APE

Audio Programming Environment

User manual for APE v0.5.0

Lightbridge Studios

Table of contents

INTRODUCTION 3

Features 3

WIP features 4

REQUIREMENTS 4

Minimum 4
Recommended 4
Tested & shipped versions 4
Experimental support 4

USER INTERFACE 5

Splash 5

Editor tab 6

Disk synchronization & autosave 6
Menus 7
Scope probe 7

Oscilloscope tab & settings 8

Console tab 8

Plugin tab 9

Parameter controls 9

CONFIGURATION 10

Application settings 10

Editor settings 11

Language settings 12

CPPAPE: USER SCRIPTING & API DOCUMENTATION 13

Plugin anatomy: Getting started 13

Methods for basic events and callbacks 14
Details on lifetime, locals, exceptions 14
Nesting plugins 14

Plugin SDK 15

C++ standard library 15
SIMD and template precision language extensions 16

STANDALONE WORKFLOW 16

DEVELOPMENT SETUP & SOURCE CODE 17

Prerequisites 17

Repository folder layout 17

Plugin folder layout 18

CppAPE compiler folder layout 18

Building APE 18

COMPILER API EXTENSIONS 19

MIGRATION FROM PREVIOUS COMPILERS AND LANGUAGES 20

Tcc4APE 20

syswrap 20

RELEASES AND CHANGELOG 20

Alpha 0.5.0 21

Alpha 0.3.0 23

Alpha 0.2 23

Alpha 0.1 23

LICENSES 23

CREDITS AND THANKS 24

DOWNLOADS, VIDEOS, CONTACT, LINKS 24

ENDNOTES 25

Introduction

Welcome to the manual for the APE program, Audio Programming Environment. APE is my biggest procrastination yet, having struggled to learn to write DSP code in a practical manner. I grew tired of spending most of my time writing interfaces, frameworks, GUI's and whatnot, managing different libraries, dependencies etc. for very small plugins until I ultimately decided that it would be better to abstract the whole package away - enter APE.

APE is a comparatively small program that hosts user-written code and integrates it into the signal path of your host program. It provides a set of commonly used math and DSP tools, and a generally idiomatic plugin declaration syntax - that maps to a final plugin SDK well - to hit the ground running. Integrated, automatable real time parameters with automatic UI to adjust variables inside your code, and access to certain system information like tempo, channels etc. together with a console and an editor allows easy development and testing.

APE's intended use is the development process - testing out and fine-tuning algorithms in an efficient manner before integrating it into your primary project. It allows for on-the-fly compilation and testing; hence iteration speed is fundamental and very optimized.

You don't have to worry about anything but the actual relevant code, since it's intended to provide a simple introduction to writing DSP code. It is therefore an ideal tool for teaching/educational projects and demonstrating small algorithms and / or effects.

I hope that someday it will evolve into a user provided library of effects with a working implementation, with a design that allows to easily use the code in your own projects. Many great sites on the internet provide a lot of great examples, though they are mostly scattered, incomplete and with a difficult path to testing.

If you are more of a visual type, see the following videos:

- Rundown of features: https://youtu.be/srJ6EWiXOTo
- Developing an effect: https://youtu.be/7_jtCDy39FU

Features

- Completely self-contained - ships with SDKs, compiler, linker, editor, console and runtime
- Bleeding edge Clang/LLVM C++ compiler for class leading diagnostics and code optimization
- Full C++17 support (partial C++20 support)
- Most of the C++ standard library included (containers, math, numeric, algorithmic etc.)
- I/O Audio file streaming with a variety of codecs
- System-level exception handling to avoid crashing your host on small errors like integral division
- Console with logging
- Themeable editor with syntax highlighting
- Oscilloscope with expression evaluation
- Precise, smoothed automatable parameters
- Built-in optimized FFTs
- 26 included effect scripts that are documented (54 legacy scripts for inspiration, as well)
- Extensible backend with ability to interface to any compiler installed on the computer - write in any language you want!
- Cross platform support (AU/VST Windows, OS X, Linux for anyone who wants to compile it)
- Extensive configuration file

WIP features

- Support for compilation of programs to self-contained plugins
- Support for presets
- Actual AST expression evaluator for oscilloscope, not just a define hack
- Integrated projects and tabbed editor
- Explodable GUI and better graphics (for controls, as well), less taxing
- More languages supported, potentially Python
- Stable releases for Linux variants
- Instrument / MIDI support
- Standalone workflow support
- Release / debug switchable code generation (current: optimized debug)
- User-toggleable oversampling factor
- Scrollable, selectable console
- FPU exceptions and control flow

Requirements

Minimum

- Processor: 1 GHz, SSE 4.2
- RAM: 4 gigabytes
- Disk: 200 megabytes free
- Graphics: OpenGL 2 compatible

Recommended

- Processor: 3 GHz, AVX2, 4 cores+
- RAM: 8 gigabytes
- Disk: SSD, 1 gigabyte free
- Graphics: Discrete, 2 gigabyte RAM, OpenGL 3.0

Tested & shipped versions

- Audio Unit 2 & VST 2.4: MacOS 10.11 - 10.14
- VST 2.4: Windows 7 - 10

Experimental support

(These require building the plugin from source)

- Audio Unit 2 - 3, VST 2.4 - 3.6, AAX: MacOS 10.8 - 10.15
- VST 2.4 - 3.6, AAX: Windows XP - Windows 10
- VST 2.4 - 3.6: Debian, Ubuntu

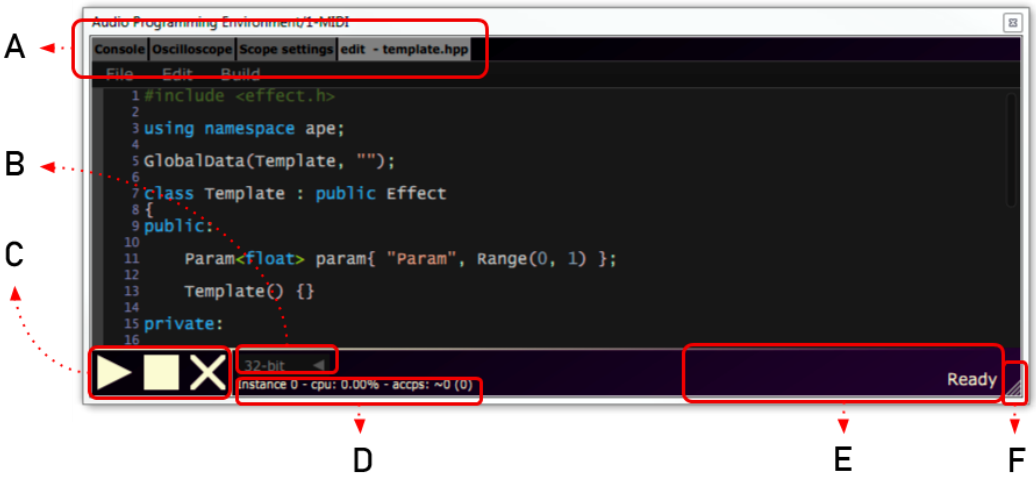
User interface

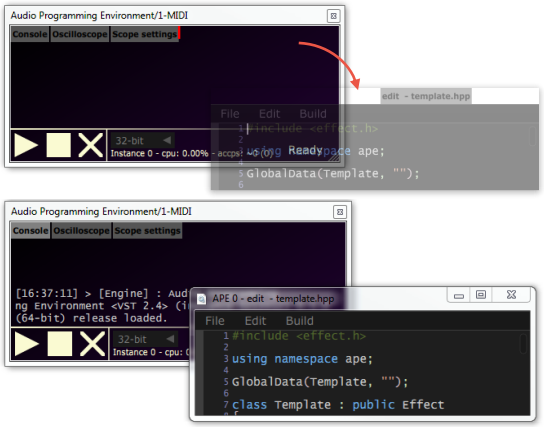
The UI uses a composite tabbed/dockable system to better utilize screen real estate. There are five main tabs you will work with:

1. The code editor
2. The running plugin UI
3. Console
4. Oscilloscope
5. Settings pane of the oscilloscope.

But for now, let us look at the plugin when you open it the first time - and how the general UI is structured.

Splash





A This is the main tabbing area. Tabs here can be selected by clicking to show content in the main area, and additionally also dragged out to a separate window, like shown to the right. Each window will have an identifier for this plugin instance in the window title. Simply close the window to reset the tab back into the dock.

B Here you can select the suggested floating point precision. While optional for plugins and compilers, if utilized it allows you to quickly recompile the plugin with a higher / lower floating point precision.

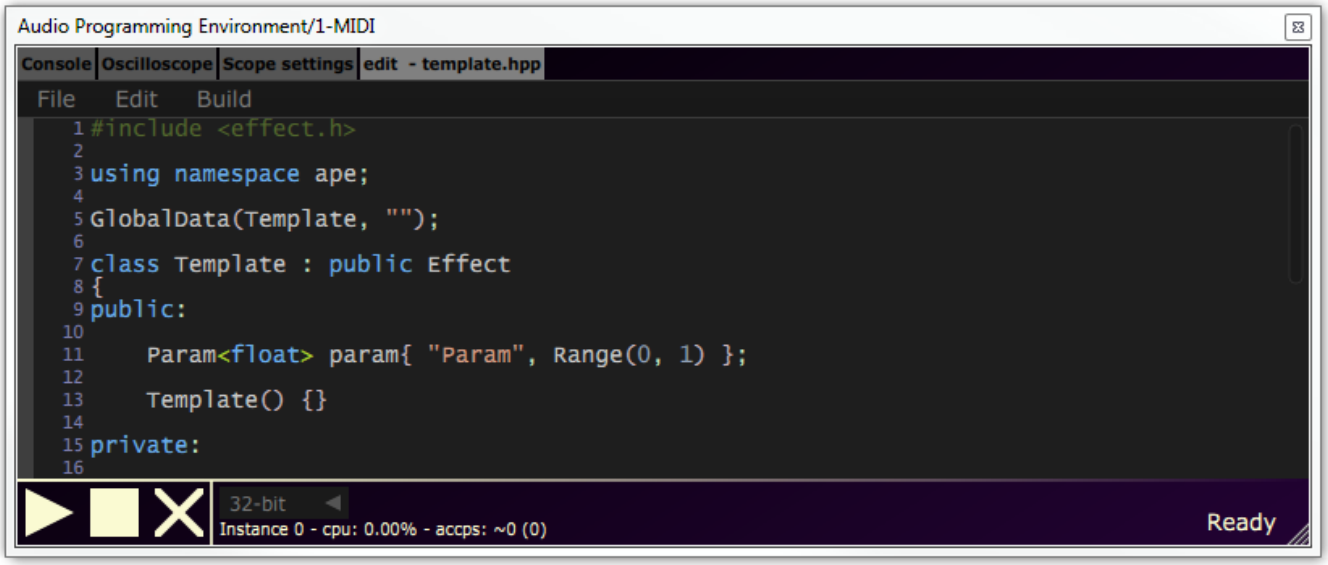
C These are your main "transport" controls. The play button recompiles and activates your plugin, while the stop button kills all ongoing activity. See the Editor settings for detailed activity hotkeys or the Menus section for a rundown of menu options pertaining to this. The cross cleans the compiler cache.

D This is a metrics bar - "instance *x*" is an identifier of the current plugin instance. CPU% is a relative measure of how much your plugin uses of a *single* core on your CPU. *accps* means "average cycle count per sample" and is a normalized per-sample (channel invariant) measure. Contrary to the percentage, this allows you to project CPU usage to other sample rates or different CPUs.

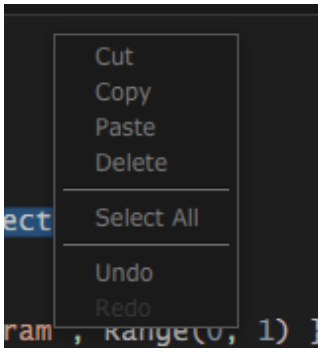
E Status messages - will let you know what is happening.

F Drag here to resize window.

Editor tab



This is the code editor - the heart of the plugin. Changes you make in here reflect whenever you recompile. It supports standard operations as shown to the right, with syntax highlighting for C++.



The editor by default auto indents your code, and you can scale the text size by holding control and scrolling. Defaults for these, as well as colour scheming, can be edited in the Editor settings.

Disk synchronization & autosave

The title of the editor will include a dash if you don't have any unsaved changes, and an asterisk if you do. If your currently edited file actually exists on disk, you will be asked whether to save once the plugin exists.

If you reload a saved plugin state with file contents mismatching whatever is saved on disk (the script is saved as a part of the session), you will be asked whether you want to reload the disk version instead. This behavior can be changed in the Editor settings.

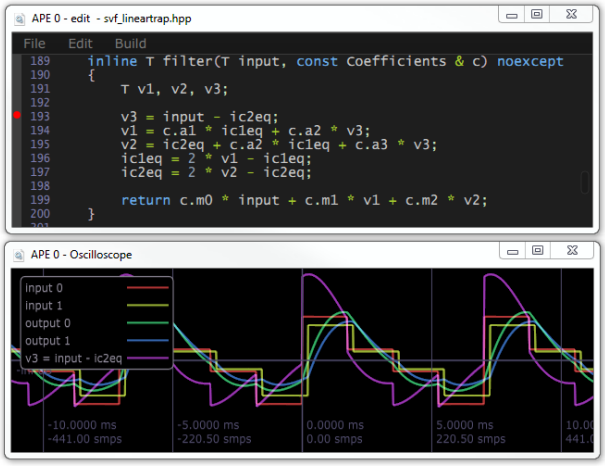
By default, your script will be autosaved every 60 seconds. If the session unexpectedly exits, you will be asked whether you want to recover your lost changes the next time you open the editor.

You can also edit your code externally, see Menus for setting this up and Development setup & source code if you want to have intellisense / auto-complete.

Menus

File		
	New File	Create a completely blank file
	New from Template	This loads the template file, as configured in the Language settings
	Open...	Open any file
	Open recent...	Open from a list of recently touched files
	Save	Save the current contents into the file, if it already exists
	Save as...	Save the current contents into a custom location
	Open home...	Open the home directory of your scripts, as configured in the Language settings
Edit		
	Edit externally	This will open the file in your system's default editor and watch it for changes. Whenever you save the file externally, APE will recompile the session automatically.
Build		
	Compile	Compile your script, but not activate it.
	Compile and Run	Recompile your script and activate it if possible - crossfading the old sound with the new one.
	Activate	Activate any currently compiled script. Activating and deactivating is a fast way of turning your script on and off.
	Deactivate	Deactivate any active script.
	Clean	This cleans the cached compilation of the runtime and library code - this is useful if you are developing or rewriting the runtime code (separate from the plugin code).

Scope probe



This is an experimental feature, but when used properly, it's an indispensable tool for understanding your code - it specifically aims to provide a similar workflow to oscilloscope probing in analog circuits.

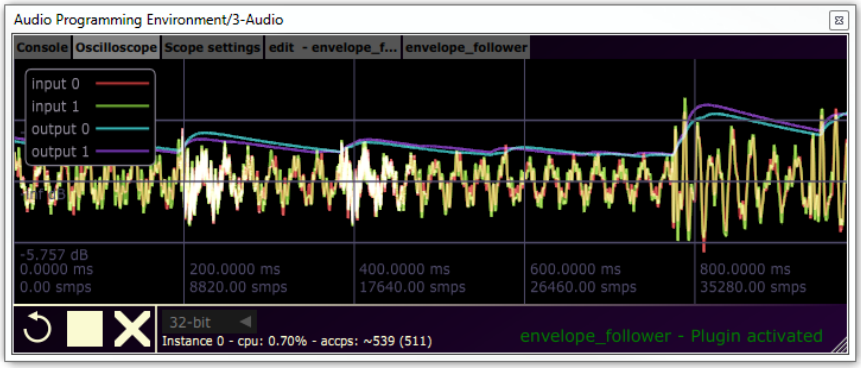
Clicking to the left of the line numbers in the editor inserts a "probing point" at an expression, which will - after a recompile - display it as an audio signal, with a sample every time the expression is evaluated. As you can see in the legend, there's now a new signal titled after the expression line you probed. You can probe up to 16 lines simultaneously.

This only really makes sense in a rendering context, and the plugin will protect you to some degree if you place a breakpoint that isn't called from a processing callback. Similarly, if you try to probe something that's not really an expression.

If your expression is evaluated at a different rate than your sampling rate, it will be resampled.

There's basic support for data types that can be converted to floating point values, including complex numbers. You can add traits support for custom data types - see the implementation here.

Oscilloscope tab & settings

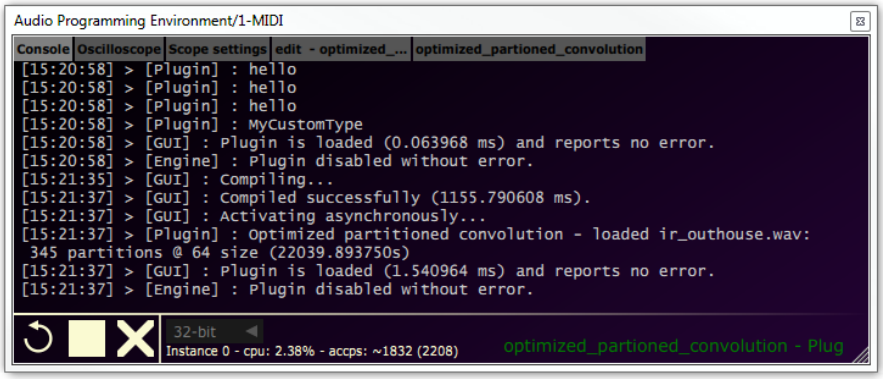


The oscilloscope displays a real time signal of your plugin's input and outputs, together with any scope probing points you have added - see Scope probe. You can drag around in the display and zoom in/out with the mouse wheel.

You can configure the oscilloscope in the settings tab:

For more information on how to use the oscilloscope, please check out the documentation and general usage of Signalizer instead.

Console tab

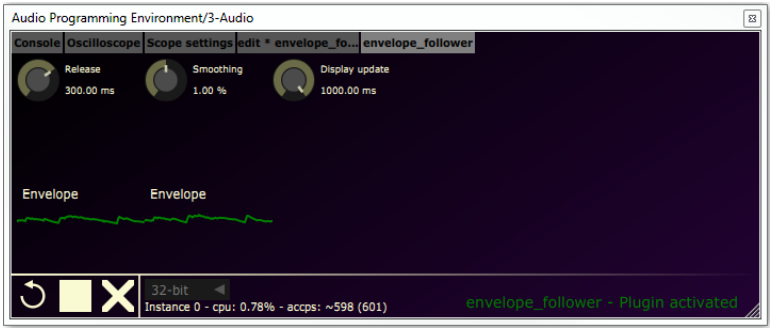


The console will in detail print info, warning and error messages from the GUI, compiler and runtime instances. If something goes wrong, you can find it in here. You can also print to the console from the plugin - see the various print APIs: print.h

You can also log the console to disk - see Configuration.

Plugin tab

When you have an active, running plugin, a new tab is opened for the parameters and widgets for the plugin script.



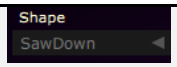
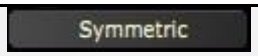
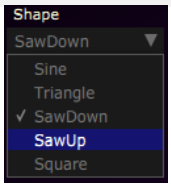
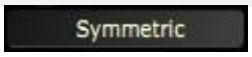
Parameter controls

Some of the widgets display various metered values like plots, but most importantly are the parameter controls. A complete list of widgets that can appear on the screen are defined here, in the online documentation. As for the interactive controls for parameters, there are three types:

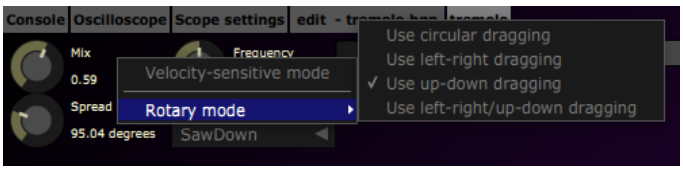
1. Knobs / sliders - for a range of values
2. Combo boxes - for a choice of predefined values
3. Buttons - for toggles.

The UI controllers for parameter values in scripts are automatically created and managed, with a class depending on the defined parameter type in the script. Knobs are for ranges (regardless of whether they are floating point or discrete), combo boxes are for enumerations while buttons are for boolean values.

The controllers reflect changes in parameters from the script or the host as well, as the parameters can be automated. Combo boxes and buttons have the simplest interface: Click them to change the value.

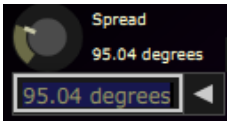
	Combo box	Button
Idle		
Clicked		

Knobs provide a range of customization features as well, to precisely dial in values. By right-clicking a knob, you can choose how it should behave when being dragged (A):



A

If you hold control while dragging, it temporarily goes into a very precise velocity sensitive mode. Note that all parameters internally are 64-bit for higher dialing precision.

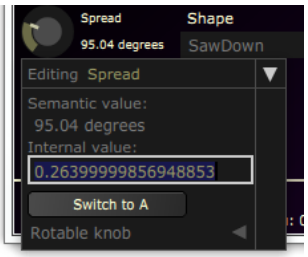


B

You can also double-click a knob to type a precise, semantic value (B):

Note that the values, units, and scaling is defined by the plugin script itself. You can study the options in detail in the online documentation.

If you click the triangle, you open an expanded edit space view (C):



C

In here, you can edit the normalized 0-1 value as well as A/B edits.

Configuration

The configuration file resides in the root of the APE directory and is called *config.cfg*. APE uses the open source library *libconfig* to read and parse the configuration file. The syntax is easy and structured, and can be learned from the linked homepage. Currently, the file consists of two segments, the application- and language settings.

Application settings

This section contains the global application settings.

```
application:
{
    log_console = false;
    use_buffers = true;
    use_fpe = false;
    num_channels = 2;
    force_single_precision = true;
    greeting_shown = true;
    unique_id = 1634755960;
    ui_refresh_interval = 80;
    console_std_writing = false;
    autosave_interval = 60;
    render_opengl = false;
    use_tcc_convention_hack = true;
    preserve_parameters = true;
}
```

Name	Type	Description
log_console	boolean	logs the output of the console into APE/logs/
use_buffers	boolean	deprecated
use_fpe	boolean	deprecated
num_channels	integer	sets APE to use the number of channels (ignored currently)
force_single_precision	boolean	if set to false, APE will default to the highest possible bit depth in the audio streams (ignored currently)
greeting_shown	boolean	if set to false, APE will show a welcoming message on startup
unique_id	integer	the id APE will use to identify itself (1634755960 = ascii constant 'apex')
ui_refresh_interval	integer	milliseconds, time between each redraw. This can be set very high if you don't care for quickly updating plots or displays (note, this will not affect normal redraw of the gui on events)
console_std_writing	boolean	logs the output of the console to stdout
autosave_interval	integer	amount in seconds between each autosave. Note that autosave only occurs if the code document has changed.
render_opengl	boolean	renders the gui using opengl. Can give performance increase on OSX
use_tcc_convention_hack	boolean	toggle this if you're having problems with knobs on x64

Editor settings

This section controls the code editor component.

```
editor:
{
    zoom = 1.20;
    x_offset = 50;
    y_offset = 50;
    auto_indent = true;
    enable_scopepoints = true;
    check_restored_against_disk = true;
    external_edit_command_line = "";

    hkey_save = "cmd+s";
    hkey_open = "cmd+o";
    hkey_new = "cmd+n";

    hkey_externaledit = "f10";

    hkey_compile = "f7";
    hkey_run = "f5";
    hkey_clean = "f8";
    hkey_activate = "f3";
    hkey_deactivate = "f4";

    colours:
    {
          background = "FF1E1E1E";
          highlight = "FF264F78";
          caret = "FFFFFFFF";
          line_number:
          {
                background = "FF1E1E1E";
                text = "FF7E7EAE";
          }
    }
}
```

Beyond ability to adjust general editor color scheme and hotkeys, there's a couple of UX settings:

Name	Type	Description
`zoom`	float	controls the text point scaling, based on a standard 12 point mono-spaced font.
`x_offset` & `y_offset`	integer	ignored currently
`auto_indent`	boolean	controls whether a newline is automatically indented similarly to the previous line
`enable_scopepoints`	boolean	if true, scope points will be enabled. Otherwise, they are ignored. See usage in Scope probe.
`check_restored_against_disk`	boolean	if true, whenever a session is reloaded, the plugin script contents will be compared against the disk version. If different, the user will be asked whether they want to use the disk version instead.
`external_edit_command_line`	string	by default, when editing a file externally, the system shell will be invoked with just the path to the file. With this setting, you can prepend some arguments or select a particular program.

This will allow you to change the hotkeys of the editor. `cmd` is `command` on OSX and `control` on Windows.

Language settings

This section contains the language settings. It has the following structure:

```
languages:
{
    default = "c";
    default_file = "examples/template.c";

    <languageₙ-specification>
}
```

The `default` setting is the default language that the editor will select, and what will be assumed if you save a file without extension. `default_file` is what the editor will open on a fresh load, or when you use the "New from template" menu option in the editor.

The language specification instructs the program on what to with a specific language. APE supports any number of language specifications, as languages and compilers are selected and identified from the file extensions. They have the following structure:

```
<language-name>
{
    extensions = ("");
    compiler:
    {
            name = "";
            path = "";
            arguments = "";
            exports:
            {
                    <exportₙ> = <exportₙ-name>;
            }
    }
    syntax_highlight:
    {
            error =              "FF8B0000";
            comment =            "FF008000";
            keyword =            "FF69ADEE";
            operator =           "FFA6CE2E";
            identifier =         "FFC8C8C8";
            integer =            "FF8E9EB3";
            float =              "FF9E8EB3";
            string =             "FFD69D85";
            bracket =            "FFCCB7C0";
            punctuation =        "FFC9E6BD";
            preprocessor_text =  "FF556B2F";
    }
}
```

`language-name` is the name of the language. `extensions` is a list of strings, comma-separated, that denotes what file extensions the editor can open and what language they are to be associated with. `name` is the name of the compiler (irrelevant, but for debugging), `path` is the relative path to the module that is the compiler, and `arguments` is a classic command line that is passed to the compiler.

`exports` is an optional group of aliases for the names APE look up. See the section Compiler API for more info. You can change the name APE looks up by using this pattern:

```
GetSymbol = "x";
```

Where "x" is a string representing the usually decorated name.

The `syntax_highlight` part allows you to change the color scheme for text in the editor for that particular language.

CppApe: User scripting & API documentation

CppApe is the default, shipped compiler and user facing "scripting" language in APE. CppApe implements C++17 and most of C++20, being based on bleeding edge (as of today) clang[ii] for the C++ frontend, ccore[iii] for the C standard library and libc++[iv] for the standard C++ library. libCppJit[v] for the execution engine (which is based on llvm[vi]).

This together forms a modern and fast C++ JIT environment: You should be able to expect hot recompilation times of 400-1000ms, depending on your computer and the script size. There's complete documentation online of the scripting API:

- http://jthorborg.com/content/ape/doc_05/namespaceape.html

Plugin anatomy: Getting started

Here we see the standard template, which is by default what is shown when you open the plugin. The default file can be changed in the Language settings. CppApe uses an object-oriented approach to defining a plugin, similar to all plugin SDKs out there. Basic knowledge of C++ is assumed here.

```
#include <effect.h>
using namespace ape;

GlobalData(Template, "");

class Template : public Effect
{

public:

   Param<float> param { "Param", Range(0, 1) };
   Template() {}

private:

   void start(const IOConfig& cfg) override
   {

   }

   void process(umatrix<const float> inputs, umatrix<float> outputs, size_t frames) override
   {
         const auto shared = sharedChannels();
         const float paramValue = param;

         for (std::size_t c = 0; c < shared; ++c)
         {
               for (std::size_t n = 0; n < frames; ++n)
                     outputs[c][n] = inputs[c][n];
         }
   }
};
```

The first line of interest is:

```
GlobalData(Template, "");
```

Here you declare what plugin is to be instantiated in this file. This needs to be kept in synchronization with the plugin effect name. The string literal is an optional small description of the plugin.

Next, your choice of base class for your plugin defines available capabilities and callbacks: `Effect`s are uniform input/output processors, while `Generator`s are plugins that don't take any input but still provide sound. There is also `Transport*` variants of these, see the hierarchy here.

Methods for basic events and callbacks

APE communicates with your plugin through the following methods, whenever information is needed, or events are available to you. All of these methods are invoked synchronously with respect to each other. Again, see the online documentation for in depth explanations.

- `init` / `close`

These are optional but gives you a chance to do setup / teardown outside of object constructors / destructors.

- `start` / `stop`

`start` is called whenever the plugin I/O changes (including first time, after `init`). This is where you want to do static resampling and coefficient calculations, load files, resource allocation etc. `stop` is called whenever the plugin stops processing. Good place to release resources allocated in `start` - these are always called in pairs.

- `play` / `pause`

These are only available if you use a `TransportEffect` or a `TransportGenerator`. These track the transport status in the user's host and produce callbacks whenever the user starts playing the timeline or pauses it. These are called in a real-time context.

- `process`

Here you can read the audio inputs to your plugin and write to your audio outputs. This is called in a real-time context, so be careful with blocking calls. You can stream audio to files as well. As a `Transport*` you additionally have access to advanced timing info (BPM, time etc.) through the member function `getPlayHeadPosition()`.

Details on lifetime, locals, exceptions

- The plugin class in the loaded file gets constructed whenever you activate or disable the project
- Globals are created and destroyed in the same manner, static locals are only created if needed
- There's no type info
- RAII is supported, with limited support for exceptions (due to lack of type info), you should use the custom abort or assert to check invariants and abort execution of the plugin in a controlled way
- Otherwise, all local variables (including special types like parameters, files etc.) should be a part of your plugin class so you can nest and embed plugins

Nesting plugins

In order to reuse another plugin, simply `#include "your_other_plugin.hpp"` in your file, and wrap it inside your own plugin using the `Embedded*` class of wrappers.

Plugin SDK

The CppApe plugin SDK is automatically included so long as you `#include <effect.h>` or `#include <generator.h>`. All headers are available as system includes as well, so you can angle bracket `parameter.h` for instance. An in-memory precompiled header is automatically created for all standard includes.

C++ standard library

The following headers are guaranteed to exist, with varying levels of conformance.

Standard header	Conformance	Notes
`algorithm`	Full	
`any`	Full	
`atomic`	Partial	Sequential consistency `std::atomic<T>` guaranteed to function
`bitset`	Partial	Missing I/O
`cassert`	Full	
`complex`	Full	
`cfloat`	Full	
`climits`	Full	
`cmath`	Full	
`cstdarg`	Full	
`cstddef`	Full	
`cstdint`	Full	
`cstring`	Full	
`functional`	Full	
`initializer_list`	Full	
`iterator`	Full	
`limits`	Full	
`list`	Full	
`map`	Full	
`memory`	Partial	No `get_temporary_buffer`
`new`	Partial	Not allowed to overload global operator `new`, `delete`
`numeric`	Full	
`optional`	Full	
`queue`	Full	
`random`	Partial	
`ratio`	Full	
`set`	Full	
`string`	Partial	No I/O, only works for `char`
`string_view`	Partial	No I/O, only works for `char`
`tuple`	Full	
`type_traits`	Full	
`unordered_map`	Full	
`unordered_set`	Full	
`utility`	Full	
`valarray`	Full	
`variant`	Full	
`vector`	Full	

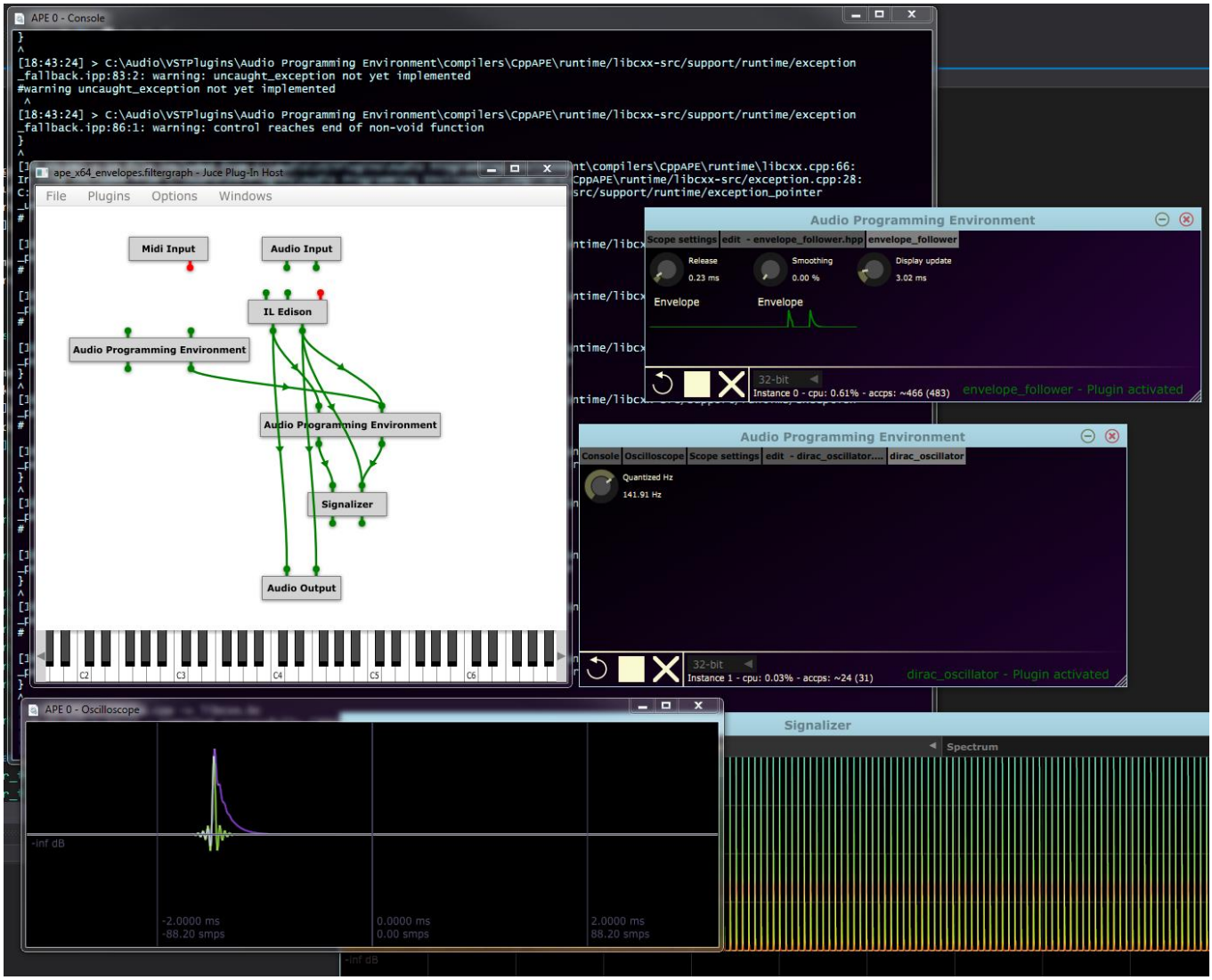
SIMD and template precision language extensions

The header `<fpoint.h>` declares a type called `ape::fpoint`, which is a floating point type with user-specified precision. Use this type generally for floating point state variables, so the precision of the plugin can be controlled by the user.

The header `<simd.h>` declares a SIMD type called `ape::vector_register<T>`, which can hold primitive arithmetic types and automatically contains the most efficient number of lanes for the host computer. Basic scalar and `std::complex<>` arithmetic operators are available. LLVM will optimize this type aggressively using target specific vector instruction sets (like AVX). See more in the documentation.

Standalone workflow

While it would be nice to have a standalone version, for now the free and open source JUCE Plugin-In Host generally provides a really good workflow, including patching support, I/O device selection and project recall. I would suggest just using this, instead of trying to compile the plugin as a standalone for now.



There's a tutorial on how to get set up like this from JUCE:

https://docs.juce.com/master/tutorial_create_projucer_basic_plugin.html

Development setup & source code

To get the source code for APE, visit the online repository here on BitBucket:

https://bitbucket.org/Mayae/ape

It is also mirrored on GitHub:

https://github.com/jthorborg/ape

Prerequisites

Note the target plugins that will be produced (e.g. VST2.4 / VST3.6, AAX, AU) needs to be configured in

```
projects/plugin/JuceLibraryCode/AppConfig.h
```

Windows 7, 8 or 10:

- Git
- Python 3.6
- Visual Studio 2017 at least, with C++ development installed.
- You will need to add your target plugin SDK as an include directory to the Plugin project.

MacOS 10.14+:

- Git
- Python 3.6
- Xcode 10.12
    - C++ `std::filesystem` support:
        - Brew
        - Boost
    - If building audio units, you probably need to add the old audio unit SDK into the Xcode `extras` folder.
- Xcode command line tools
- You will need to add your target plugin SDK as an include directory to the Plugin project.

Repository folder layout

Here are the most noteworthy entry points in the repository:

Path	Contains
`doc/`	• Doxygen files for generating documentation • This manual
`external/`	• git submodules • external projects and dependencies
`make/`	• Post processing scripts • Build configuration: Here you instruct where to install the plugin when building APE • Plugin release skeleton (see Plugin folder layout for details)
`projects/`	• These are the different projects that make up APE together. • `builds/` - in here you will find project configurations under for different IDEs • `src/` contains the source code.
`shared-src/`	• The APE compiler extension headers, shared definitions between compilers, scripts, APE etc.
`solutions/`	• Contains a complete solution with all projects, for different IDEs. See also Building APE. • This also contains a project for CppAPE scripts, where you can develop with syntax highlighting
`prepare.py`	• Preparation script for setting up the development environment. See Building APE.

Plugin folder layout

Here are the most noteworthy entry points in an APE release plugin:

Path	Contains
`compilers/`	• Here you will find included compilers, that you can load - see the Language settings.
`examples/`	• These are the shipped examples in the plugin. This is a clone of the following repository: bit-bucket.org/Mayae/ape-snippets
`includes/`	• Include files for CppApe scripts • `common.h` is special: It is cached as a precompiled header • `ccore/` contains the minimal embedded C standard library headers • `libcxx/` contains the headers subset of the Error! Reference source not found. • `shared-src/` contains the shared headers between APE, CppApe (and others) and the final scripts.
`licenses/`	• Licenses for the end user to agree to
`presets/`	• Preconfigured presets for the end user. Primarily oscilloscope presets. `init.oscillo-scope.sgn` is special and is always loaded by default.
`resources/`	• Icons, primarily (and mostly for Signalizer's Oscilloscope)
`config.cfg`	• The plugin configuration file. See Configuration.

CppAPE compiler folder layout

Path	Contains
`compilers/CppApe/build/`	• Temporary build files. • If you build APE in debug configuration, intermediate LLVM IR files will also be written to disk here. • `postfix.cpp` - this file is appended to any file compiled inside APE.
`compilers/CppApe/runtime/`	• Persistent build files: precompiled headers of `common.h`, compiled versions of libcxx and the CppApe runtime • `runtime.cpp` - this file contains the runtime C binding interface to the CppApe compiler. • `libcxx-src/` - this is the source of the runtime of the C++ standard library.

Building APE

Clone the repository firstly:

```
$ git clone https://bitbucket.org/Mayae/ape.git
```

Checkout a suitable branch (`master` for released checkpoints, `dev/stable` for bleeding edge):

```
$ git checkout master
```

Then, you need to prepare your development setup. This is automated in the `prepare.py` script:

```
$ py -3.6 prepare.py
```

This checks out all submodules, links projects together, downloads any needed SDKs and binaries.

From here on, just open your preferred solution and execute one of the built in targets, like `debug`, `testing` or `release` / `archive`.

Compiler API extensions

APE is designed to be extensible and open. It supports any language so long as it has a compatible compiler. This way, you can write in your preferred language. When APE is requested to compile and load the current script in the editor, it requests the editor for a project information struct. The editor composes this struct with information about the current relevant files and identifies the language based on the file extension of the main file.

APE passes this information on to the code generator inside APE. The code generator is responsible for communication between APE and the script. The code generator looks up the desired language in the configuration file, based upon the language ID (derived from the file extension). Here it collects information about the compiler settings to be used.

Using the supplied path, it loads a module (DLL on windows, Bundle on Mac, SO on *nix). APE will then try to retrieve pointers using the exported names in the configuration file for all the required functions. Here's a list of what APE expects to be exported (if nothing is otherwise defined):

```
CreateProject
CompileProject
InitProject
ActivateProject
DisableProject
ProcessReplacing
OnEvent
ReleaseProject
CleanCache
```

As long as the above functions returns correctly, APE doesn't have to know anything about any specific language or whatever. Indeed, the compiler may host a client process to do the actual processing, and pass buffers using shared memory.

If you want to create your own compiler, first take a look at the CppApe project in the repository which is the current recommended approach. You will find extensive, although slightly out of date, documentation in the former manual. The following files in the repository provide the skeleton for creating a new compiler, that can be loaded directly be APE:

```
CompilerBindings.h
```

This is the C interface for communicating with APE.

```
CppCompilerInterface.cpp
ProtoCompiler.hpp
```

`ProtoCompiler` is a scaffolding base class for creating an object oriented stateful compiler.

Migration from previous compilers and languages

If you are looking for documentation for previous versions, take a look at the former manual.

Tcc4APE

Tcc4APE was a restricted subset of the C language. All the same functionality exists as safe API for CppApe: User scripting & API documentation instead, in a faster and more modern language.

While we had a good run with TCC, its performance is limited and C itself does not provide much in terms of removing boilerplate.

syswrap

Syswrap is still shipped in source in the repository, but currently not a part of the binary release. It simply hasn't been ported yet, if you were relying on the feature set please contact me: Downloads, videos, contact, links

Releases and changelog

Here are links for the binary releases of the software.

- Version 0.5.0
  - https://bitbucket.org/Mayae/ape/downloads/
- Version 0.3.0
  - http://jthorborg.com/content/ape/ape_release_0.3.0.zip

The changelog is in DD-MM-YYYY format.

Take a look at revisions in the following branch for extensive details:

- https://bitbucket.org/Mayae/ape/branch/dev/stable

Alpha 0.5.0

Released 01-06-2020.

- Version controlled
- New repository: https://bitbucket.org/Mayae/ape
  - New source structure with permanent platform projects
  - Most dependencies now included as submodules
  - Modular and testable components
  - Unit tests
- Build system
- Headers shared between plugin and user scripts, removes stale errors
- Complete rewrite to modern C++
- All manual memory management and leaks removed
- All UI, utility etc. now uses cpl
- Extended platform support, including technical Linux support. See requirements.
- Parameters
  - Much more precise user controls, with ability to type in precise 64 bit values
  - More flexible and extensible format / range options for parameter values
  - Enumeration / lists of strings now supported as combo boxes
  - Now automatable by host
  - 64-bit precision internally
- Widgets
  - Meters are now per-sample evaluated and properly decaying. Also contains peak markers.
- Iteration
  - Compatible parameter values preserved
  - Hotkeys for all major operations
  - True multithreaded compilation across plugins
  - Old/new sound blended on swapping instances
- Engine
  - Optimized and built-in FFT
  - Support for streaming audio files to and from disk, optionally resampled
  - Audio thread interactions now completely lock free
  - Precise transport access and playback state events to the plugin
- Quality of life
  - SDKs and libraries now ship included, removed reliance on user development setup
  - Plugin callbacks for initialization and reconfigurations now run asynchronously to avoid stutter on audio thread and hiccups on main thread
  - Many more checks of resource managements, assertions etc. to make it much more safe
  - Working code is serialized into the project as well, instead of referencing a script on disk.
  - User is notified mismatched / out-of-date scripts
  - Removed nonsensical errors on abandoned save dialogs
  - Long operations timed and printed to the console
- Plugin GUI
  - Resizable
  - Redesigned, bit buttons removed in favor of simple icons and hotkeys
  - Now completely uses vector graphics instead of bitmaps
  - Switched to a tabbed system to increase real estate
  - Tabs can be orphaned into separate desktop windows, and redocked back
  - Graphics optimized and employs precise redrawing, much faster on OS X using core graphics renderer
  - Subpixel text rasterization for normal DPI displays

    - Removed "fpu exceptions" and "protected buffers" switches. These are now determined by compilation mode.
- Source code editor
    - Externally editing files is now supported, reloading and recompiling whenever the file is saved externally
    - Full project and intellisense when working in the source repository for user scripts
    - Evaluate source code expressions as "breakpoints"
    - Text scaling
    - Auto indentation
    - Saving a file without extension and determined language appends the default language extension
    - Menu option to open "home" (also configurable) scripting directory
    - Menu option to create a new file, cloned from the template file
    - Menu options for build events (compile, activate, clean, edit externally etc.)
    - Default now with a dark theme
- New compiler / language: CppApe
    - C++ 17 bleeding edge compiler, based on Clang
    - Runtime vehicle is libCppJit: Multithreaded, lazy JIT based on LLVM
    - User scripts can now include and use other scripts
    - Completely revised front end together with safe and idiomatic user API, boiler plate removed
    - Access to most of C++ standard library, based on libcxx
    - Subset of C standard library available, based on ccore
    - Built-in SIMD vectorized math
    - DSP primitives, interpolation algorithms
    - Type safe and much faster print() family functions
    - 32-bit / 64-bit / 80-bit templated math precision, switchable by user in UI
    - Typical scientific math constants available as templated constant expressions
    - Complete user API documentation here: TODO
    - Assertions supported
    - RAII and unwind support
    - Some exception support
    - Globals, static constructors and destructors supported
    - Memory mapped and precompiled system headers for compilation speed
- Oscilloscope
    - Based on Signalizer
    - Per-sample source code expression evaluation and graphing
    - Color coded inputs / outputs
    - User-defined triggering
- Bugs
    - Console is now thread safe
    - Compilation is now thread safe
    - Fixed crashes on immediate deserialization
    - Many user file bugs fixed
- Tcc4APE
    - More or less deprecated, still ships in source form but complete support is missing
    - Same for syswrap.

Alpha 0.3.0

Released on 08-04-2014.

- Source code rewritten to support JUCE also, which is the primary target platform now.
    - this affects several things; notably the editor is switched from Scintilla to JUCE's inbuilt code editor now
        - this has the welcome side effect of hotkeys working again
    - syntax highlight only for C++ and friends for the moment.
- x32 and x64 builds on both Windows and OSx as AudioUnit and VST 2.4
- Countless bug fixes / code rewrite
- project recall now implemented
- autosave now implemented
- support for high dpi display
- an actual threading- and multi-instance model is now implemented; it should be completely safe to run multiple plugins in the same or other processes
- fix of fpu exceptions
- improved header support for other compilers than tcc

Alpha 0.2

Released on 10-02-2014

- fixed uninitialized variable 'Engine::clocksPerSample'
- scilexer now properly adds filenames to project struct even in case of singleString-compilation
- scilexer now properly sets amount of files in the project
- the console should now properly print strings with linebreaks in them, this affects the core, api and scripts.
- fixed a bug where newlines will crash the console code.
- output logging of console now properly contains newlines.
- due to larger amounts of info being printed to the console, it is now scrollable and has a longer history
- added new compiler: syswrap. syswrap allows to interface to installed system compilers.
- fixed a bug where closing the editor would not reset the editor button in APE
- pressing the editor button now properly restores the window if user had minimized it before
- fixed a memory leak in the TCC compiler (early return caused no deallocation of plugin data)
- fixed a wrong return value in CInterface.h
- added a new knob function: api.createRangeKnob(). This knob formats its display value based off a minimum, maximum and a callback function.
- fixed a bug where knobs initially would have the wrong format
- to enhance c++ compability, 'this' is now an illegal identifier
- CInterface.h:
    - added new valuestruct: scale
    - added f_mod() and f_sin()
    - added pi values
    - compatized header with various compilers
- updated the example scripts to reflect these changes.

Alpha 0.1

Initial version.

Licenses

See `/licenses/`.

Credits and thanks

Thanks to the helpful community at *kvraudio*[vii] - extremely helpful resource and excellent site.

Thanks to *stackoverflow*[viii] - always helping with design / coding issues

Thanks to *musicdsp*[ix] - besides being a hugely helpful site with many examples, it inspired me, this project and delivered source code to several of the included plugin examples.

Thanks to innovators and coders of Scintilla, TCC, VSTGUI[x], libconfig, JUCE[xi], Steinberg[xii], LLVM project, and MinGW[xiii] for delivering incredibly nice products that without doubt makes projects like mine realizable.

Thanks to Jim Credland for developing dockable windows in JUCE, and to Roland Rabien for developing a file system watcher utility for JUCE as well.

Huge thanks to Mikkel Juel Gregersen for software testing, manual / logo / general visual design, assistance with video.

Downloads, videos, contact, links

Did you create a cool effect you want to share and possibly include in further releases of APE? Need support? Got inquiries about the product? Have some feedback/suggestions? Any violations I need to know about? I can be contacted at the following email: contact@jthorborg.com

See also here for more information:

http://jthorborg.com/index.html?ipage=contact

Check in at APE's online page for updates and more setup:

http://jthorborg.com/index.html?ipage=ape

Go here to download ready-to-use plugins:

https://bitbucket.org/Mayae/ape/downloads/

Keep up to date on my projects on my YouTube channel:

https://www.youtube.com/channel/UCFjuzzviaLdTh71-ZiWEcKw

Here's the previous manual for v. 0.3.x:

http://jthorborg.com/content/AudioProgrammingEnvironment_Manual.pdf

Don't forget to study the online documentation if you have a problem:

http://jthorborg.com/content/ape/doc_05/namespaceape.html

Get more scripts here:

https://bitbucket.org/Mayae/ape-snippets/src/master/

Above all, have fun and hopefully this is useful to you.

Janus Lynggaard Thorborg

Endnotes

[i] libconfig: http://www.hyperrealm.com/libconfig/

[ii] clang: https://clang.llvm.org/

[iii] ccore: https://bitbucket.org/Mayae/ccore

[iv] libc++: https://libcxx.llvm.org/

[v] libCppJit: https://bitbucket.org/Mayae/cppjit/src/master/

[vi] llvm: http://llvm.org/

[vii] kvraudio: http://www.kvraudio.com/

[viii] stackoverflow: http://stackoverflow.com/

[ix] musicdsp: http://www.musicdsp.org/

[x] vstgui: http://sourceforge.net/projects/vstgui/

[xi] juce: http://www.juce.com

[xii] steinberg: http://www.steinberg.net

[xiii] minGW: http://www.mingw.org/