

SynStart

Version 1.1

Beat Frei

Institute for Computer Music and Sound Technology (ICST)

Zurich University of the Arts

Baslerstrasse 30, CH-8048 Zurich, Switzerland

beat.frei@zhdk.ch, <http://www.icst.net>

Download the recent version of SynStart and the Digital Sound Generation book from

<http://www.icst.net/dsgdownload>

What is SynStart?

SynStart is a free open source framework for real-time audio application programming in C++. The current version includes all the interfaces and administrative routines required to build stand-alone sound generators and audio processors for the Windows platform: A polyphonic multitimbral voice manager, ASIO and DirectX audio, MME and DirectX MIDI, and a basic graphical user interface. It does not provide predefined audio algorithms however. Instead, the developer retains full control over signal processing and can concentrate his design effort on the core of the instrument. There's example code included that shows how to create a MIDI-controlled polyphonic sine wave synthesizer with 18 code lines.

Applications

Designed from scratch with focus on efficiency and low latency, Synstart not only makes a practical research tool but an initial point for professional projects as well. The recent version 1.1 is released under the zlib license (see last page), which is liberal and OSI-approved.

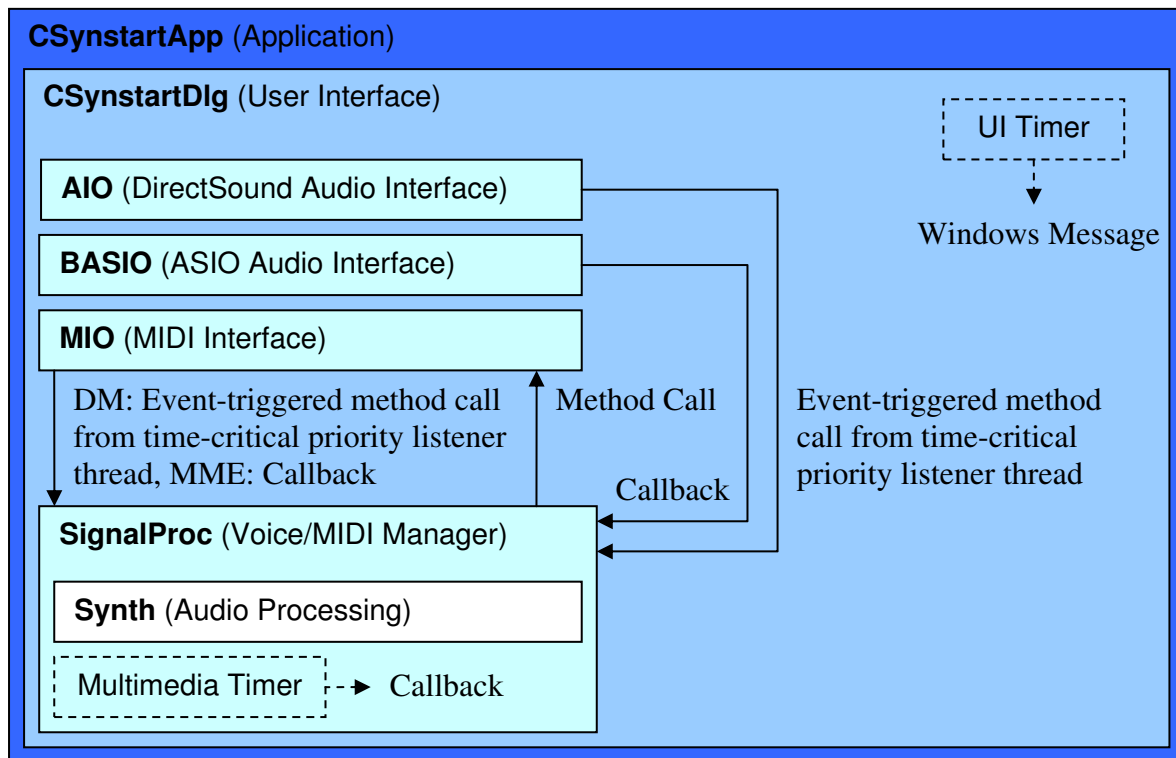
Get it running

1. *Optional (if you want MIDI)*: Attach a MIDI interface to your computer or get a virtual MIDI cable (e.g. www.hurchalla.com/Maple_driver.html or <http://www.midiox.com>).
2. *Optional (if you want ASIO)*: Attach an ASIO audio interface to your computer or get an ASIO driver for your soundcard (e.g. www.asio4all.com).
3. Run *Syntest.exe* to test the interfaces. The program uses the standard audio and MIDI devices selected in the sound settings of Windows. They should appear by name and active. The audio output consists of a constant tone and the fed through input whose pitch and level can be varied with sliders. Play the sine wave synthesizer on MIDI channel 1. A flashing star to the right of the port name indicates incoming MIDI data. Tick the checkbox to toggle between DirectSound and ASIO. If everything works so far, you are ready to set up the SynStart environment.
4. Download and install the appropriate DirectX9 SDK (see page 9, DirectX10 requires Windows Vista): www.msdn.microsoft.com/directx/sdk. The default installation should add the following paths to your MSVC environment (VC6, SDK Summer 03):
 - a. Extras > Options > Directories > Include: `C:\DX90SDK\Include`
 - b. Extras > Options > Directories > Libraries: `C:\DX90SDK\Lib`

In some cases, you might have to enter them manually at the top of the lists.

5. Download the ASIO SDK: www.steinberg.de > Company > 3rd Party Developer. Copy the following files to the SynStart project folder: *asio.cpp/h*, *asiodrivers.cpp/h*, *asiolist.cpp/h*, *asiosys.h*, *ginclude.h*, *iasiodrv.h*. Read and follow the license terms of Steinberg. For non-commercial use this mainly results in including a copyright notice in the *About* box of your program (click the title bar of *Syntest.exe* to see it).
6. Open the test project *Synstart.dsw* and build it. Microsoft Visual C++ pragmas are used. In other environments, you may have to link the following libraries manually: *winmm.lib*, *dsound.lib*, *dxguid.lib*.
7. Run the test project. It should behave like *Syntest.exe*.

Structure



Each class owns only instances of the next lower layer classes and communicates with them top-down via method calls. Arrows indicate additional intermediate communication paths. You may find it quite easy to replace a class or rip it for use in another project.

The Synth Class

Audio processing happens here. **All of the following methods are called automatically by the framework. Along with associated voice variables, they are usually the only thing you have to modify apart from the user interface to build individual sound generators.** Insert your own code to perform the desired actions. Refer to appendices A1 and A2 to learn how a generic sound generator maps to the SynStart framework.

Method	Description
UpdateSynAudio (float *audioout, float *audioin, int outchs, int inchs, int samples, float smprate, float smpinv)	Called when the next audio frame has to be calculated. Insert code to generate new audio data and fast control signals. Parameters: audioout = pointer to the new audio output data, audioin = pointer to the recent audio input data, outchs = number of output channels, inchs = number of input channels, samples = number of samples per frame (every audio channel requires/holds this many values), smprate = sample rate in Hz smpinv = 1/smprate = sampling interval in s
UpdateSynControl (int samples, float smprate, float smpinv)	Called by UpdateSynAudio. Insert code to generate control signals at a lower rate. Parameters: As in UpdateSynAudio.

Reset()	Insert code to set the sound generator to a default state (e.g. zero envelopes).
NoteOn(int vnr, int key, int vel, bool legato)	Insert code to play a new note on a specific voice (e.g. update oscillator frequencies, trigger envelopes). Parameters: vnr = voice number, key = MIDI key number (min = 0, max = 127), vel = velocity (min = 1, max = 127), legato = true if the note is played legato.
NoteOff(int vnr)	Insert code to end a note on a specific voice (e.g. enter release phase of envelopes). Parameter: vnr = voice number
MidiClockEvent()	A MIDI clock event occurred.
ConvParameter(int nr)	Called by SetParameter. Convert a parameter to a more practical representation if desired. Parameter: nr = parameter index.

There are additional methods not intended for change:

Method	Description
Init(int polyphony)	Initialize the audio processor. Parameter: polyphony = maximum number of simultaneously playing voices (0 is also valid, e.g. for effectors).
ResetCont()	Set controls to a default state.
SetControlValue(int nr, float val)	A MIDI control change occurred. Updates the corresponding value of the <i>cval</i> array. Parameters: nr = controller number, val = value normalized to 1.
SetKeypress(int vnr, float val)	A MIDI key pressure change occurred. Updates the corresponding keypress value of the voice. Parameters: vnr = voice number, val = value normalized to 1.
SetParameter(int nr, float val)	An audio processing parameter changed. Updates the corresponding value of the <i>param</i> array. Parameters: nr = parameter index, val = value normalized to 1.
GetParameter(int nr)	Read parameter value. Parameters: nr = parameter index.

Some general remarks:

- **All audio data, MIDI controller values, and synthesis parameters are represented as single precision floating point numbers between -1 and 1.** Example: A slider of the user interface ranges from -20 to 100. The associated parameter will then span from -0.2 to 1.0. Occasionally, conversion is desirable, e.g. to form an integer index for a table or a controller number. In this case, add some code to the *ConvParameter* method and store the result in a common variable. Overwriting the parameter itself is discouraged because *GetParameter* would require support for exact back-conversion.
- **Audio buffers audioout/audioin are one-dimensional arrays:**

Channel 0 Sample 0	Channel 0 Sample (samples-1)	Channel 1 Sample 0	Channel (outchs/inchs-1) Sample (samples-1)
-----------------------	---------------------------------	-----------------------	--

- Taking *smprate* and *smpinv* into account allows your sound processor to work well with different sample rates (e.g. 44.1/48/88.2/96 kHz).
- The *SignalProc* class zeroes the audio output buffer before it calls *UpdateSynAudio*. It also applies a brickwall limiter to the data returned by *UpdateSynAudio*.

A quick-and-dirty polyphonic sine wave synthesizer shows what it takes to get started.

```

void Synth::UpdateSynAudio( float *audioout, float *audioin, int outchs, int inchs, int samples, float smprate,
                           float smpinv)
{
    // *** begin example code ***
    int i,j; float phi,amp,pitch,level; static const float twopi = 6.283185f;
    for (j=0; j<poly; j++) // for all voices
    {
        pitch = v[j].pitch*smpinv*(param[0] + 1.0f); // pitch control by slider 1 incl.
        phi = v[j].phi; amp = v[j].amp; level = v[j].level; // sample rate compensation
        for (i=0; i<samples; i++) // for all samples
        { // create sine wave inefficiently
            audioout[i] += (amp*(float)sin(phi));
            phi += pitch; if (phi > twopi) {phi -= twopi;}
            amp = 0.99f*amp + 0.01f*level; // update amplitude smoothly
        }
        v[j].phi = phi; v[j].amp = amp;
    }
    // *** end example code ***
    UpdateSynControl(samples,smprate,smpinv);
}

// play a new note
void Synth::NoteOn(int vnr, int key, int vel, bool legato)
{
    if ((vnr < 0) || (vnr >= poly)) {return;}
    // *** begin example code ***
    // convert MIDI key and velocity to frequency and amplitude
    static const float scl = (float)(log(2.0)/12.0), cntr = 51.3657f;
    v[vnr].pitch = cntr*(float)exp(scl*(double)key);
    v[vnr].level = 0.001f*(float)vel;
    // *** end example code ***
}

// a playing note has been released
void Synth::NoteOff(int vnr)
{
    if ((vnr < 0) || (vnr >= poly)) {return;}
    // *** begin example code ***
    v[vnr].level = ZERO_AD; // near zero value (1e-20) to
    // *** end example code *** // avoid denormal numbers
}

// set synth to startup state
void Synth::Reset()
{
    ResetCont();
    // *** begin example code ***
    for (int i=0; i<poly; i++) {v[i].pitch = v[i].amp = v[i].phi = 0.0f; v[i].level = ZERO_AD;}
    // *** end example code ***
}

class Voice
{
public:
    float keypress; // key pressure control variable
    // *** begin example code *** // add your own voice variables
    float pitch; // oscillator
    float phi;
    float amp; // amplifier
    float level;
    // *** end example code ***
};

```

Essential variables of the *Synth* class are:

- cvar[]:** An array that contains the current MIDI controller values. The index equals the controller number and ranges from 0 to 119. Additionally, the pitch wheel is mapped to *cvar*[120] and channel pressure to *cvar*[121]. Values are updated automatically by the framework via *SetControlValue*. Some standard controller numbers are already defined by name in *synth.h*.
- param[]:** An array that contains the sound synthesis parameters. The framework updates them via *SetParameter* whenever a control element of the user interface has been moved or a new sound program is being loaded.
- poly:** Polyphony. It indicates the number of voices that have to be processed. The framework sets the value when a synthesizer is initialized.
- v[]:** A variable length array of **voices**.

A **voice** is a data set that contains all the variables associated with the generation of a single note. Since many notes may sound simultaneously, there's a plurality of voices, as opposed to *cvar* and *param*, which are common to all voices. Voice variables typically serve one of the following purposes:

- They store a voice-specific state for use with synthesis algorithms, e.g. the phase accumulator of an oscillator or an internal counter of an envelope generator.
- They hold a voice-specific control value that has been calculated from common parameters as found in *param* and *cvar* and voice-specific control data like note number and velocity.

In the example code on page 5, we see how they are used efficiently in *UpdateSynAudio*:

1. Process a single voice as follows:
2. Copy all voice variables to local variables for faster access.
3. Calculate new audio data (typ. 64 to 256 samples) of the voice using local variables. State variables change each time a new sample is calculated whereas control variables remain constant.
4. Copy the updated local state variables back to the voice variables. This ensures that the current state can be recalled when *UpdateSynAudio* is invoked next time. Local control variables are unchanged, hence they needn't be copied.
5. Proceed with the next voice.

Last but not least, the *Synth* class includes some constants you may like to change:

- P_MAX** = Maximum parameter index, range: [0..P_MAX]
- ACRR** = Audio to control rate ratio. A larger value means more but slower controls like envelope generators or LFOs for a given amount of processing time. Analyze *UpdateSynControl* to see how it works.

The SynstartDlg Class

This class encapsulates the user interface. Initialization is also done here. With respect to the latter, some code segments are especially interesting:

```
BOOL CSynstartDlg::OnInitDialog()
```

```
:
```

```
int inport = 0, outport = 0; // select MIDI in + out ports by index
```

To select a specific port by name, iterate *Init* with *strict* = true counting up from 0. Obtain the port names with *GetDeviceInfo*. If the index exceeds the port range, *E_INDEX* is returned. In case you don't need a certain port, call *Init* with the index set to -1 to disable it. If you disable both ports, the interface doesn't use any resources at all. Important return values of *Init* are: 0 = requested ports assigned, *E_NOOUT* = no ports assigned (if output assignment fails, no attempt is made to assign an input), other non-zero value = output but no input port assigned.

```
int CSynstartDlg::SetupAudioInterface(int type)
```

```
:
```

```
else {srate = 48000; inch = outch = 2; buf = 128;} // DirectSound settings
```

```
:
```

```
else {srate = 48000; inch = 2; outch = 2; buf = 0; drv = 0;} // ASIO settings
```

Set the sample rate *srate* and the number of audio channels *inch/outch*. Direct Sound: 1 (mono) or 2 (stereo) full-duplex I/O pairs. ASIO: Independent number of inputs and outputs ranging from 0 to any desired value. In the current version of SynStart, channel numbers are assigned consecutively from 0 up.

Set the buffer size *buf*, which is equal to the number of samples per channel processed at once. A larger value results in higher latency, a smaller value increases the processor load and may cause dropouts by overextending the soundcard driver. DirectSound adds about 20 ms of latency to the output chain due to the always enabled K Mixer of Windows. Expect typical input-to-output delays of 35 ms for DirectSound and 5 to 10 ms for ASIO (*buf* = 64..128, *srate* = 48000..96000). ASIO only: *buf* = 0 makes the driver choose its preferred buffer size.

On some systems, multiple ASIO drivers are present. Select one by the index *drv* or iterate *Init* with *strict* and *test* = true incrementing *drv* from 0 in combination with *GetDriverName* to choose a specific one by name. If the index exceeds the driver range, *Init* returns *E_INDEX*.

Assign and stack up to 16 synthesizers with 0 to 128 voices each to any MIDI channel:

```
sbc->SetRange(0,0); // change to number of synths – 1
```

```
// iterate for the desired number of synths
```

```
NofParams = Dsp->CreateSynth(
```

```
    0,
```

```
    // create synth 0
```

```
    1,
```

```
    // synth number
```

```
    16,
```

```
    // MIDI channel
```

```
    false);
```

```
    // polyphony
```

```
    // true: inverted hold pedal
```

```
if (NofParams >= 0) {
```

```
    synthnr = 0;
```

```
    // link controls to synth 0
```

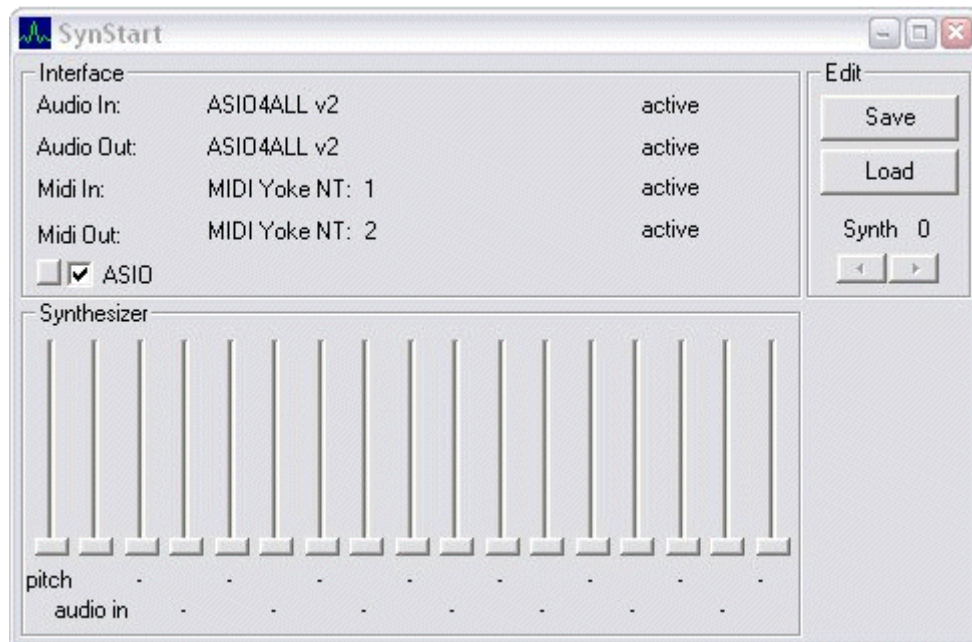
```
    InitSynthControls();
```

```
    // init controls and synth parameters
```

```
    Dsp->EnableSynth(synthnr); }
```

```
    // enable synth
```

The user interface provides basic functionality without intentions to win a design award. By default, a slider has a resolution of 100 steps and is mapped linearly to a parameter *param[n]* of the selected synthesizer (see *Synth* class) in the following way: Top position = 1, bottom position = 0, parameter number *n* of the leftmost slider = 0, incremented by one to the right. *Save/Load* act on the parameter values of the selected synthesizer, not on the slider positions themselves. Click the button beneath the ASIO checkbox to open the ASIO control panel. Click the title bar to open the *About* box.



Change a slider's numerical properties in *InitSynthControls* and the assignment to a parameter number in *IdToPara* and *SetSynthControls*. The maximum absolute value of the slider range is always mapped to a parameter absolute value of 1. The increment of the slider position is 1. Example: You want 6 positions to cover the range from -0.25 to 1.0. The initial position has to be 0.5. Set max = 4, min = -1, val = 2.

To add a new control, follow this procedure:

1. Create the control in the resource editor and give its ID a descriptive name.
2. Go to *InitSynthControls* and initialize it. This step also determines the initial value of the associated parameter.
3. Go to *IdToPara* and map the control ID to a parameter number.
4. Go to *SetSynthControls* to add the inverse mapping.
5. If the control is not a slider, add an appropriate handler. Due to the strange messaging of certain MFC controls, this step may be tricky. See *OnVScroll* for converting the control value into a parameter value. Bear in mind that the latter should be normalized to a maximum absolute value of 1 for consistency.

Profile

License: zlib
 Language: C++, incl. MFC and DirectX support
 Development Platform: Windows XP (tested with MSVC 6.0)
 Required Add-Ons (free): DirectX9 SDK (MSVC version must support SDK version, e.g. VC6 with SDK Summer 2003)
 ASIO 2.2 SDK
 Target Platform: Windows XP
 Windows Vista (not tested)
 Windows 2000/98 (ASIO audio + MME MIDI only)
 Running Mode: Standalone Executable
 Size of Executable: ≈ 64 kB
 Size of Download: ≈ 344 kB
 Source Code Lines: ≈ 3000 (excl. ASIO SDK sources)
 Audio I/O: ASIO 2, DirectSound 8
 MIDI I/O: DirectMusic (in, SysEx ignored), MME (in + out, incl. SysEx)

Files	Class	Description
Folders <i>asiosdkcpp/h</i> : asio.cpp/h, asiolist.cpp/h, asiodrivers.cpp/h, asiosys.h, ginclude.h, iasiodrv.h		Required files from the Steinberg ASIO SDK.
aio.cpp/h	AIO	DirectSound audio interface.
Synstart.cpp/h	CSynstartApp	Main application.
SynstartDlg.cpp/h	CSynstartDlg	User interface and initialization. Manage your own GUI here!
basio.cpp/h	BASIO	ASIO audio interface.
mio.cpp/h	MIO	MIDI interface.
Resource.h		GUI resource data. Generated automatically. Don't edit.
SignalProc.cpp/h	SignalProc	Voice/MIDI manager.
StdAfx.cpp/h		Standard system includes. Don't edit.
synth.cpp/h	Synth	Create your own synthesizer here!

Libraries to include: winmm.lib, dsound.lib, dxguid.lib

Project Settings in MSVC 6.0, Release (*Debug*) version:

General: MFC Shared DLL
 Runtime Library: Multithreaded DLL (*Debug Multithreaded DLL*)
 Precompiled Headers: use automatically by the header stdafx.h
 Preprocessor Definitions: WIN32,NDEBUG(_*DEBUG*),_WINDOWS,_AFXDLL,_MBCS
 Project Options: /nologo /MD /W3 /GX /O2 /D "WIN32" /D "NDEBUG" /D
 "_WINDOWS" /D "_AFXDLL" /Fp"Release/Synstart.pch"
 /YX"stdafx.h"
 (*Project Options*): /nologo /MDd /W3 /Gm /GX /ZI /Od /D "WIN32" /D "_*DEBUG*"
 /D "_WINDOWS" /D "_AFXDLL" /Fp"Debug/Synstart.pch"
 /YX"stdafx.h" /Fo"Debug/" /Fd"Debug/" /FD /GZ /c

Troubleshooting

Conditions: An ASIO device is configured as standard audio output of Windows and a DirectSound application is currently using it. A common case is the MS wavetable synth which has been launched unintended by opening the default (and often only) MIDI out port of Windows. Now, a second application, e.g. SynStart, tries to share the audio output.

Problem: May work or exhibit various malfunctions from silence to a deadlocked audio system.

Solution: Avoid ASIO devices for system audio. Initialize MIDI output in SynStart with the *nosynth* flag set.

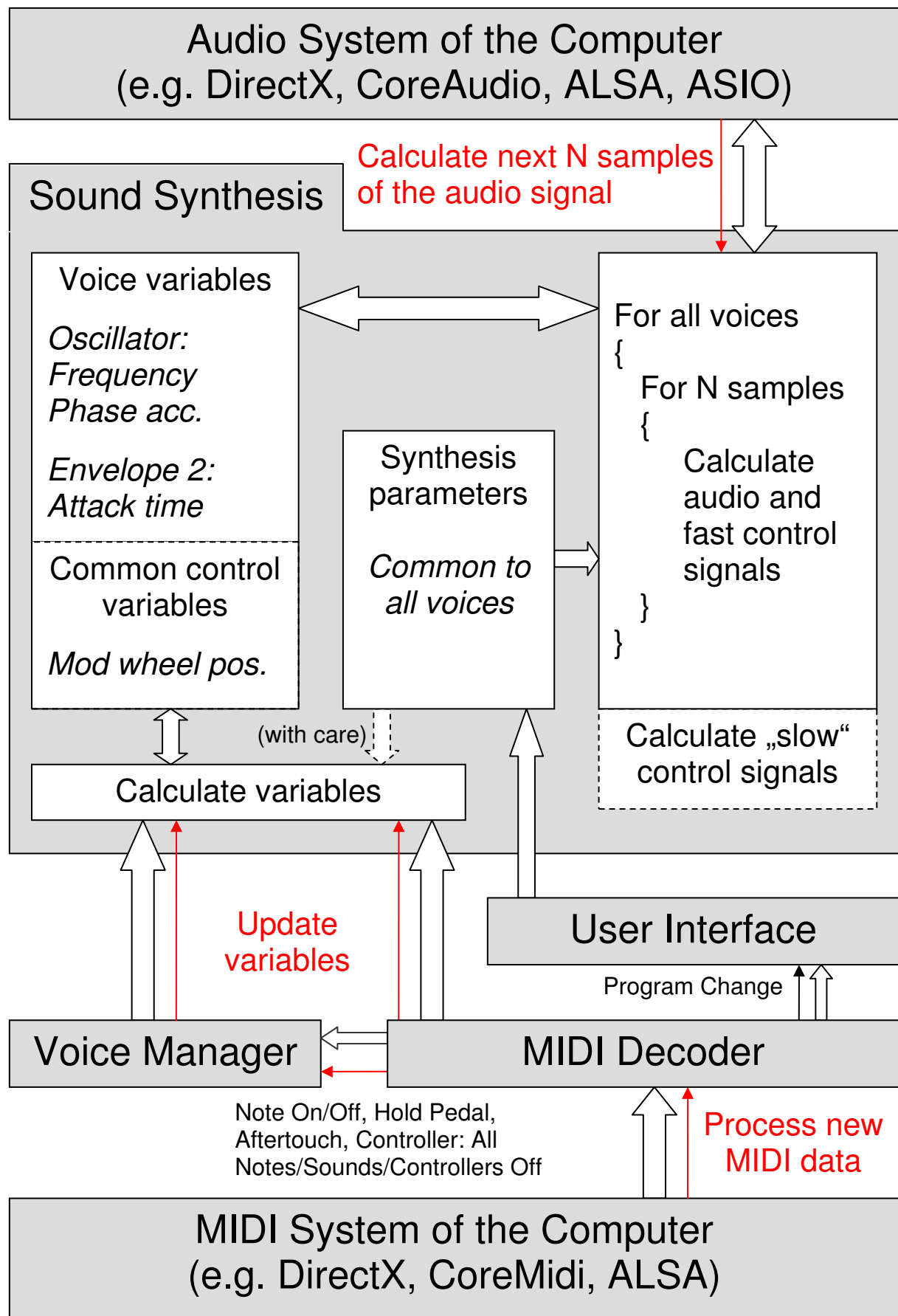
Conditions: DirectSound audio. The computer is a notebook with CPU power management technology or uses an external interface to play system audio or runs a lot of system tray tasks.

Problem: Dropouts occur with the default settings of SynStart at low CPU load.

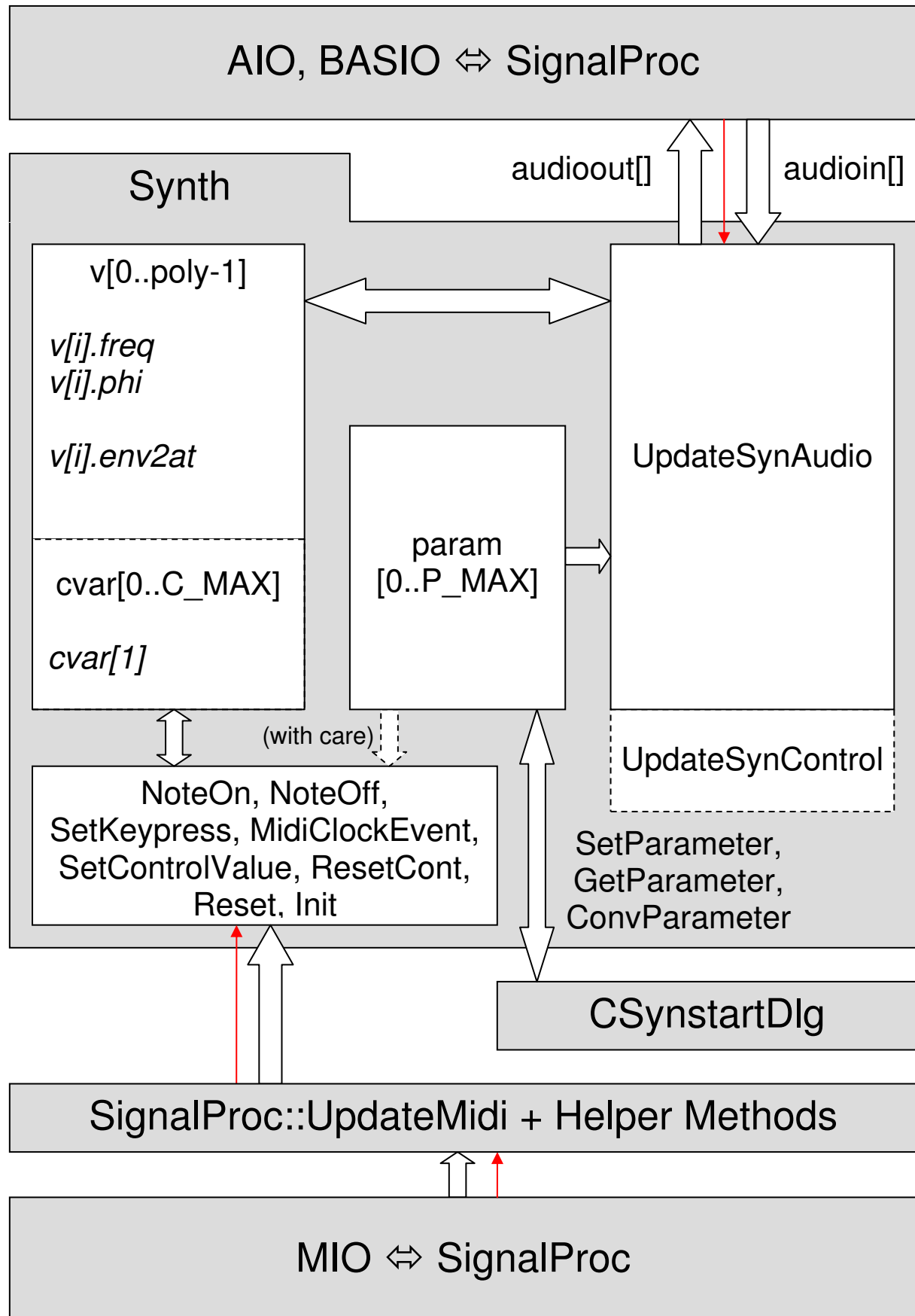
Solution: Change CPU power management (battery manager, SpeedStep, PowerNow) from automatic to maximum performance even if you run the notebook from the adapter. Configure the computer for professional audio. This is especially important if you also have to set large buffer sizes with non-system ASIO. The DirectSound interface of SynStart may be initialized with the flag set to *conservative* accepting increased latency. Try ASIO4ALL in stubborn cases.

Background: All of the above conditions may add significant constant or sporadic delay to audio handling. The DirectSound interface of SynStart has proven to work reliably with the setting *fast* at CPU loads around 80% on average desktop and notebook PCs with standard on-board audio. Under these conditions, setting it to *normal* (default) or *conservative* usually allows a CPU load beyond 90% up to the point where the system starts to exhibit noticeably longer reaction times of the graphical user interface.

Appendix A1: Generic Sound Generator Structure



Appendix A2: How the Generic Structure Maps to SynStart



Appendix B: ZLib License (<http://opensource.org/licenses/zlib-license.html>)

*** License text ***

Copyright (C) 2008 Beat Frei

This software is provided 'as-is', without any express or implied warranty. In no event will the author be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

*** End of license text ***

The following practice is recommended to meet the conditions of the license.

Case 1: You want to distribute the original software as an executable freely or commercially.

Just do it.

You may license it under your own conditions.

Case 2: You want to distribute the original software as a source freely or commercially.

Just do it under the zlib license.

Case 3: You altered the software or use parts of it in your own code and want to distribute it as an executable freely or commercially.

Rename the software.

Use your own copyright in the “About” box.

License it under your own conditions.

Case 4: You altered the software or use some parts of it in your own code and want to distribute it as a source freely or commercially.

Rename the software.

Use your own copyright in the “About” box of the executable.

Add an entry to the history of any source file you altered that states that the original code has been modified.

If you modify the descriptive header of a source file, make sure that the name of the original software is mentioned in the history.

If you rename a source file or use parts of it in other files, include the original file name and the name of the original software in the history.

Include the zlib license text in your source distribution.