Contents

Style Guide

- overview
- general notes
  - underscore separators
  - tab size
  - allman braces
  - using const reference arguments*
  - constant names
  - trailing underscores for private member variables
  - inline markdown comments
  - circular dependency avoidance
  - enum definitions*
- Misc.
- Example
  - daisysp module

* still being discussed, not strictly followed yet.

Overview

general notes

Naming

Type Names

Type names should begin with a capital letter, and have a capital letter at the start of each word. Compound words (i.e. Id, Callback, etc.) should only have a single capital letter.

Function Names

Function names should begin with a capital letter, and have a capital letter at the start of each word. Compound words (i.e. Id, Callback, etc.) should only have a single capital letter.

Variable Names

all variable names use underscores as a separator, and contain only lower case letters.

trailing underscores for private member variables

to prevent name clashes with function arguments and/or names, trailing underscores will be used for private member variables.

ex.

```
private:
float foo_;
```

Constant Names

consts, whether class members or not, should be prefixed with, 'k' and use capital letters at the beginning of each word.

For example, `float kConstantName = 1.542`

enums

This may still change

for now, we're putting enums outside the class with a `MODULENAME` prefix so that enums can be directly accessed without having to use the C++ syntax namespaces for them.

Enums are also currently not given type names to prevent issues when casting from standard types.

trailing commas are used at the end of every line of an enum definition including the last.

`MODULENAME_ENUM_LAST` is always used to have a valid "number of values" that can be used when dealing with the enum.

ex.

```
enum
{
    MODULENAME_STATE_FOO,
    MODULENAME_STATE_BAR,
    MODULENAME_STATE_LAST,
};
```

This can be accessed simply by `state = MODULENAME_STATE_FOO;`

alternatively we may change this so that it doesn't use a text-based namespace prefix, and move it within the class so that access would then be:

`state = ModuleName::STATE_FOO;`

This is still in discussion.

Giving enums named types is also a point of discussion, all though when moving between types it may not be very user friendly to have to:

```
// somewhere in init
uint8_t val = 0;

// somewhere being called periodically
if (button1.rising_edge())
{
    val = (val + 1) % OSC_WAVEFORM_LAST;
    osc.set_waveform(static_cast<osc_waveform>(val));
}
```

Misc

Tab Size

A tab = 4 spaces

allman braces

We use allman style braces for both libraries.

```
// So functions look like:
void do_something()
{
    foo = bar;
```

```
}

// instead of:
void do_something() {
    foo = bar;
}
```

this permits easier copy/paste when creating new modules, but also makes visual levels of indentation a little easier to see while working/reading code.

When used with if statements this also makes it very easy to comment out different logic statements, or even remove them for testing without having to worry about moving braces around.

using const references for arguments

```
inline float Process(const float &in) { return DoSomething(in * 0.5f); }
```

When an argument can be expected to come from a stored value (variable, etc.) a const reference should be used.

When it makes sense, pass values by reference, and to ensure that the implementation cannot change the value coming in, they should be passed in with the const qualifier. As a result any attempt to unintentionally change the input value will cause an error during compilation.

Previously we had this rule primarily for setters, but on simple parameters its common to enter a value directly (with no memory address), which is not possible when passing by reference.

Process() functions that operate on an input are a good example of when to use this.

This is a new rule, and there are several modules that don't follow it exactly right now.

inline markdown comments

Header files are parsed with a simple python script to generate reference documentation.

The parser checks for comments, and will generate markdown for any such lines.

There needs to be a blank comment line in between lines that require a line break in the output. Otherwise, consecutive lines will be treated as a single line in the markdown file.

ex.

```
// # Here's a new section
// With some text
//
```

```
// This will be on a newline,
// but this will be added to the previous.
//
// Put the following around code you wish to export into the documentation:
// ~~~~
    void some_function();
// ~~~~
```

circular dependency avoidance

We use both `#pragma once` and header guards to prevent circular includes, and similar issues that can arise without such a mechanism.

These should be used in all header files added to the libraries.

Naming convention for the header guards should be: `DSY_MODULENAME_H` where `MODULENAME` is the file/class name.

Example files

DaisySP module

Header:

```
// # Markdown Title
// Description
//
// Details

#pragma once
#ifndef DSY_MODULENAME_H
#define DSY_MODULENAME_H

namespace daisysp
{

enum state
{
    MODULENAME_STATE_A,
    MODULENAME_STATE_B,
    MODULENAME_LAST,
};

class ModuleName
{
    public:
    ModuleName () {}
    ~ModuleName () {}
```

```
    void Init();

    float Process(const float &in);

    inline void SetParam(const float &param) { param_ = param; }

    void SetComplexParam(const float &complex_param);

    private:

    float param_;
    float foo_bar_;
    float a_, b_;

};

} // namespace daisysp

#endif // DSY_MODULENAME_H
```

Implementation:

```
#include <system_include.h>
#include "modulename.h"

using namespace daisysp;

void ModuleName::Init()
{
    // Set private members to defaults
    param_ = 0.0f;
    a_ = 0.0f;
    b_ = 1.0f;
    // Do stuff
}

float ModuleName::Process(const float &in)
{
    // Do something and return the output.
    return (in * param_) + a_ - b_;
}

void ModuleName::SetComplexParam(const float &complex_param)
{
    a_ = complex_param;
    b_ = 1.0f - complex_param;
```

```
}
```