Audio Programming Environment - Manual

[Introduction 3](#_Toc39239138)

[Features 3](#_Toc39239139)

[WIP features 4](#_Toc39239140)

[Parameter controls 4](#_Toc39239141)

[Requirements 6](#_Toc39239142)

[Minimum 6](#_Toc39239143)

[Recommended 6](#_Toc39239144)

[Tested & shipped versions 6](#_Toc39239145)

[Experimental support 6](#_Toc39239146)

[User interface 7](#_Toc39239147)

[Splash 7](#_Toc39239148)

[Editor tab 10](#_Toc39239149)

[Disk synchronization & autosave 11](#_Toc39239150)

[Menus 13](#_Toc39239151)

[Scope probe 13](#_Toc39239152)

[Oscilloscope tab & settings 15](#_Toc39239153)

[Console tab 15](#_Toc39239154)

[Plugin tab 16](#_Toc39239155)

[Parameter controls 16](#_Toc39239156)

[Standalone workflow 18](#_Toc39239157)

[Configuration 19](#_Toc39239158)

[Application settings 19](#_Toc39239159)

[Editor settings 20](#_Toc39239160)

[Language settings 21](#_Toc39239161)

[CppApe: User scripting & API documentation 23](#_Toc39239162)

[Getting started 23](#_Toc39239163)

[Development setup & source code 24](#_Toc39239164)

[Compiler API 24](#_Toc39239165)

[Compilation, activation, idle and run process - function reference 24](#_Toc39239166)

[Shipped compilers 28](#_Toc39239167)

[Tcc4Ape 28](#_Toc39239168)

[Script structure 28](#_Toc39239169)

[TCC API Function Reference 29](#_Toc39239170)

[User-defined functions and instance pointers 33](#_Toc39239171)

[Macros and functions 34](#_Toc39239172)

[Types 35](#_Toc39239173)

[syswrap 35](#_Toc39239174)

[Scripts 35](#_Toc39239175)

[syswrap compiler API 36](#_Toc39239176)

[Download links 38](#_Toc39239177)

[Changelog 38](#_Toc39239178)

[Licenses 41](#_Toc39239179)

[Credits and thanks 41](#_Toc39239180)

[Contact 42](#_Toc39239181)

[Donations 42](#_Toc39239182)

# 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.

## 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
* 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

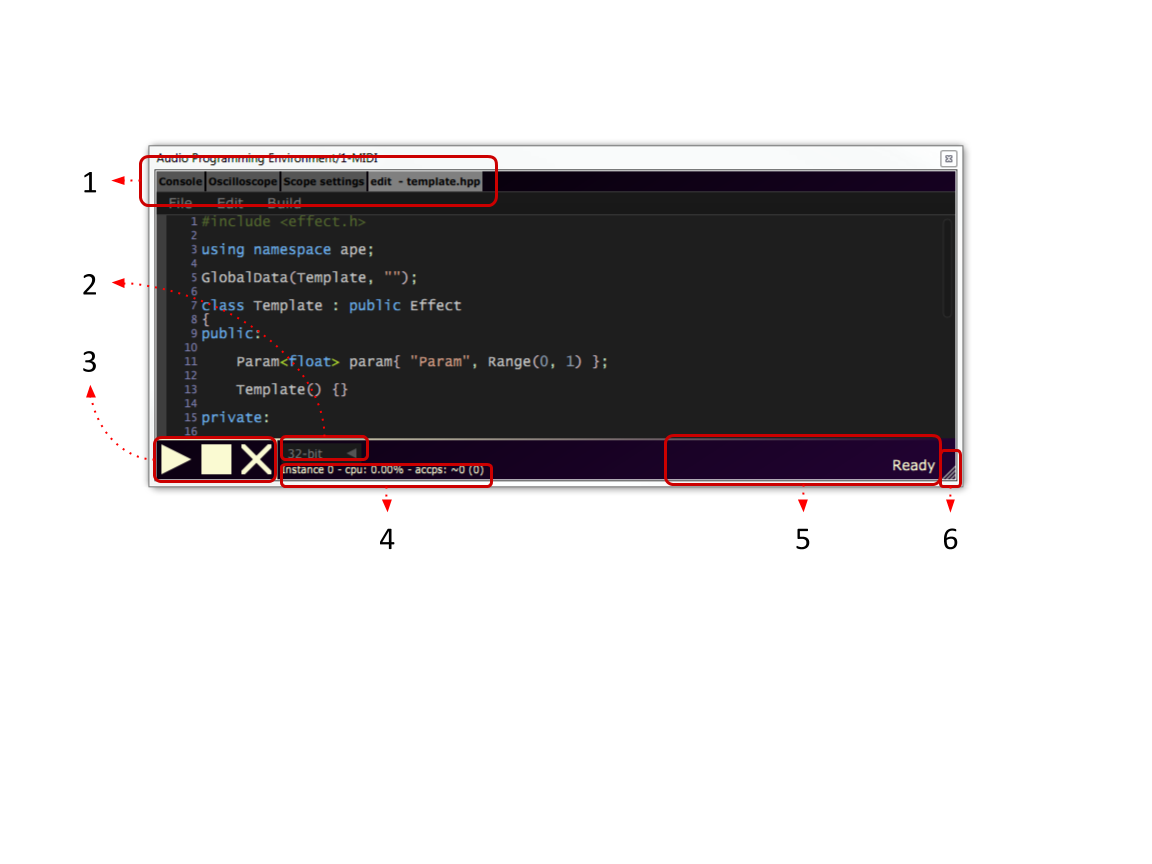
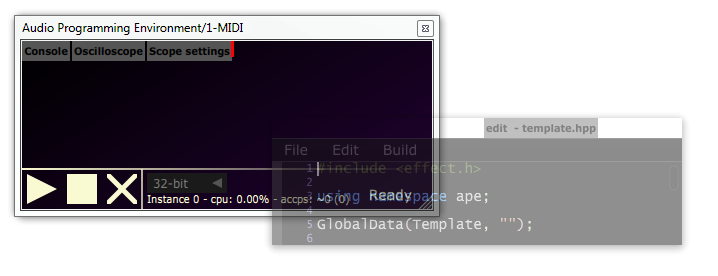
# User interface

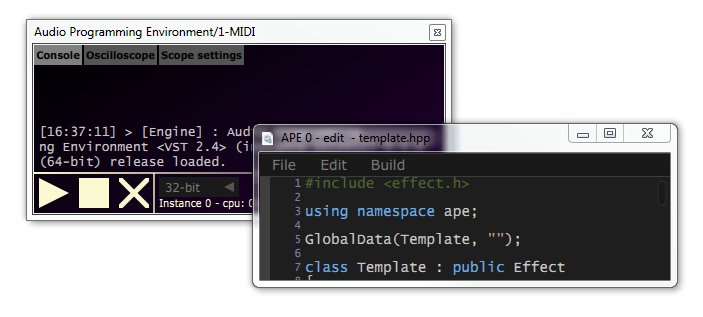
The UI uses a composite tabbed/dockable system to better utilize screen real estate. There are five main tabs you will work with:

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

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

## Splash



1) 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.

2) 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.

3) 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

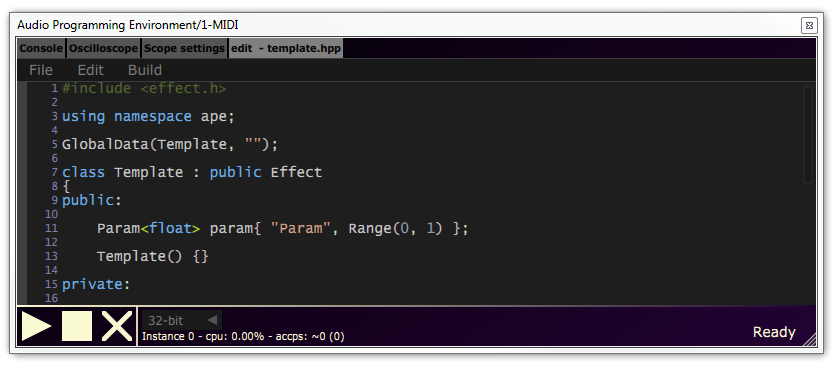
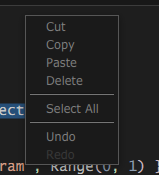
Editor for a rundown of menu options pertaining to this. The cross cleans the compiler cache.

4) 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.

5) Status messages - will let you know what is happening.

6) 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 .

### 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 behaviour can be changed in the .

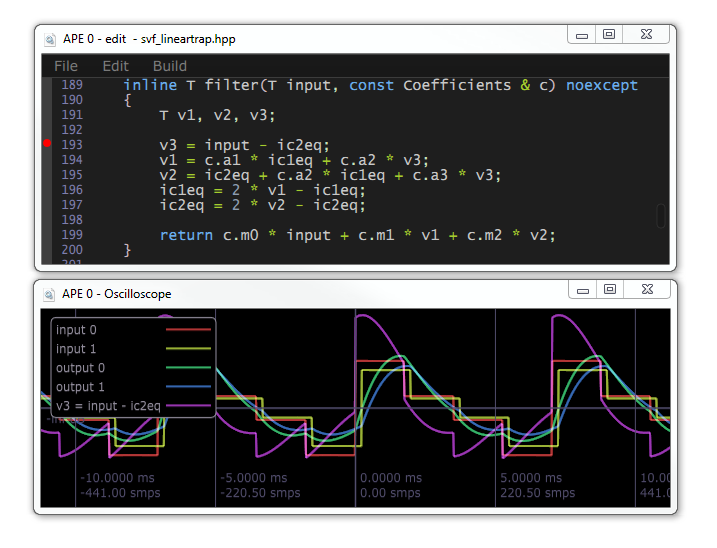
By default, your script will be autosaved every 60 seconds. If the session unexpectedly exits, you will be asked whether you want to recover your lost changes the next time you open the editor.

You can also edit your code externally, see for setting this up and 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.

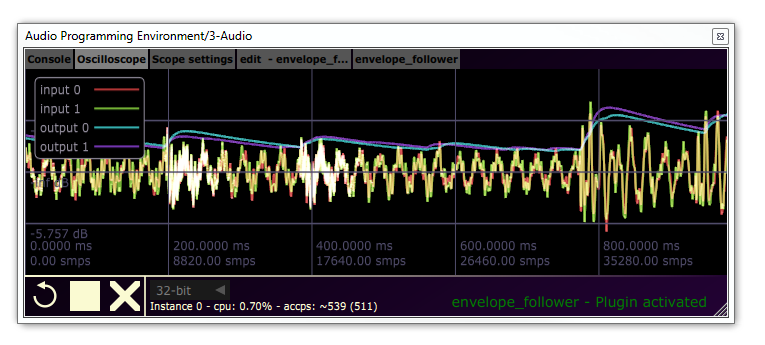
**Clicking to the left of the line numbers in the editor** inserts a "probing point" at an expression, which will - after a recompile - display it as an audio signal, with a sample every time the expression is evaluated. As you can see in the legend, there's now a new signal titled after the expression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

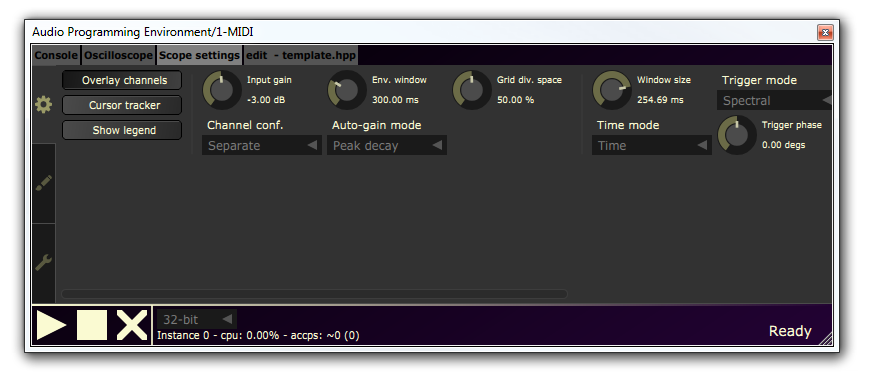
There's basic support for data types that can be converted to floating point values, including complex numbers. You can add traits support for custom data types - see the implementation [here](https://bitbucket.org/Mayae/ape/src/550e981a9f6db712c97612d64dadd0901ed12c45/make/skeleton/includes/trace.h?at=dev%2Fstable#lines-68).

## Oscilloscope tab & settings



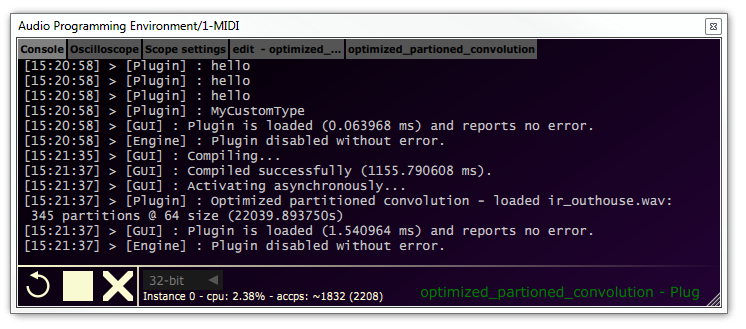
The oscilloscope displays a real time signal of your plugin's input and outputs, together with any scope probing points you have added - see . You can **drag around in the display**, and **zoom in/out with the mouse wheel**.

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](http://jthorborg.com/index.html?ipage=signalizer) instead.

## Console tab

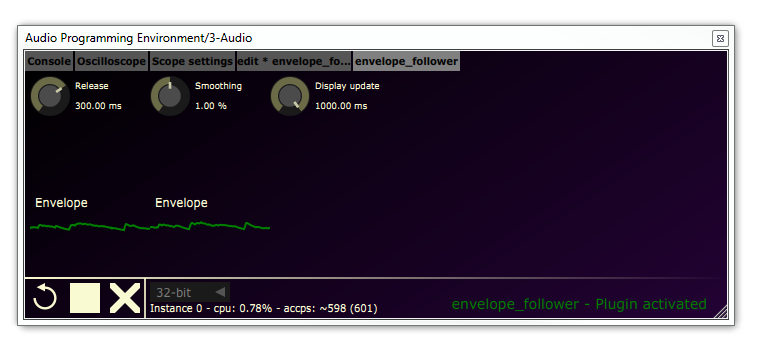


The console will in detail print info, warning and error messages from the GUI, compiler and runtime instances. If something goes wrong, you can find it in here. You can also print to the console from the plugin - see the various print APIs: [print.h](http://jthorborg.com/content/ape/doc_05/print_8h.html)

You can also log the console to disk - see .

## 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](http://jthorborg.com/content/ape/doc_05/classape_1_1_u_i_object.html), in the online documentation. As for the interactive controls for parameters, there are three types:

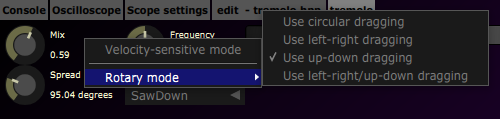
1. Knobs / sliders - for a range of values
2. Combo boxes - for a choice of predefined values
3. Buttons - for toggles.

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.

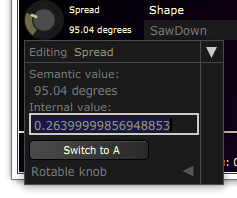
|  |  |  |
| --- | --- | --- |
|  | Combo box | Button |
| Idle |  |  |
| Clicked |  |  |

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:

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.

You can also **double-click** a knob to type a precise, semantic value:

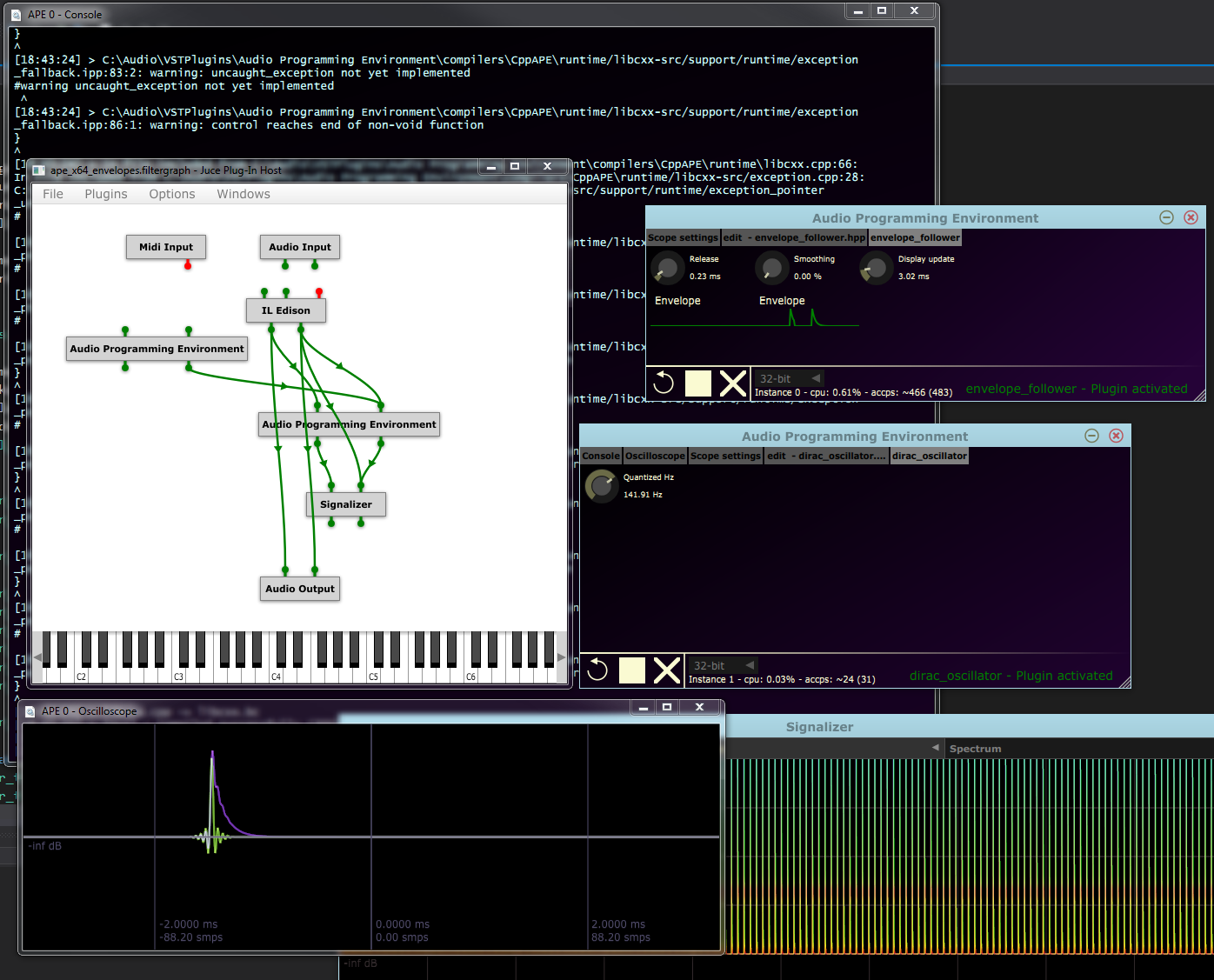
Note that the values, units and scaling is defined by the plugin script itself. You can study the options in detail in the [online documentation](http://jthorborg.com/content/ape/doc_05/classape_1_1_param_3_01double_01_4.html#af55994f48d5760e9fa331fbd19342db1).

If you **click the triangle**, you open an expanded edit space view:

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

# 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>

# Configuration

The configuration file resides in the root of the APE directory and is called *config.cfg*. APE uses the open source library *libconfig[[1]](#endnote-2)* 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;

}

* 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 amount of channels (ignored currently)
* force\_single\_precision (boolean): if set to false, APE will default to the highest possible bitdepth 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): miliseconds, time between each redraw. This can be set very high if you dont care for quickly updating plots or displays (note, this will not affects 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 colour scheme and hotkeys, there's a couple of UX settings:

* zoom (float): controls the text point scaling, based on a standard 12 point monospaced 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 .
* 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 colour 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[[2]](#endnote-3) for the C++ frontend and libc++ [[3]](#endnote-4)for the standard C++ library. libCppJit[[4]](#endnote-5) for the execution engine (which is based on llvm[[5]](#endnote-6)).

This together forms a modern and fast C++ JIT environment.

* Rundown of a plugin structure, different entry points and documentation link.

## Getting started

# Development setup & source code

The source code for both the APE plugin and the compiler Tcc4Ape and syswrap can be found in the */src/* directory. See */licenses/* for details on using it, and *readme* for compiling instructions.

# Compiler API

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 info 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):

GetSymbol

CompileProject

SetErrorFunc

ReleaseProject

InitProject

ActivateProject

DisableProject

GetState

AddSymbol

ProcessReplacing

OnEvent

As you probably can see, all communication between the script and APE is dynamically loaded and done with several layers of indirection. This means that 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.

## Compilation, activation, idle and run process - function reference

To give an complete idea of what happens while creating a function reference, let's do it chronologically. All of the mentioned function takes a pointer to a CProject struct. All of the functions must return a Status, identifying success. This struct contains information about all of the relevant files needed to compile the project, settings for the compiler, state of the code and lastly, space for the compiler to keep its relevant instance data. Specific information about all of this is found in the file *APESupport.h*

Once the codegenerator has loaded the compiler module and ensured all relevant functions exists, it calls the following function:

Status \_cdecl CompileProject(CProject \* p, void \* op, errorFunc\_t e);

The compiler must store the op parameter when it uses the callback error function e:

typedef void (\_cdecl \* errorFunc\_t)(void\*, const char \*);

op is to be used as the first parameter, the second being an error message back to APE. This callback function prints text to the associated console (identified through the op parameter). Example:

typedef void (\_cdecl \* errorFunc\_t)(void\*, const char \*);

struct my\_data

{

...

errorFunc\_t errorBack;

...

void \* op;

...

void printError(const char \* msg) {

if(errorBack && op)

errorBack(op, msg);

}

};

Status \_cdecl CompileProject(CProject \* p, void \* op, errorFunc\_t \* e) {

my\_data \* data = new my\_data;

data->op = op;

data->errorBack = e;

p->userData = data;

...

return Status::STATUS\_OK;

}

Status \_cdecl ActivateProject(CProject \* p) {

my\_data \* data = reinterpret\_cast<my\_data>(p->userData);

...

data->printError("Error activating project!");

return Status::STATUS\_ERROR;

}

The CompileProject() function is a request from APE to compile the current files in the CProject struct. The compiler is not supposed to link the code in this step.

If the return value of CompileProject() is Status::STATUS\_OK, APE later issues a call to InitProject().

Status \_cdecl InitProject(CProject \* p);

InitProject() finalizes the link time code generation and ensures the compiled code is executable. InitProject is assumed to return Status::STATUS\_OK. This marks the end of the compilation process.

After this, the compiled code is assumed to be in either an active or deactivated state, starting with deactivated. When the user tries to activate the project, the following function is called:

Status \_cdecl ActivateProject(CProject \* p);

This function is supposed to initiate any runtime libraries, constructors and whatnot the compiled code requires and ensure that the compiled code is (in)directly executable - this is what happens next. The compiler is allowed to run user-written code in this step. If this step is successful (the return shall be Status::STATUS\_READY), APE assumes that the script is activated and ready to process audio through the following function:

Status \_cdecl ProcessReplacing(CProject \* p, float \*\* in, float \*\* out, int sampleFrames);

The return of the function should be Status::STATUS\_OK. Any other return directly changes APE's internal status. For example, returning Status::STATUS\_DISABLED will initiate disabling process. Returning Status::STATUS\_ERROR will discard everything in the project to avoid compromising host stability. This is true for all of the mentioned functions here.

The purpose of this function is to give the script the ability to modify the audio stream how it seems fit, *however* it must respect that the output should be filled with values, modified or not. The size of the buffer is specified in sampleFrames. The amount of buffers is usually two (stereo), however changing settings in the configuration file can alter this amount. The amount of buffers can be retrieved through api.getNumInputs() and api.getNumOutputs().

While the plugin is in an activated state, APE may call the following function asynchronously:

Status \_cdecl OnEvent(CProject \* p, CEvent \* e);

This function is not required to be supported as it may be overkill for simple functions. The compiler is allowed to return Status::STATUS\_NOT\_IMPLEMENTED immediately. If it is to be supported, implementation details of CEvent can be found in *src/APE/APE/APESupport.h*. Use of events is described in *examples/events.h*. Supposed return value is either Status::STATUS\_OK or Status::STATUS\_HANDLED.

When the plugin is put in a deactivated state, the following function is called (synchronously, ie. never while an event is handled or while the plugin is processing):

Status \_cdecl DisableProject(CProject \* p);

This function ensures correct termination of the script and runtime, and puts the compile code in a deactivated state, from which it may be activated again through ActivateProject().

When the project is no longer used, the following function is called:

Status \_cdecl ReleaseProject(CProject \* p);

This function shall ensure correct cleanup of the compiled code and any associated resources the compiler has allocated. The module the compiler resides in may not be used again before it is unloaded, so it's important to release everything that won't be destroyed automatically in this function.

The following function are not yet required to be implemented, however they must exist:

Status \_cdecl AddSymbol(CProject \* p, const char \* name, const void \* mem);

Status \_cdecl GetState (CProject \* p);

void \* \_cdecl GetSymbol(CProject \* p, char \* s);

Status \_cdecl SetErrorFunc(CProject \* p, void \* op, errorFunc\_t e);

It is advised to return Status::STATUS\_NOT\_IMPLEMENTED from these functions - or NULL in case of pointers. To properly learn the interaction and process between these functions, study the provided compiler(s) and their associated scripts.

# Shipped compilers

APE currently ships with *Tcc4Ape*, an open-source compiler wrapper for APE based upon the open-source C compiler *TCC[[6]](#endnote-7)*. Also APE includes a wrapper called *syswrap*, that allows to interface to system compilers.

## Tcc4Ape

*Tcc4Ape* is the provided, standard compiler for the language C. Tcc4Ape, together with */includes/CInterface.h/* provides the foundation for language, which consists of C mixed with some preprocessor definitions for some syntactic sugar.

### Script structure

The program structure is built like an object (with good reasoning), where you have a constructor (onLoad), a destructor (onUnload), an optional event-handler (onEvent) and a processor function (processReplacing). The typical script looks like this:

#include <CInterface.h>

struct PluginData

{

};

GlobalData("template.c");

enum Status onLoad()

{

return status.ready;

}

enum Status onUnload()

{

return status.ok;

}

enum Status processReplacing(VstFloat \*\* in, VstFloat \*\* out, VstInt32 sampleFrames)

{

for(unsigned i = 0; i < sampleFrames; ++i)

{

out[left][i] = in[left][i];

out[right][i] = in[right][i];

}

return status.ok;

}

enum Status onEvent(struct eventInfo \* e)

{

return status.ok;

}

Nothing here should look too unfamiliar to those who have worked with audio-plugins before. The PluginData struct is where you declare all of your instance variables, as this will be passed to all of these functions. The specific instance is called \_this. Dereferenced alias is self:

#define self (\*\_this)

Along with \_this, another variable is passed: iface. iface holds pointers to all the API functions delivered by APE. Dereferenced alias is api:

#define api (\*iface)

### TCC API Function Reference

* float api.getSampleRate();

Returns the host's current sample rate as a float.

* int api.printLine(unsigned color, const char \* fmt, ...);

Prints a line to the associated console with the color given. Works like printf. Returns number of characters written. Color can be any of the following:

color.black

color.grey

color.blue

color.green

color.red

* int api.msgBox(const char \* txt, const char \* title, int style, int nBlocking);

Presents a classic message-box with the given text, title and style. Style is a bitmask which can be a combination of the following two groups (using the mbs (message-box-style) constant struct):

mbs.icon.stop // msgbox carries a stop icon

mbs.icon.question // msgbox carries a question mark

mbs.icon.info // msgbox carries a info icon

mbs.icon.warning // msgbox carries an exclamation point

mbs.style.ok // msgbox has an OK button

mbs.style.yesnocancel // msgbox has a yes, no and cancel button

mbs.style.contrycancel // msgbox has continue, try again and cancel button

nBlocking shall either be 0 or 1. If it is 1 (blocking), the message-box blocks and waits for the user to choose a button. If it is 0, the function returns immediately, returning an non-usable opaque value. The message-box is spawned in another thread. There can at most be 16 simultaneous message-boxes.

If the message-box is blocking, it will return either of these values, that usually represents the button pressed:

mbs.button.yes

mbs.button.no

mbs.button.retry

mbs.button.tryagain

mbs.button.con

mbs.button.cancel

Example usage:

auto ans = api.msgBox("Hi there!", "Hello!", mbs.icon.info | mbs.style.yesnocancel, true);

if(ans == mbs.button.yes)

doSomething();

else

doSomethingElse();

* Status api.setStatus(Status new);

Requests APE to change status to the new status. APE does not guarantee to do that, however it will return its internal state every time.

* int api.createKnob(const char \* name, float \* val, int type);

Creates a knob-control with the given name. All controls have a value between 0.0f and 1.0f, inclusive. The knob will update val whenever it changes its value (if it isn't null). The type can be any of the following:

knobType.percent // knob displays a value from 0 .. 100%

knobType.hertz // knob displays a value from 0 .. 8000 Hz

knobType.decibel // knob displays a value from -62.5 dB to 0 dB

knobType.fpoint // knob displays a value from 0.0 to 1.0

knobType.ms // knob displays a value from 0 to 1000 ms

Please note the display value has no influence on its internal value (0.0 .. 1.0). See *events* section for options on changing value and display text to arbitrary values. The return value is a tag that uniquely identifies that control.

* int api.createRangeKnob(const char \* name, const char \* unit, float \* val, scaleCB formatFunc, float \_min, float \_max);

Creates a knob-control with the given name. The display value is formatted using the callback function scaleCB, which allows to create a scale over the range:

typedef float (APE\_API\_VARI \* scaleCB)(float value, float \_min, float \_max);

value is the knobs internal value from 0 .. 1. \_min and \_max are the original parameters passed to createRangeKnob(). The knob-control will use the return of this function as the display value and suffix it with the unit parameter. However, as with the other knob-functions, val will contain the knobs internal value, from 0 .. 1.

The idea is to provide a custom range and apply a scale on it. For instance, when displaying a frequency scale in hertz, it would be logical to create a exponential scale (since human hearing is roughly logarithmic). You can provide your own callback function, but APE includes a couple:

scale.log()  
scale.exp()

scale.linear()  
scale.polyLog()  
scale.polyExp()

These can be passed directly to the function, like so:

auto tag = api.createRangeKnob("Frequency", "Hz", &self.value, scale.exp, 10, 1000);

You can also use these casually:

float scaledValue = scale.log(1, 10, 1000); // scaledValue is 1000

Please note that for scale.log and scale.exp, \_min has to be above zero (anything else wouldn't make mathematical sense). If you need this functionality, consider scale.polyLog or scale.polyExp, which provides fixed curves based on polynomials.

* timer api.timerGet();

Returns an opaque handle that represents a starting point in time.

* double api.timerDiff(timer start);

Returns the difference in time from a previous call to api.timerGet() in milliseconds.

* void \* api.alloc(size\_t size);

Allocates a block of zero-initialized memory of size size in bytes. Memory will be deallocated automatically when the your code is destructed. Do not release memory allocated through api.alloc() with anything else than api.free().

* void api.free(void \* ptr);

Frees a block of memory from a previous call to api.alloc(). Do not free memory from other sources with this function.

* int api.createKnobEx(const char \* name, float \* val, const char \* values, const char \* unit);

Creates a knob-control that will display its value using the |-seperated list given in values and suffix it with the given unit. Example usage:

const char \* values = "left|center|right";

auto tag = api.createKnobEx("pan", &self.pan, values, "pan");

Return value is a tag.

* void api.setInitialDelay(int samples);

Informs the host that the plugin delays the audio stream by samples amount. Useful when creating anything with lookahead. Note that the host may ignore this call, and is not obliged to perform delay compensation.

* int api.createLabel(const char \* name, const char \* fmt, ...);

Creates a label-control with the given name. Its display value depends on the format string given. The fmt string describes a printf-like syntax, but the additional parameters are pointers to the elements. Example:

char \* s;

int var;

float result;

auto tag = api.createLabel("Test result", "%d, %f, %s", &var, &result, &s);

while (true) {

var = rand() % 20;

if(var > 10) {

result = var \* M\_PI \* 2;

s = "succes";

}

else

{

result = 0;

s = "failure";

}

}

The label will then automatically display the values concurrently. Here are the supported format specifiers and what they correspond to:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| %x | %u | %i | %d | %f | %lf | %c | %s | %ld | %lu | %li |
| void \* | unsigned int | int | int | float | double | char | char \* | long double | unsigned long long | signed long long |

api.createLabel() does not support precision-specifiers or anything like that. Double %% yields a single %. Remember that doubles and floats are seperate specifiers, because normally in C floats are automatically promoted to doubles. This conversion does obviously not happen when you pass a pointer to a float, therefore the difference.

Note that, especially for strings, you must change the pointer atomically (eg. strcpy'ing directly on the pointer might lead to a crash). Also make sure the pointers are valid throughout your code's lifetime.

* int api.getNumInputs();

Returns the current number of input channels.

* int api.getNumOutputs();

Returns the current number of output channels.

* int api.createMeter(const char \* name, float \* val);

Creates a meter-control with the name name. This control shows a meter representing the val. Returns a tag.

* int api.createToggle(const char \* name, float \* val);

Creates a toggle control with the name name. Updates the value val points to with its internal value, if val isn't null. Return a tag.

* double api.getBPM();

Returns the host's tempo in beats per minute as a double.

* float api.getCtrlValue(int tag);

Returns the value of the control, which is uniquely identified by the tag.

* void api.setCtrlValue(int tag, float val);

Updates the control, identified by tag with the new value val.

* int api.createPlot(const char \* name, float \* valList, unsigned size);

Creates a plot of the values in the valList, that is, it interconnects all of the values into a graph. The values are interpreted as y values in a graph, with x being incremented by one each value. size is the amount of floats in the list.

### User-defined functions and instance pointers

To use api and self in your own function, they will have to accept these parameters:

int myFunc(struct PluginData \* this, struct CSharedInterface \* iface) {

self.timer = api.timerGet(); // works like normal  
 this->timer = iface->timerGet(); // can also do this

}

You call the function like so:

...  
myFunc(this, iface);  
...

### Macros and functions

log2(x)

Returns the base-2 logarithm.

isbadf(x)

Tests a float for +/- infinity, quiet and signaling NaN.

round(x)

Rounds the value to the nearest integer.

sgn(x)

Signum function, return -1, 0 or 1 depending on sign of function.

ArraySize(x)

Returns the size of the array (it has to be known at compile-time, no pointers!)

if\_sin(angle, result)

Calculates the sine to the angle and sets result to that value. Result should be an lvalue of floating-point type. This is an inline parabolic approximation of sin(x), that is around 15 times faster than standard library sin(x). The domain of this function is . Naturally, it is less precise. Beware that it is two lines of code, so this example is illegal:

if(something)  
 if\_sin(angle, result);

Do this instead:

if(something) {  
 if\_sin(angle, result);  
}

### 

float f\_fmod(float x)

Fast version of the library fmod() function.

### Types

typedef unsigned char bool;

Assumes a boolean value (true and false is also declared constants 1 and 0).

typedef long long timer;

Holds opaque timer values.

struct PluginData;

A user-defined struct which is passed to the four normal functions.

typedef int VstInt32;

An integer type compatible with VST-platforms.

typedef ... VstFloat;

A floating point type with a precision matching the host (eg. double for 64-bit signal paths).

## syswrap

syswrap acts as another layer of indirection. It will run a couple of shell scripts upon the various events described in the compiler api. It expects these scripts to create a compatible dynamic library in the same folder, that syswraps loads and installs in the signal chain. This means that you can use any compiler installed on your system for any language, and APE can run it directly.

### Scripts

There are three scripts located in the /compilers/syswrap/ directory. You will have to modify these scripts to your own system. The type of script will change depending on operating system, but here's an example for Windows (which runs batch files, \*.bat) and cl.exe, the Visual Studio C/C++ compiler.

#### environment script

The environment script is run when the syswrap instance is run the first time. It's intended use is to prepare the system/compiler/environment to compile commands, however it doesn't necessarily have to contain anything.

#### build script

The build script is run when the user wants to compile the code. The purpose of this script is therefore to turn the input files into a dynamic library. syswrap passes the expected output file name as the first commandline parameter, and a single string of input files, compatible with the current system's command line syntax, as the second parameter. The build script could look like this:

@echo off

call "C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\vcvarsall.bat"

cl.exe /I "C:\Audio\VstPlugins\APE\includes" /arch:SSE2 /Oi /O2 /Ot /GL /MT /fp:precise /Gd %2 /link /DEF:"C:\Audio\VstPlugins\APE\compilers\syswrap\syswrap.def" /DLL /OUT:%1

In this example, the command @echo off disables line debugging in the console to avoid verbose spam. It then runs the file vcvarsall.bat which sets up the environment for the visual studio compiler. Then it runs the visual studio compiler - here's a rundown of the arguments given to the compiler:

cl.exe ::

/I "C:\Audio\VstPlugins\APE\includes" -> specifies a include directory to search in

/arch:SSE2 -> specifies it should compile the program with SSE extensions

/Oi -> expands intrinsic functions that can speed up the program

/O2 -> enables optimization level 2  
/Ot -> favors fast code, but may create larger code  
/GL -> optimizes whole program  
/MT -> statically links the program with the C runtime  
/fp:precise -> floating point operations must be precice  
/Gd -> uses \_cdecl calling convention as default  
%2 -> passes all input files in second commandline argument  
/link -> following switches are for the linker  
/DEF:"C:\Audio\VstPlugins\APE\compilers\syswrap\syswrap.def" -> specify a .def file to generate correct symbols, see next section  
/DLL -> create a dynamic library  
/OUT:%1 -> specify output file name using first commandline argument

#### cleanup script

The cleanup script is run when syswrap is unloaded, and allows a clean up process (for instance, the system compiler may have created several intermediate files in the directory) where you can delete the generated files. syswrap passes the output file name (without extension!) as the first commandline argument. It could look like this:

@echo off

echo cleaning up files...

set "path1=%1%"

set "path2=%path1%.dll"

del "%path2%"

set "path2=%path1%.exp"

del "%path2%"

set "path2=%path1%.lib"

del "%path2%"

This will clean up all files the Visual Studio compiler created with the previous arguments.

### syswrap compiler API

After the scripts have run successfully, syswrap tries to load the dynamic library it specified to the build script. If successful, it expects the following function to be exported:

struct \_program\_info \* getProgramInfo()

This function returns a pointer to a filled \_program\_info struct, that determines how syswrap should interface with the program, and more importantly, who allocates the PluginData struct.

struct \_program\_info {

size\_t allocSize;

size\_t version;

const char \* name;

int selfAlloc;

void \* (\_cdecl \*palloc)(struct CSharedInterface \* iface);

void \* (\_cdecl \*pfree)(void \* pluginData);

};

As with TCC4Ape, the four normal functions (onEvent, processReplacing, onLoad, onUnload) first two arguments are a pointer to plugin-defined instance data (PluginData), and a pointer to the interface API. The PluginData struct is defined by the plugin itself and is the object, that represents this specific instance of the plugin. The plugin can specify its own allocators for this struct (it's completely opaque to syswrap/APE), or it can specify how large it is - then syswrap will allocate and free it, and zero-initialize it.

allocSize is the size of the PluginData struct. This field is ignored if selfAlloc is set. version is the version of the plugin. name is the name of the script/plugin. If selfAlloc is nonzero, syswrap will not allocate memory for the PluginData struct, but instead call palloc() and pfree() on creation/destruction to allow the library to create its own object (useful if PluginData is non-POD).

This process and information is only relevant if you do not use the included header "CInterface.h" - if you use "CInterface.h" all of this is done transparently, provided you follow the guidelines found in the Tcc4Ape API reference. It is done this way to allow complete compability with Tcc4Ape scripts (ie. you can run Tcc4APE scripts directly using syswrap), but also to provide a C interface that does not rely on a C-header (so you can use other languages).

The following four functions shouldn't be new (see Tcc4Ape API reference).

enum Status onLoad(void \* pluginData, struct CSharedInterface \* iface)  
-> onLoad

enum Status processReplacing(void \* pluginData, struct CSharedInterface \* iface, float \*\* in, float \*\* out, int sampleFrames)  
-> processReplacing

enum Status onEvent(void \* pluginData, struct CSharedInterface \* iface, struct eventInfo \* e)  
-> onEvent

enum Status onUnload(void \* pluginData, struct CSharedInterface \* iface)  
-> onUnload

Because of how syswrap expects these to be exported, it might be useful to specify to the compiler that these functions should be exported as-is. This can be done with Visual Studio compiler by using a module-definition-file (\*.def), and a compatible one would look like this:

EXPORTS

processReplacing=processReplacing

onLoad=onLoad

onUnload=onUnload

onEvent=onEvent

getProgramInfo=getProgramInfo

To specify that APE should use the syswrap compiler for certain languages, just edit the config file accordingly (see the section).

# Download links

* 0.5.0
  + <https://bitbucket.org/Mayae/ape/downloads/>
* 0.3.0
  + <http://jthorborg.com/content/ape/ape_release_0.3.0.zip>

# Changelog

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 21-04-2014
  + 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](https://www.jthorborg.com/cpl.html)
  + Extended platform support, including technical linux suport. See [requirements](#_Platform_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](http://clang.llvm.org/)
    - Runtime vehicle is [libCppJit](https://bitbucket.org/Mayae/cppjit): Multithreaded, lazy JIT based on [LLVM](https://llvm.org/)
    - 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](https://libcxx.llvm.org/)
    - Subset of C standard library available, based on [ccore](https://bitbucket.org/Mayae/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](https://www.jthorborg.com/signalizer.html)
    - Per-sample source code expression evaluation and graphing
    - Colour 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: 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 sideeffect 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: 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 it's 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 release.

# Licenses

See /licenses/.

# Credits and thanks

Thanks to the helpful community at *kvraudio[[7]](#endnote-8) ­­*- extremely helpful resource and excellent site.

Thanks to *stackoverflow[[8]](#endnote-9)* - always helping with design / coding issues

Thanks to *musicdsp[[9]](#endnote-10)* - 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[[10]](#endnote-11), libconfig, JUCE[[11]](#endnote-12) and Steinberg[[12]](#endnote-13) for delivering incredibly nice products that without doubt makes projects like mine realizable.

# Contact

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:

*dyanuzz@hotmail.com*

## Donations

I study full time and use most of my free time developing free, open-source tools like these. If you want to show your appreciation, you are very welcome to donate to the following PayPal:

*dyanuzz@hotmail.com*

If nothing else is noted, I will split the donations 50/50 to me and the other half to the rest of the free, open-source projects that made this project possible.

Thank you.

*Janus Lynggaard Thorborg*

*studying bachelor's degree in sonic communication and sound design*

*Sonic College, Haderslev in Denmark*

1. libconfig: <http://www.hyperrealm.com/libconfig/> [↑](#endnote-ref-2)
2. clang: <https://clang.llvm.org/> [↑](#endnote-ref-3)
3. libc++: <https://libcxx.llvm.org/> [↑](#endnote-ref-4)
4. libCppJit: <https://bitbucket.org/Mayae/cppjit/src/master/> [↑](#endnote-ref-5)
5. llvm: <http://llvm.org/> [↑](#endnote-ref-6)
6. tiny c compiler: <http://bellard.org/tcc/> [↑](#endnote-ref-7)
7. kvraudio: <http://www.kvraudio.com/> [↑](#endnote-ref-8)
8. stackoverflow: <http://stackoverflow.com/> [↑](#endnote-ref-9)
9. musicdsp: <http://www.musicdsp.org/> [↑](#endnote-ref-10)
10. vstgui: <http://sourceforge.net/projects/vstgui/> [↑](#endnote-ref-11)
11. juce: [http://www.juce.com](http://www.steinberg.net/) [↑](#endnote-ref-12)
12. steinberg: [http://www.steinberg.net](http://www.steinberg.net/) [↑](#endnote-ref-13)