRange** (const string &parmName, double minValue, double maxValue, const vector< double > &defaultValue, const char *scaleType="lin", const char *unit="") *A short-cut method for setting some parameter properties, common for scalar-valued parameters.*
bool	**ExistsDataParm** (const string &parmName) const *Auxilary-function to test whether a data-parameter (name) exists.*
bool	**ExistsTransmitterParm** (const string &parmName) const *Auxilary-function to test whether a transmitter-parameter (name) exists.*
bool	**ExistsParmProperty** (const string &parmName, const string &property) const *Auxilary-function to test whether a data-parameter property (name) exists.*
AFData *	**ReadProbe** (const string &probeName) const throw (const char*) *Read the current value of the specified coefficient probe.*
bool	**IsInfrastructureReady** () const *Called by the engine, prior to the first **SignalProcess()**.*

14.3 Public Attributes

const int	**NO_INPUT_CHANS_MIN** *the minimum number of input channels*
const int	**NO_INPUT_CHANS_MAX** *the maximum number of input channels*
const int	**NO_OUTPUT_CHANS_MIN** *the minimum number of output channels*
const int	**NO_OUTPUT_CHANS_MAX** *the maximum number of output channels*
const bool	**CAN_PROVIDE_SYNC** *Specify whether the block can be a sync-master block, that is, whether the block has implemented **WaitForNextSample()**.*
const char *	**BLOCK_NAME** *This string specifies the full name and version of the processing-block.*

14.4 Static Public Attributes

static const char *	**HELP_TEXT** *With this static string, the engine can obtain the block's help-text, without allocating an AlgoBlock object.*
static const float	**ALGOBASE_VERSION** *The version of the **AlgoBase** API.*

14.5 Protected Member Functions

double	**GetInputSample** (int chan) const *Used by the AlgoBlock class to obtain the next input sample, for the specified channel.*
void	**PutOutputSample** (int chan, const double x) *Used by the AlgoBlock class to return the next processed sample, for the specified channel.*
void	**PutOutputSample** (int chan, const double x, int samplesToStay) *Used in cases where a block only recalculates and outputs a new value on a certain channel*

	every n'th sample, or asynchronously.
void	**RegisterDataParm** (const string &parmName) throw (const char *) *The user-code (AlgoBlock) calls RegisterDataParm with the name of each data-parameter.*
void	**RegisterDataParm** (const string &parmName, const **AFData** **aflexargPointerAddr) throw (const char *) *Optionally, also register the address of a member-variable which should point to the data (i.e., the pointer is automatically updated by the **AlgoBase** functions).*
void	**RegisterDataParm** (const string &parmName, const string **dataPointerAddr) throw (const char *) *If the block knows that a parameter will only contain data-objects of a certain type, a member-variable can automatically be updated to point to the actual data (rather than to the enclosing **AFData** object).*
void	**RegisterDataParm** (const string &parmName, const double **dataPointerAddr) throw (const char *) *If the block knows that a parameter will only contain data-objects of a certain type, a member-variable can automatically be updated to point to the actual data.*
void	**RegisterDataParm** (const string &parmName, const complex< double > **dataPointerAddr) throw (const char *) *If the block knows that a parameter will only contain data-objects of a certain type, a member-variable can automatically be updated to point to the actual data.*
template<class T>	
void	**RegisterDataParm** (const string &parmName, FUNCTORDEF) throw (const char *) *We may the DataParmCallback argument to pass an (individual) AlgoBlock member function to each of the data-parameters registered with RegisterDataParm(.*
template<class T>	
void	**RegisterDataParm** (const string &parmName, const **AFData** **aflexargPointerAddr, FUNCTORDEF=NULL) throw (const char *) *Optionally, also register the address of a member-variable which should point to the data (i.e., the pointer is automatically updated by the **AlgoBase** functions).*
template<class T>	
void	**RegisterDataParm** (const string &parmName, const string **dataPointerAddr, FUNCTORDEF=NULL) throw (const char *) *If the block knows that a parameter will only contain data-objects of a certain type, a member-variable can automatically be updated to point to the actual data (rather than to the enclosing **AFData** object).*
template<class T>	
void	**RegisterDataParm** (const string &parmName, const double **dataPointerAddr, FUNCTORDEF=NULL) throw (const char *) *If the block knows that a parameter will only contain data-objects of a certain type, a member-variable can automatically be updated to point to the actual data.*
template<class T>	
void	**RegisterDataParm** (const string &parmName, const complex< double > **dataPointerAddr, FUNCTORDEF=NULL) throw (const char *) *If the block knows that a parameter will only contain data-objects of a certain type, a member-variable can automatically be updated to point to the actual data.*
void	**UnregisterDataParm** (const string &parmName) throw (const char *) *Unregister a previously registered data-parameter.*
template<class T, class T2>	
void	**RegisterCallbackDependency** (FUNCTORDEF, FUNCTORDEF3) throw (const char *) *Register that one callback (i.e., parameter-group) depends on another.*
void	**RegisterTransmitterParm** (const string &parmName) throw (const char *) *The user-code (AlgoBlock) calls RegisterTransmitterParm with the name of each transmitter-parameter, i.e., the "output" parameters offered by the block.*

void	**UnregisterTransmitterParm** (const string &parmName) throw (const char *) *Unregister a previously registered transmitter-parameter.*
void	**TransmitParm** (const string &parmName, const **AFData** &data) const throw () *Called from the AlgoBlock code, usually in **SignalProcess()**, whenever the named parameter has a new value.*
void	**TransmitParm** (const string &parmName, double data) const throw () *Short-cut method, to spare the block-author allocating an **AFData** object, when just transmitting a double.*
void	**TransmitParm** (const string &parmName, const string &data) const throw () *Short-cut method, to spare the block-author allocating an **AFData** object, when just transmitting a string.*
void	**RegisterPortName** (const string &portName, const vector< int > &portChans, bool isInputChans) throw (const char*) *A 'port' is more convenient way of addressing sets of audio input/output channels.*
void	**SetupProbe** (const string &probeName, const double *probeVarPtr, int probeLen=1, int stride=1) throw (const char*) *A 'probe' - or coefficient probe - allows the client to read the current value of certain private AlgoBlock variables.*
void	**SetupProbe** (const string &probeName, const int *probeVarPtr, int probeLen=1, int stride=1) throw (const char*) *A 'probe' - or coefficient probe - allows the client to read the current value of certain private AlgoBlock variables.*

14.6 Protected Attributes

const double *	**m_fs** *A pointer to the sample-rate.*

15 AFData Class Reference

[From \AlgoFlex\doc\doxygen\html\class_a_f_data.html]

AFData is the AlgoFlex data-container class. More...

`#include <AFData.h>`

List of all members.

15.1 Public Types

typedef vector< size_t >	**DimsType** *A typedef for convenience and forward-compatibility.*

15.2 Public Member Functions

	AFData () *Construct an empty object (of type AF_REAL).*
	AFData (const **DimsType** &dims, **AF_DATA_TYPE** type) *The most general constructor.*
	AFData (const **AFData** &obj) *Copy constructor.*
	AFData (size_t length, **AF_DATA_TYPE** type) *Short cut for constructing an object containing a vector.*
	AFData (double x) *Short cut for constructing an object containing only a real scalar.*
	AFData (const string &str) *Short cut for constructing an object containing only a string; the data is copied.*
	AFData (const char *str) *Short cut for constructing an object containing only a string; the data is copied.*
	AFData (const vector< string > &strVect) *Short cut for constructing an object containing a string vector; the data is copied.*
	AFData (const vector< double > &realVect) *Short cut for constructing an object containing a double vector; the data is copied.*
	AFData (const double *realArray, size_t n) *Short cut for constructing an object containing a double array; the data is copied.*
	AFData (const **DimsType** &dims, const vector< double > &realMat) *Short cut for constructing an object containing a double matrix; the data is copied.*
	AFData (const string &str, double x) *Short cut for constructing an object containing a (string,scalar) pair; the data is copied.*
	~AFData () *Destructor.*
const **DimsType** &	**GetDims** () const *Returns the dimension(s) of the object.*
size_t	**GetNumItem** () const

	Returns the number of data items that the object can contain.
AF_DATA_TYPE	**GetType** () const *Returns the data-type of the object.*
const string &	**GetStringAt** (size_t pos) const throw (const char*) *Returns the data element at position pos.*
double	**GetRealAt** (size_t pos) const throw (const char*) *Returns the data element at position pos.*
complex< double >	**GetComplexAt** (size_t pos) const throw (const char*) *Returns the data element at position pos.*
const **AFData** *	**GetAFDataAt** (size_t pos) const throw (const char*) *Returns the data element at position pos.*
bool	**operator==** (const **AFData** &x) const *Equality operator.*
bool	**operator!=** (const **AFData** &x) const *Inequality operator.*
AFData &	**operator=** (const **AFData** &rhs) *Assignment operator.*
	operator double () const throw (const char*) *An easy way to get the value of a scalar double object.*
	operator int () const throw (const char*) *An easy way to get the value of a scalar AF_REAL object, that is cast to int.*
	operator const string & () const throw (const char*) *An easy way to get the value of a string object.*
bool	**IsScalar** () const *Returns true if the object contains a numeric scalar value.*
bool	**IsRealVector** () const *Returns true if the object contains a 1D vector of one or more numeric scalar values.*
bool	**IsEmpty** () const *Returns true if the object is _empty_, and thus can not contain any data.*
void	**ToString** (string &str) const *Generates a textual representation of the data object and its content.*
template<typename T>	
const T &	**GetAt** (size_t pos) const throw (const char*) *Returns the data element at position pos.*
template<typename T>	
const T &	**GetAt** (size_t x, size_t y) const throw (const char*) *Returns the data element at position pos.*
template<typename T>	
const T &	**GetAt** (size_t x, size_t y, size_t z) const throw (const char*) *Returns the data element at position pos.*
template<typename T>	
void	**SetAt** (const T &data, size_t pos) throw (const char*) *Set the data at position pos to data (data is copied).*
template<typename T>	
void	**SetAt** (const T &data, size_t x, size_t y) throw (const char*) *Set the data at position pos to data (data is copied).*
template<typename T>	

void	**SetAt** (const T &data, size_t x, size_t y, size_t z) throw (const char*) *Set the data at position pos to data (data is copied).*
template<typename T>	
const T *	**GetAll** () const throw (const char*) *Get a (const-)pointer to all the elements of this **AFData** object.*
template<typename T>	
void	**SetAll** (const T *dataArray) throw (const char*) *Set (overwrite) all the data elements of this **AFData** object with the dataArray.*
template<typename T>	
void	**SetNext** (const T &data) throw (const char*) *Add the next data element to the object.*
template<typename T>	
void	**CopyAllReal** (vector< T > &v) const throw (const char*) *All the data elements are copied into the vector v, after casting into type T (e.g., for possible double to int conversion).*

15.3 Static Public Member Functions

static void	**ToString** (const **AFData** *data, string &str) *Generates a textual representation of a given data object and its content.*
static const char *	**ToString** (**AF_DATA_TYPE** t) *Returns a textual representation of an AF_DATA_TYPE.*
static string	**ToString** (const **DimsType** &dims) *Generates a textual representation of a 'dimensions' object.*
template<size_t n>	
static **DimsType**	**MakeDims** (const int(&dims)[n]) *A static function to make it easy to create "dims" objects, for constructing multi-dimensional **AFData** objects.*
static **DimsType**	**MakeDims** (size_t n, const int dims[])
static void	**Clear** (**AFDataVect** &afv) *A static function which first deletes (deallocates) each of the **AFData** objects that are pointed to by the AFDataVect elements; then the elements are cleared from the vector.*

15.4 Protected Member Functions

void	**TypeCheck** (**AF_DATA_TYPE** type) const throw (const char*)
void	**RangeTypeCheck** (size_t pos, **AF_DATA_TYPE** type) const throw (const char*)
template<typename T>	
const T &	**GetItemAt** (size_t pos) const throw (const char*)
template<typename T>	
void	**SetItemAt** (const T &x, size_t pos) throw (const char*)

15.5 Detailed Description

AFData is the AlgoFlex data-container class.

The data can be parameters, coefficients, or any other data that is exchanged within AlgoFlex. Most of AFData's methods are concerned with getting the data into and out of the objects.

Some methods may throw an exception, which will be a string containing the function name and an error message. When compiled in Release mode (as opposed to Debug mode), the type- and range-checks performed in AFData's methods are switched off (i.e., compiled into nothing) for performance reasons.

16 The AlgoFlex client/server protocol

[http://wiki/bin/view/TCDesign/AlgoFlexProtocol?skin=plain]

```
MESSAGE = [ COMMAND N_ARGS ARG* ]
COMMAND = Command, null terminated ASCII string (case insensitive)

RETURN_MESSAGE = [ STATUS N_ARGS ARG* ]
STATUS = Return message, null terminated ASCII string (case insensitive),
empty string if no error occurred

N_ARGS = Number of arguments, a double scalar 0,1,...
ARG = [ TYPE DIMENSION DATA ]
TYPE = [ STRING | REAL | COMPLEX ]
DIMENSION = [ DIM_LEN DIM+ ]
DIM_LEN = Numbers of following DIMs, 1,2,...
DIM = Length of dimension 1,2,...
      NB: How is an empty vector represented? Simple by DIM = 0.0 ?
      Examples:
      A scalar corresponds to DIMENSION = [1.0 1.0]
      A 5x3 matrix (5 rows, 3 columns) corresponds to DIMENSION = [2.0 5.0
3.0]

STRING = 1.0
REAL = 2.0
COMPLEX = 3.0
DATA = [ String+ | Real+ | Complex+ ]

String = Null_terminated_ASCII_string
Real = IEEE_754_double (64 bits)
Complex = [ Real Real ]
          First entry is the real part, second entry is the imaginary part
```

TODO: The recursive data-type is in fact implemented in both the client/server-protocol, and in the XML data representation, but is missing here.

17 The AlgoFlex XML Schema

[AlgoFlex\Server\xsd\AlgoFlexXML.xsd]

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified"
           attributeFormDefault="unqualified">

   <xs:annotation>
      <xs:documentation xml:lang="en">
         XML Schema for AlgoFlex block-diagram and parameter documents.
         Revision: 2008-01-23
         Author: Esben Skovenborg
         Copyright 2008 TC Group. All rights reserved.
      </xs:documentation>
   </xs:annotation>


   <!--This XML schema has been built with named types in order to maintain the code
easily,
       and for the names to be used in the code generated by the XSD compiler.-->

   <!--Simple types-->

   <!--NB: The arbitrary-precision numerical types (e.g. xs:positiveInteger) are not
used here.
       Instead we use xs:int (in this case), which maps to 32-bit signed int by the
XSD compiler.-->
   <xs:simpleType name="posInt32">
      <xs:restriction base="xs:int">
         <xs:minInclusive value="1"/>
      </xs:restriction>
   </xs:simpleType>

   <xs:simpleType name="nonNegInt32">
      <xs:restriction base="xs:int">
         <xs:minInclusive value="0"/>
      </xs:restriction>
   </xs:simpleType>

   <!--A valid AlgoFlex-identifier is essentially a valid C/C++ identifier-->
   <xs:simpleType name="algoflexIdentifierType">
      <xs:restriction base="xs:string">
         <xs:pattern value="[a-zA-Z_$][a-zA-Z0-9_$]*"/>
      </xs:restriction>
   </xs:simpleType>

   <xs:simpleType name="blockNameType">
      <xs:restriction base="algoflexIdentifierType"/>
   </xs:simpleType>

   <xs:simpleType name="parmNameType">
      <xs:restriction base="algoflexIdentifierType"/>
   </xs:simpleType>

   <!--The propertyValueType list's itemType is restricted to real-numbers and/or
       strings-without-whitespace implicitly via the propertyTypeType-->
   <xs:simpleType name="propertyItemType">
      <xs:restriction base="xs:normalizedString">
         <xs:pattern value="[^ ]+"/>
      </xs:restriction>
   </xs:simpleType>
   <!--It would not make sense to define a propertyValueItemType as a union, when
       realNumberType is a subset of propertyStringItemType!
       <xs:simpleType name="propertyValueItemType">
          <xs:union memberTypes="propertyStringItemType realNumberType"/>
```

```
      </xs:simpleType>  -->

   <!--The type of a parameter-property is specified explicitly by the
propertyTypeType
       attribute: Either a list of numbers or a list of strings.
       The parameter-property value can be a single item or an ordered list of items.
       When the value is a number it is represented as a string, for simplicity.
       The strings can NOT contain white-space (which is the list-seperator) but
       anything else.-->
   <xs:simpleType name="propertyValueBaseType">
      <xs:list itemType="propertyItemType"/>
   </xs:simpleType>

   <!--This implies that an empty string is NOT a valid property-value.
       Also an empty string wouldn't make sense as an item in a (string-)list.-->
   <xs:simpleType name="propertyValueType">
      <xs:restriction base="propertyValueBaseType">
         <xs:minLength value="1"/>
      </xs:restriction>
   </xs:simpleType>

   <xs:simpleType name="propertyTypeType">
      <xs:restriction base="xs:string">
         <xs:enumeration value="real"/>
         <xs:enumeration value="string"/>
      </xs:restriction>
   </xs:simpleType>

   <!--As a channel is only identified by a number, this type is a restriction to an
integer-->
   <xs:simpleType name="channelType">
      <xs:restriction base="posInt32"/>
   </xs:simpleType>

   <xs:simpleType name="exeRateType">
      <xs:restriction base="xs:double">
         <xs:minExclusive value="0"/>
      </xs:restriction>
   </xs:simpleType>

   <xs:simpleType name="realNumberType">
      <xs:restriction base="xs:double"/>
   </xs:simpleType>

   <xs:simpleType name="stringType">
      <xs:restriction base="xs:string"/>
   </xs:simpleType>


   <!--Complex types-->

   <!--DIAGRAM-->

   <!--This type defines audio connections between two blocks-->
   <xs:complexType name="audioConnectionAttributesType">
      <!--The attributes are required which means that the element would not
          be completely described if they were not specified-->
      <xs:attribute name="srcChannel" type="channelType" use="required"/>
      <xs:attribute name="dstChannel" type="channelType" use="required"/>
   </xs:complexType>

   <!--This type defines parameter connections where the transmitter/receiver will be
an attribute-->
   <xs:complexType name="parameterConnectionAttributesType">
      <xs:attribute name="transmitterParm" type="parmNameType" use="required"/>
      <xs:attribute name="dataParm"        type="parmNameType" use="required"/>
   </xs:complexType>

   <!--This type defines the audio connection type with at least one connection-->
```

```
   <xs:complexType name="audioConnectionType">
      <xs:sequence>
         <xs:element name="audioConn" type="audioConnectionAttributesType"
maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>

   <!--This type defines the parameter connection type with at least one connection-->
   <xs:complexType name="parameterConnectionType">
      <xs:sequence>
         <xs:element name="parameterConn" type="parameterConnectionAttributesType"
maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>

   <!--This type defines the connections between two blocks: parameter- or audio-
connections-->
   <xs:complexType name="connectionType">
      <!--Either audio connections OR parameter connections - not both at the same
time-->
      <xs:choice>
         <xs:element name="audioConnections" type="audioConnectionType"/>
         <xs:element name="parameterConnections" type="parameterConnectionType"/>
      </xs:choice>
      <!--Source block-instance name-->
      <xs:attribute name="srcBlock" type="blockNameType" use="required"/>
      <!--Destination block-instance name-->
      <xs:attribute name="dstBlock" type="blockNameType" use="required"/>
   </xs:complexType>

   <!--This complex type defines the attrbutes of the IODimension tag-->
   <xs:complexType name="IODimensionType">
      <xs:attribute name="nIn"  type="nonNegInt32" use="required"/>
      <xs:attribute name="nOut" type="nonNegInt32" use="required"/>
   </xs:complexType>

   <!--This type defines the information that a block of the diagram contains-->
   <xs:complexType name="blockInstanceType">
      <xs:sequence>
         <!--Name of the block (unique)-->
         <xs:element name="blockName" type="blockNameType"/>
         <!--Library name (block type)-->
         <xs:element name="libName" type="blockNameType"/>
         <!--Block nIn and nOut channels-->
         <xs:element name="IODimension" type="IODimensionType"/>
         <!--Block's execution rate-->
         <xs:element name="executionRate" type="exeRateType"/>
         <!--Connections of the block (only those which start from this block) with at
least one connection-->
         <xs:element name="connection" type="connectionType" minOccurs="0"
maxOccurs="unbounded"/>
      </xs:sequence>
      <!--blockID has been omitted as an atribute-->
   </xs:complexType>

   <!--The partial execution sequence - 'partial' simply means that the sequence only
contains the blocks
       included in this document.-->
   <xs:complexType name="partialExecutionSequenceType">
      <xs:sequence>
         <xs:element name="bl" type="blockNameType" maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>

   <!--PARAMETERS-->

   <xs:complexType name="complexNumberType">
      <xs:attribute name="re" type="realNumberType" use="required"/>
      <xs:attribute name="im" type="realNumberType" use="required"/>
```

```
   </xs:complexType>

   <!--This type defines a vector of (0 or more) real numbers-->
   <xs:complexType name="vectorRealType">
      <xs:sequence>
         <xs:element name="va" type="realNumberType" minOccurs="0"
maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>

   <!--This type defines a vector of (0 or more) complex numbers-->
   <xs:complexType name="vectorComplexType">
      <xs:sequence>
         <xs:element name="va" type="complexNumberType" minOccurs="0"
maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>

   <!--This type defines a vector of (0 or more) strings-->
   <xs:complexType name="vectorStringType">
      <xs:sequence>
         <xs:element name="va" type="stringType" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>

   <!--This type defines a vector of (0 or more) n-dimensional data objects-->
   <xs:complexType name="vectorMatrixDataType">
      <xs:sequence>
         <xs:element name="va" type="matrixDataType" minOccurs="0"
maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>

   <!--Dim is the dimension, which is a vector in order to represent n-dimensional
matrices-->
   <xs:complexType name="dimType">
      <xs:sequence>
         <xs:element name="dim" type="nonNegInt32" maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>

   <!-- Could dimType be an xs:list type instead?
        Well, more compact but perhaps less natural and also less support from some
tools.
    <xs:simpleType name='dimListBaseType'>
        <xs:list itemType='nonNegInt32'/>
    </xs:simpleType>
    <xs:simpleType name='dimListType'>
        <xs:restriction base='dimListBaseType'>
            <xs:minLength value='1'/>
        </xs:restriction>
    </xs:simpleType>
    -->

   <!--This type defines n-dimensional data of a certain type-->
   <xs:complexType name="matrixDataType">
      <xs:sequence>
         <xs:element name="dimension"  type="dimType"/>
         <xs:choice>
            <xs:element name="real"    type="vectorRealType"/>
            <xs:element name="complex" type="vectorComplexType"/>
            <xs:element name="string"  type="vectorStringType"/>
            <!--Recursive data type-->
            <xs:element name="data"    type="vectorMatrixDataType"/>
         </xs:choice>
      </xs:sequence>
   </xs:complexType>

   <!--This type defines the attributes of the property tag-->
```

```
   <xs:complexType name="parmPropertyType">
      <xs:attribute name="name"  type="parmNameType"      use="required"/>
      <xs:attribute name="type"  type="propertyTypeType"  use="required"/>
      <xs:attribute name="value" type="propertyValueType" use="required"/>
   </xs:complexType>

   <!--This type defines a parameter block-->
   <xs:complexType name="parameterType">
      <xs:sequence>
         <xs:element name="parmName" type="parmNameType"/>
         <xs:choice>
            <!--We allow real and string scalars without explicit dimension-->
            <xs:element name="real" type="realNumberType"/>
            <xs:element name="string" type="stringType"/>
            <!--The data element will represent all other cases, incl. vector and
matrix-->
            <xs:element name="data" type="matrixDataType"/>
         </xs:choice>
         <!--Property is an element that describes limitations or enhancements for a
parameter;
             there can be 0 or more properties per parameter-->
         <xs:element name="property" type="parmPropertyType" minOccurs="0"
maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>

   <!--The type of the blockParameters elements (the individual blocks)-->
   <xs:complexType name="blockParametersType">
      <xs:sequence>
         <xs:element name="blockName" type="blockNameType"/>
         <!--At least one parameter per block-->
         <xs:element name="parameter" type="parameterType" maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>


   <!--
================================================================================-->

   <!--The blocks of the diagram (or blocks from a subset of the diagram)-->
   <xs:complexType name="diagramDocType">
      <xs:sequence>
         <xs:element name="partialExecutionSequence"
type="partialExecutionSequenceType"/>
         <!--One or more blocks-->
         <xs:element name="blockInstance" type="blockInstanceType"
maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>

   <!--One or more parameter values for one or more blocks-->
   <xs:complexType name="parametersDocType">
      <xs:sequence>
         <!--One or more blocks-->
         <xs:element name="blockParameters" type="blockParametersType"
maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>

   <!--The document type: either parameters or diagram-->
   <xs:complexType name="algoflexDocType">
      <xs:sequence>
         <!--A multi-line description (optional)-->
         <xs:element name="description" type="xs:string" minOccurs="0"/>
         <!--A diagram-document OR a parameters-document-->
         <xs:choice>
            <xs:element name="diagramDoc" type="diagramDocType"/>
            <xs:element name="parametersDoc" type="parametersDocType"/>
         </xs:choice>
```

```
      </xs:sequence>

      <!--The dateTime type is a date/time combination in ISO 8601 format (CCYY-MM-
DDTHH:MM:SS-Timezone)-->
      <xs:attribute name="created" type="xs:dateTime" use="required"/>
      <!--The AlgoFlex server version (that generated this document)-->
      <xs:attribute name="serverVersion" type="xs:normalizedString" use="required"/>
      <!--An author tag-->
      <xs:attribute name="author"  type="xs:normalizedString" use="required"/>
      <!--A document name tag (optional), a string suitable as a block name-->
      <xs:attribute name="name"    type="algoflexIdentifierType" use="optional"/>
   </xs:complexType>

   <!--
=================================================================================-->

   <!--The root of the XML document-->
   <xs:element name="AlgoFlexXmlDoc" type="algoflexDocType"/>

</xs:schema>
```