# Microcontroller USB Library in C++

https://github.com/TimB-QNA/ClassyUSB

Tim Brocklehurst

January 24, 2025

# Contents

# Chapter 1

# Introduction

The most pertinent question when writing a new library is why. For USB devices this is very simple. Existing libraries are highly complex, appallingly documented, and require a PhD in software engineering to add or remove components. That's ignoring the fact that many are tied to vendors, and as such the bloat is appalling, as every device has to be supported. Great for quickly throwing some code together, as a proof of concept, rubbish for actually building kit which is maintainable. For professional work, I wouldn't use the Arduino libraries, it's a good introduction to microcontrollers but it's probably not the solution you're looking for long-term.

This library attempts to fix some of those problems. There are always some tradeoffs, and its design favours ease of use over efficiency. Currently not many platforms are supported, but adding platform support is a matter of inheriting one class and writing the hardware support; everything else remains the same.

## 1.1 Supported Platforms

Only one platform is currently supported.

| Manufacturer | IC | Core Technology |
|---|---|---|
| Atmel (Microchip) | SAMD21 | ARM Cortex M0 |

Table 1.1: Supported microcontrollers

## 1.2 Limitations

There as some limitations with the way this library is built. In most cases they will be minor.

### 1.2.1 Devices are composite.

This isn't a major limitation, but it's something to be aware of as it means that the descriptors tend to be longer, and therefore more memory is required for the primary endpoint.

### 1.2.2 Hardware has limits

Microcontrollers have a limited number of endpoints, often arranged as endpoint pairs. At present the code does not try to use these pairs intelligently, so you only get half the endpoints you would expect. This will limit the number of components you can have.

This will be addressed at some point in the future.

### 1.2.3 Definitions for code tuning

The code can be tuned by means of the definitions below. This is a self-imposed limitation, but the ability to tune the memory usage may be very helpful.

- USB_MAX_COMPONENT_ENDPOINTS

  Default 3

  Maximum number of endpoints allowed in and component or subcomponent.

- PLATFORM_ENDIAN

  Default 0

  Platform endian. Little (0) or Big (1). No conversion is carried out if the platform endian matches USB.

- USB_MAX_COMPONENTS

  Default 3

  Maximum number of components in the device. Can be reduced to save memory.

- USB_MAX_ENDPOINTS

  Default 8

  Maximum number of endpoints handled. Can be reduced to save memory, may need to be increased for complex devices.

- USB_MAX_STRINGDESCRIPTORS

  Default 8

  Number of available string descriptors. Strings can use memory quickly so limit this where possible.

# Chapter 2

# Code Layout

The code laid out such that it is easy to add functionalilty and hardware support. The code is split into a number of classes which handle the hardware interface and device functions.

The main class is *usbDev*. This manages all the device-level operations and the device functionality. *usbDev* must be subclasssed to provide hardware support. The subclassed version is instantiated in the main program.

The library treats every configuration as a composite device. In USB parlance, a composite device has multiple components which can have different functionality. The library follows the naming conventions of USB components, which provide functionality. Components can be added using the "addComponent()" member up to the available number of hardware endpoints.

For example, serial over usb (CDC_ACM) on an Atmel SAMD21 might be configured thus:

```
#include "ClassyUSB/src/hardware/samd21usbDevice.h"
#include "ClassyUSB/src/CDC/usbCDC_ACM.h"

samd21usbDevice usb;
usbCDC_ACM acm;

int main(){
  ...
  usb.setVendorID(0x03EB);
  usb.setProductID(0x2404);

  usb.initialise();
  usb.addComponent( new usbCDC(&acm) );
  ...
}
```

The device descriptors for these components is handled automatically by the library; the user need only set the vendor and product ID; refer to USB.org for advice regarding the selection of these IDs.

Conceptually, the code is split into three blocks. Hardware, core, and functional. As the name suggests, the hardware block manages the hardware interface for whichever microcontroller you are using. It is, in effect, an abstraction layer. Core is really a management block, but it also takes care of tasks common to all devices, such as building descriptors to send to the host. The functional block provides the user-required functionality. You add functional blocks to the USB device with the *addComponent* method. You can add as many components as your physical endpoints allow.

# Chapter 3

# Bibliography