The Wonderful World of Designing a USB Stack Using Modern C++

MADELINE SCHNEIDER

In final semester of school at San Jose State University for a bachelor’s in Computer Engineering.

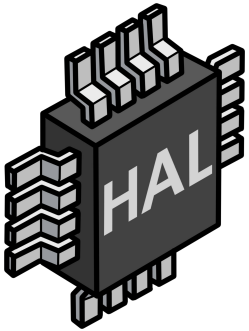


Table of Contents

Collaboration with Khalil Estell!

Intro

Motivation for this project

Motivation for a USB library

- A modular and flexible library to make USB device applications that doesn't infringe on unrelated hardware. To use components a la carte
- A solution in Modern C++ for *safety* and ergonomics
- A solution that is hardware agnostic without macros
- Use Exceptions for error handling

What makes a good library?

- Very subjective
- Well documented
- Abstractions make sense and are intuitive after reading docs

Designing a library for Hardware

Is sometimes referred to as a Hardware Abstraction Layer (HAL)

Designing Interfaces for Hardware

- By hardware we typically mean specific peripherals or modules on a microcontroller
- Generically, the only assumption that can be made are actions. Not state.

```
class output_pin {
public:
 void level(bool p_high) { driver_level(p_high); }
 [[nodiscard]] bool level() { return level(); }
private:
 virtual void driver_level(bool p_high) = 0;
 [[nodiscard]] virtual bool level() = 0;
};

class stm32f1_output_pin : public output_pin {
 void driver_level(
     bool p_high) override { /* impl to set pin on our
specific device */ }
 [[nodiscard]] bool
 driver_level() override { /*impl to read pin state on our
specific device */ }
};
```

https://godbolt.org/z/f5xWTb6fc

Why Dynamic Dispatch?

- Centralizes interfaces, creating a uniform definition
  - APIs can accept references or pointers to the abstract base class for centralized type management
- Easy to extend without breaking compatibility with older versions
  - In reality, our public facing APIs aren't virtual, they're wrappers for the virtual APIs
- Performance and size cost of a vtable is primarily negligible

```
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     eax, OFFSET FLAT:vtable for Derived+16
        mov     QWORD PTR [rbp-8], rax
        lea     rax, [rbp-8]
        mov     rdi, rax
        call    Derived::method()
        mov     eax, 0
        leave
        ret
```

Why not concepts?

- Harder to centralize, not as flexible
- No easy way to inject prologues or epilogues before implementations are called.
- Might be interesting to explore with reflections to emulate Rust Traits

Data management and ownership

Ownership - The concept of data belonging to a specific scope. Data (variables, etc) can have **one** owner*

```
void foo() {
 // my_arr is owned by the foo scope
 std::array<int, 3> my_arr{1, 2, 3};
}
```

Unowned data - References, spans, pointers, etc

```
void foo() {
 // my_arr is owned by the foo scope
 std::array<int, 3> my_arr{1, 2, 3};
 bar(std::span(my_arr));
}
// Bar has a view into my_arr via s
void bar(std::span<int> s) {
 // s is not owned by bar
}
```

Dynamic allocation in embedded software? 🤯

- Yes! Although we do not have a heap, that doesn't mean we cannot dynamically allocate things via PMR at initialization.
- Why? Lifetimes. Hardware interfaces should live the life of the program
  - Could mark them all as static and pass references

```
void blink(hal::output_pin& p) { p.level(p.level() ^ 1); }

// Main app loop (agnostic)
void application(resource_list r) { blink(*r.op); }

// Init platform(s) (typically in a separate file,
// Per platform
void initalize_platform() {
 // What if you forget static?
 static hal::my_device::output_pin op(
     /* Configuration Settings */);
   // ...
 application({&op, ...});
}
```

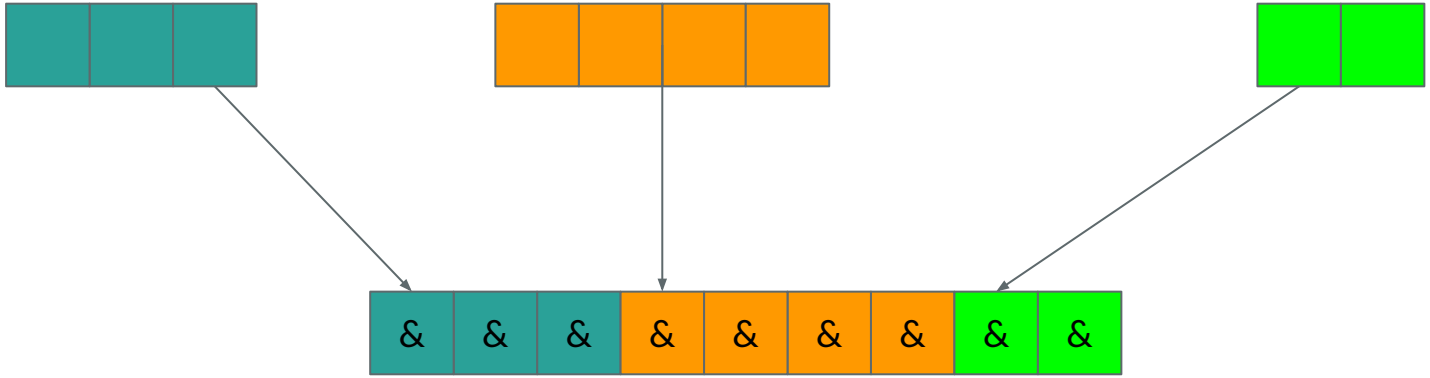
```
// Main app loop (agnostic)
void application() {
 std::pmr::polymorphic_allocator<> =
resources::driver_allocator();
 hal::strong_ptr<hal::output_pin> op_ptr =
     resources::acquire_output_pin(/* Settings
*/); // In another file, define resource
acquisition per platform

 blink(op_ptr); // RC goes up
 // Object is destroyed only when everything is
done using it.
 // Opens the door for concurrency as well!
}
```

strong_ptr

- Lightweight reference counting smart pointer
  - shared_ptr was a bit too heavy for our needs
- Is guaranteed to be non-null
  - shared_ptr can be null which is dangerous and/or inefficient for us
- *The resource strong_ptr points to is considered owned by the strong_ptr. Scopes do not own resources and only the smart pointer may manage a resource's memory.*

Scatter Span

- What is it? `using scatter_span = std::span<std::span<T>>;`
- Why? A way to have a single view of unowned data from multiple sources/buffers.

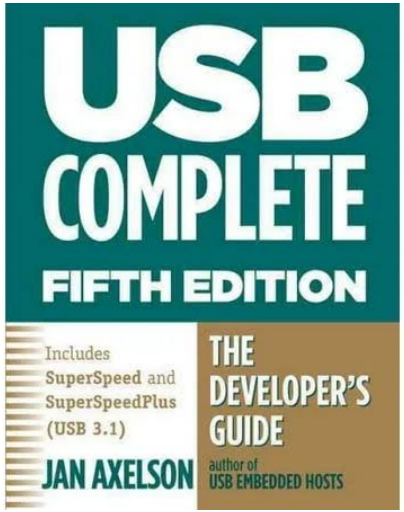


The basics of USB

Ok… Now the hard part

Great Resources for more information!

Beyond Logic



Some USB Terms

The Bus - The Universal Serial Bus, representing the data lines that have connected all USB systems together.

Host - The system driving all communication on the Bus, very commonly this is a computer. (The Controller)

Device - The peripheral, examples would be mice, keyboards, game controllers, hard drives, microphones, speakers, etc.

Endpoints - Places for USB systems to store data to be sent or received. Think mailboxes where letters are sorted into type of mail, if its outbound, or inbound.

Enumeration - The process of the host discovering the capabilities of the devices on bus.

Types of Communications - Transfers

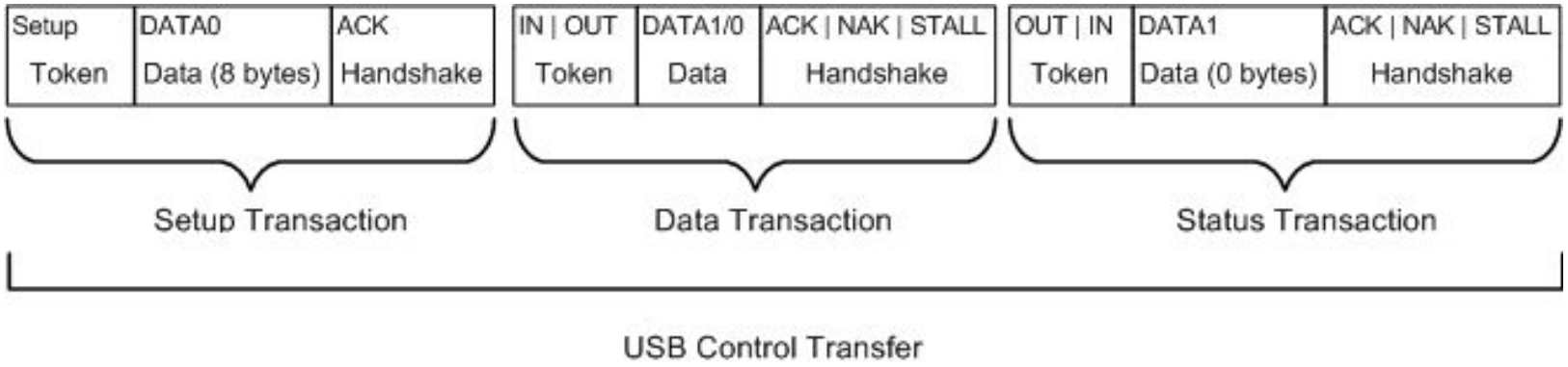
- **Control Transfers**
  - Used for setting up the device
  - Used by both parties to learn about each side's configurations and capabilities
  - The only transfer used during the enumeration process
- Interrupt Transfers
- Isochronous Transfers
- Bulk Transfers

Quick note: Control Transfer Transactions

Request payload (from host)

Data transmitted (depending on direction of token)

Receiver is indicated that data stream has ended and completes handshake

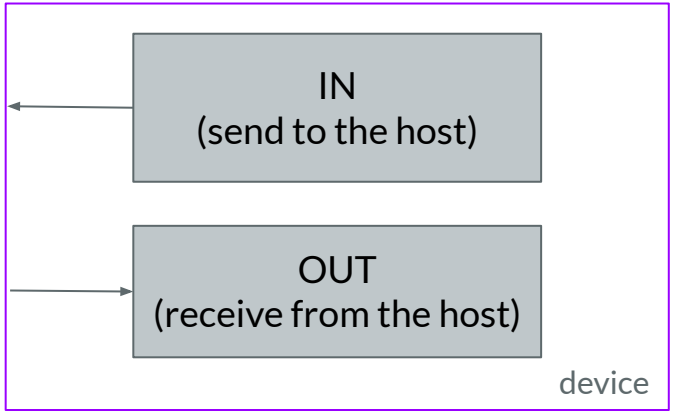


https://learn.microsoft.com/en-us/windows-hardware/drivers/usbcon/usb-control-transfer

Endpoints

- Endpoints can be thought of as small buffers that either emit or receive data (depending on their type).

- Four types of Endpoints:
  - **Control**
  - Interrupt
  - Isochronous
  - Bulk



How does the host learn about the device?

Enumeration

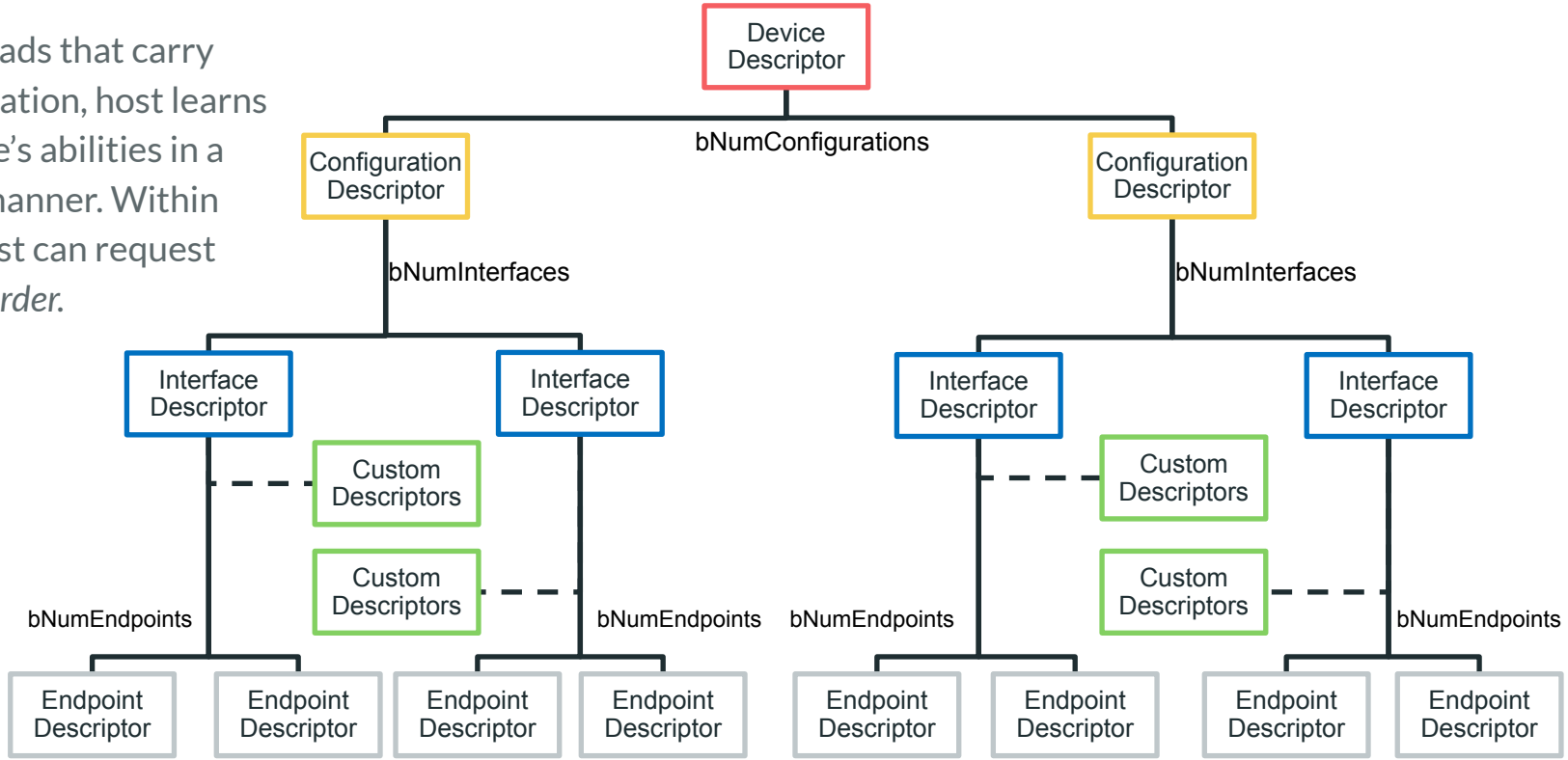
- The process the host performs to learn about devices on the bus
- Requests descriptors and configures devices (order may vary)
- Not a specific algorithm, based on device

Descriptors

- Data structures that describe USB device characteristics.
- Range from high-level specs (speed, power) to detailed communication formats

Descriptors Relationships

Specific payloads that carry device information, host learns about a device's abilities in a hierarchical manner. Within each layer, host can request nodes *in any order.*



https://www.beyondlogic.org/usbnutshell/usb5.shtml

Descriptor Example

The device descriptor payload layout in bytes.

In this case, the configuration descriptor.

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of Descriptor in Bytes
1	bDescriptorType	1	Constant	Configuration Descriptor (0x02)
2	wTotalLength	2	Number	Total length in bytes of data returned
4	bNumInterfaces	1	Number	Number of Interfaces
5	bConfigurationValue	1	Number	Value to use as an argument to select this configuration
6	iConfiguration	1	Index	Index of String Descriptor describing this configuration
7	bmAttributes	1	Bitmap	D7 Reserved, set to 1. (USB 1.0 Bus Powered) D6 Self Powered D5 Remote Wakeup D4..0 Reserved, set to 0.
8	bMaxPower	1	mA	Maximum Power Consumption in 2mA units

https://www.beyondlogic.org/usbnutshell/usb5.shtml#DeviceDescriptors

Let's enumerate!

Host

Device

Hello, I am connected, listening on address 0x00

Let’s enumerate!

ACK, your address is 0x07

Let’s enumerate!

Address: 0x07

ACK

Let's enumerate!

Address: 0x07

Send 0x12 bytes for your device descriptor

Let’s enumerate!

Address: 0x07

ACK, Here it is…

Let's enumerate!

Device: {
Manufacture String Index: 1
Product String Index: 2
Serial Number String Index: 3
Number of configurations: 1
…
}

Address: 0x07

…

ACK, Send 0x02 bytes for string descriptor 1, as locale en_US

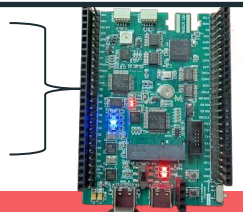
Let's enumerate!

Device: {
Manufacture String Index: 1
Product String Index: 2
Serial Number String Index: 3
Number of configurations: 1
…
}

Address: 0x07

…

ACK, Here is, the string is 0x0C bytes



Let's enumerate!

Device: {
Manufacture String Index: 1
Product String Index: 2
Serial Number String Index: 3
Number of configurations: 1
…
}

Address: 0x07

…

ACK, Send 0x0E bytes for string descriptor 1, as locale en_US

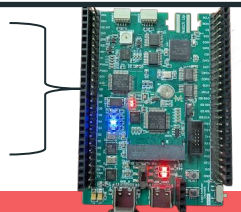
Let’s enumerate!

Device: {
Manufacture String Index: 1
Product String Index: 2
Serial Number String Index: 3
Number of configurations: 1
…
}

Address: 0x07

…

ACK, Here it is…,



Let's enumerate!

Device: {
Manufacture String Index: 1
Product String Index: 2
Serial Number String Index: 3
Number of configurations: 1
…
}

Strings: {
Manufacture String: "Company, Inc"
}

Address: 0x07

…

ACK, Send 0x09 bytes for configuration descriptor 1.

Let’s enumerate!

Device: {
Manufacture String Index: 1
Product String Index: 2
Serial Number String Index: 3
Number of configurations: 1
…
}

Strings: {
Manufacture String: “Company, Inc”
}

Address: 0x07

…

ACK, Here it is…

Let's enumerate!

Device: {
Manufacture String Index: 1
Product String Index: 2
Serial Number String Index: 3
Number of configurations: 1
…
}

Strings: {
Manufacture String: "Company, Inc"
}

Config {
Index: 1,
Number of Interfaces: 1
Total Length: 0xXX
…
}

Address: 0x07

…

ACK, Now send 0xXX bytes for all descriptors associated with configuration 1

Let's enumerate!

Device: {
Manufacture String Index: 1
Product String Index: 2
Serial Number String Index: 3
Number of configurations: 1
…
}

Strings: {
Manufacture String:
"Company, Inc"
}

Config {
Index: 1,
Number of Interfaces: 1
Total Length: 0xXX
…
}

Address: 0x07

…

ACK, Here is…

Let's enumerate!

Device: {
Manufacture String Index: 1
Product String Index: 2
Serial Number String Index: 3
Number of configurations: 1
…
}

Strings: {
Manufacture String:
"Company, Inc"
}

Config {
Index: 1,
Number of Interfaces: 1
Total Length: 0xXX
…
}

Iface {
…
};

…

Endpoints {
…
}

Address: 0x07
…

ACK, Set configuration 1

Let's enumerate!

Device: {
Manufacture String Index: 1
Product String Index: 2
Serial Number String Index: 3
Number of configurations: 1
…
}

Strings: {
Manufacture String:
"Company, Inc"
}

Config {
Index: 1,
Number of Interfaces: 1
Total Length: 0xXX
…
}

Iface {
…
};

…

Endpoints {
…
}

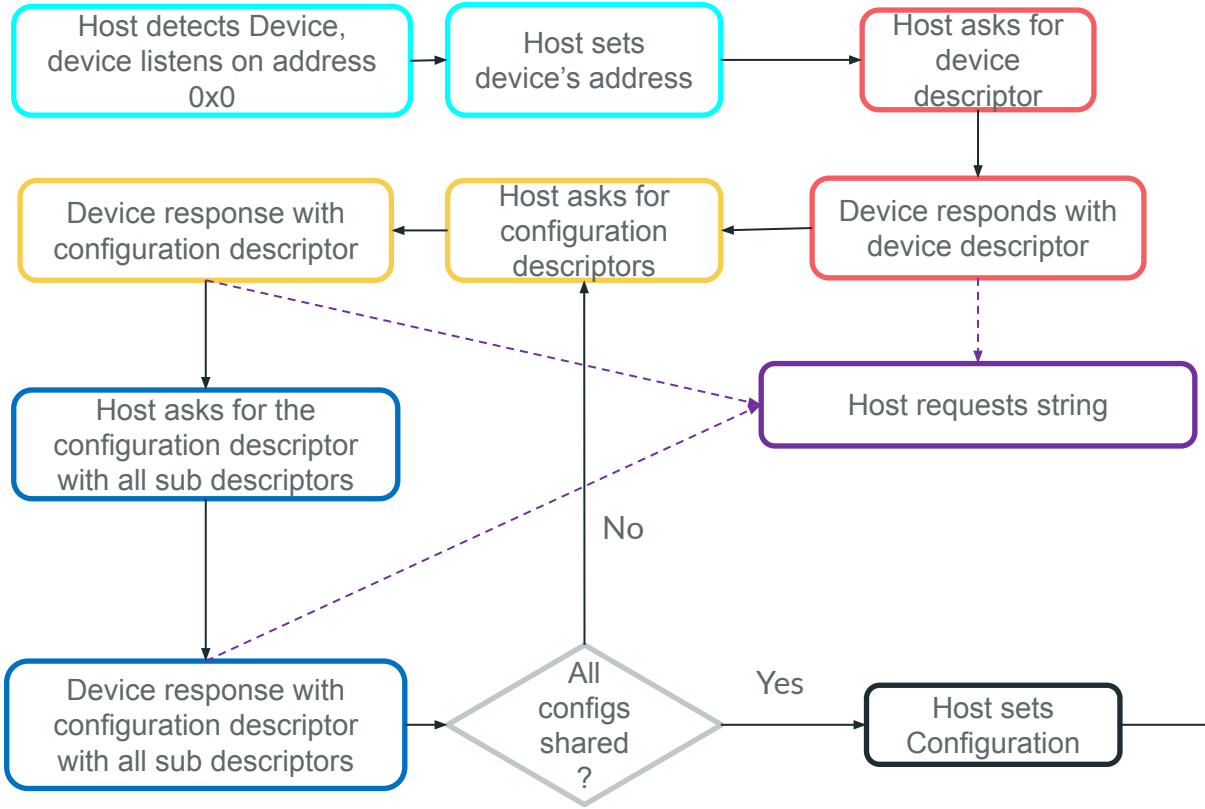
Address: 0x07

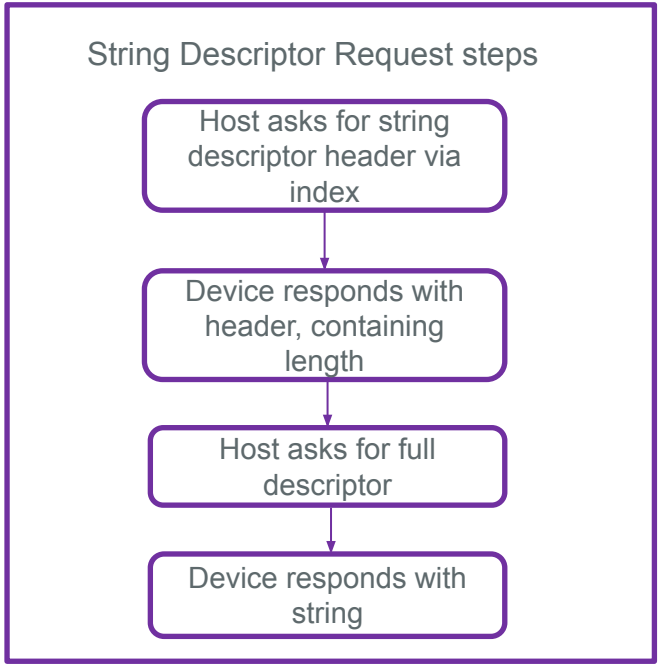
Active configuration 1

…

ACK, Here is…

Summary of (oversimplified) enumeration





All functionality outside of base USB is its own standard.

Got all that? No?
No worries, This isn’t a USB protocol talk

Library Design

Where do you even start?

Components

- The first question: What exactly is needed for a USB library, abstractly
  - Endpoints
  - Descriptor management
  - Organizational system for specific USB functions
  - Enumeration

- The next question: Which of the components actually interact with hardware?

Just the Endpoints!

Endpoint APIs

- Base endpoint class All endpoints can…
  - Get info about themselves (number, if they are stalled and max buffer size)
  - Go into a STALL state
  - Or reset themselves

```
class endpoint {
 [[nodiscard]] virtual endpoint_info driver_info() = 0;
 virtual void stall(bool p_should_stall) = 0;
 virtual void driver_reset() = 0;
};
```

```
struct endpoint_info {
 u16 size;
 u8 number;
 bool stalled;
 bool in_direction() { return number >> 7;
}
 u8 logical_number() { return number & 0xF;
}
};
```

Writing

- Writing is relatively simple, give it a scatter span it will write the payload onto the bus
- Packets are written lazily, so until the max size has been reached or a manual flush is done (by sending a Zero Length Packet)
  - Why? Dictated by the standard

```
class in_endpoint : public endpoint {
 virtual void driver_write(scatter_span<byte const> p_data) = 0;
};
```

Reading with Interrupts

- Reading is interrupt based!
- on_receive is triggered when there is new data in the endpoint to be read

```
class out_endpoint : public endpoint {
 struct on_receive_tag {};
 virtual void
 driver_on_receive(callback<void(on_receive_tag)> const& p_callback) = 0;
 virtual usize driver_read(scatter_span<byte> p_buffer) = 0;
};
```

Endpoint Type APIs

Bulk Endpoints

```
struct bulk_in_endpoint : public in_endpoint {};
struct bulk_out_endpoint : public out_endpoint {};
```

Interrupt Endpoints

```
struct interrupt_in_endpoint  : public in_endpoint {};
struct interrupt_out_endpoint  : public out_endpoint {};
```

Isochronous Endpoints

Coming Soon!

Control Endpoint

- The only bi-directional endpoint
- Has the additional responsibility of managing the USB control module
  - connect, set_address
- **NOT** to be used by interfaces/drivers directly but by the enumerator

```
class control_endpoint : public endpoint {
 virtual void driver_connect(bool p_should_connect) = 0;
 virtual void driver_set_address(u8 p_address) = 0;
 virtual void driver_write(scatter_span<byte const> p_data) = 0;
 virtual usize driver_read(scatter_span<byte> p_buffer) = 0;
 virtual void
 driver_on_receive(callback<void(on_receive_tag)> const& p_callback) = 0;
};
```

Technically that's all you need!

You can manually enumerate by sending hand crafted descriptors (just LE byte arrays)

And handle any requests and functionality by just utilizing the right endpoints

That's a lot of moving parts for a single peripheral; wouldn't it be nice to have a system that manages this for you…

Detour: Inversion of Control (IoC)

- The caller gives the callee temporary control over one of its resources typically via a capturing lambda. The callee can decide when to invoke the callback.

```
void caller() {
 resource r;
 auto f = [&r](scatter_span<byte> buf) { r.write(buf); };
 callee(f); // Temporarily gives callee control of r
}

void callee(std::function<void(scatter_span<byte>)> const&
p_callback) {
 // ...
 // Takes in multiple buffers and returns an array of spans
 auto payload = make_scatter_array<u8>(/* sources */);
 p_callback(payload); // Use r, write to the resource
 // ...
}
```

Descriptor Management

- Most descriptors are ephemeral and are only needed during enumeration.
- Many descriptors can be evaluated lazily as their fields fetch from other resources such as endpoints or are calculated by the enumerator.
- For Example:

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of Descriptor in Bytes
1	bDescriptorType	1	Constant	Configuration Descriptor (0x02)
2	wTotalLength	2	Number	Total length in bytes of data returned

https://www.beyondlogic.org/usbnutshell/usb5.shtml#ConfigurationDescriptors

Putting it all together: The USB Interface

- A class to hold everything related to a given USB functionality (endpoints, strings, assets for facilitating functionality of a given interface)

- Example: Microphones, keyboards, storage devices, serial communication, etc

```
struct descriptor_count {
  u8 interface;
  u8 string;
};

struct descriptor_start {
  std::optional<u8> interface;
  std::optional<u8> string;
};
```

```
class interface {
 using endpoint_writer =
callback<void(scatter_span<byte const>)>;

 [[nodiscard]] virtual descriptor_count
 driver_write_descriptors(descriptor_start p_start,
endpoint_writer const& p_callback) = 0;

 [[nodiscard]] virtual bool
 driver_write_string_descriptor(u8 p_index, endpoint_writer
const& p_callback) = 0;

 virtual bool driver_handle_request(setup_packet const&
p_setup, endpoint_writer const& p_callback) = 0;
};
```

write_descriptor

```
[[nodiscard]] descriptor_count write_descriptors(descriptor_start p_start,
    endpoint_writer const& p_callback) override {
  // set iface index and string indexes
  p_callback(/* aggregated descriptors */);
  return descriptor_count {
    .string = /* number of indexes used */,
    .interface = /* number of interfaces */
  }
```

```
struct descriptor_count {
  u8 interface;
  u8 string;
};

struct descriptor_start {
  std::optional<u8> interface;
  std::optional<u8> string;
};
```

write_string_descriptor

```
[[nodiscard]] bool
driver_write_string_descriptor (u8 p_index,
                                endpoint_writer const &p_callback) override {
 // logic to find string
 if (found) {
   p_callback(/* string descriptor for p_index */ );
   return true;
 }
 return false;
}
```

An example implementation

- CDC = communications device class (serial communication, etc)
- Used for making a virtual COM/serial port.

```
class cdc_interface : public interface {
...
 struct iface_desc { /* fields */};
 struct cdc_desc { /* fields */};
strong_ptr<bulk_out_endpoint> m_ep_data_out;
 strong_ptr<bulk_in_endpoint> m_ep_data_in;
 strong_ptr<interrupt_out_endpoint> m_ep_status_out;

cdc_interface(strong_ptr<bulk_out_endpoint>,
strong_ptr<bulk_in_endpoint>,
strong_ptr<interrupt_out_endpoint>,
strong_ptr<interrupt_in_endpoint>, ...) {/* impl */}
...
};
```

What about device and configuration descriptors?

- Data structure utilities are provided for constructing and managing these
  - They do not fall within any of the other component's domain
- Can also be managed without first party structure representation
- Data structures are basically just a wrapper over an array and span conversion method.

```
// Small snippet from usb::device
operator std::span<byte const>() const { return m_packed_arr; }
constexpr u8& serial_number_index() { return m_packed_arr[14]; }
constexpr u8& num_configurations() { return m_packed_arr[15]; }
std::array<hal::byte, 16> m_packed_arr;
```

Configuration Structure

```
template<size_t N>
class configuration {
   // ...
   std::array<interface, N> m_ifaces;
};
```

```
 configuration<1> pri_conf;
 configuration<2> alt_conf;
 enumerator en(..., std::array<configuration<?>>{pri_conf, alt_conf}, ...);
```

Configuration Structure

```
template<typename T>
concept usb_interface_concept = std::derived_from<T, interface>;

template <usb_interface_concept... Interfaces>
constexpr configuration(std::string_view p_name, bitmap&& p_attributes, u8 p_max_power,
std::pmr::polymorphic_allocator<> p_allocator, strong_ptr<Interfaces>... p_interfaces)
   : name(p_name), interfaces(p_allocator) {
 interfaces.reserve(sizeof...(p_interfaces));
 (interfaces.push_back(p_interfaces), ...);
```

The Enumerator Design

- Responsible for device and configuration wide state management
- Two phases
  - Realization of descriptors
  - Descriptor writing state machine

```
template <size_t num_configs>
class enumerator {
 enumerator(
     strong_ptr<control_endpoint> const& p_ctrl_ep,
     strong_ptr<device> const& p_device,
     strong_ptr<std::array<configuration, num_configs>> const& p_configs,
     std::span<u16> p_lang_strs,
     u8 p_starting_str_idx) { /* impl */ };
...
};
```

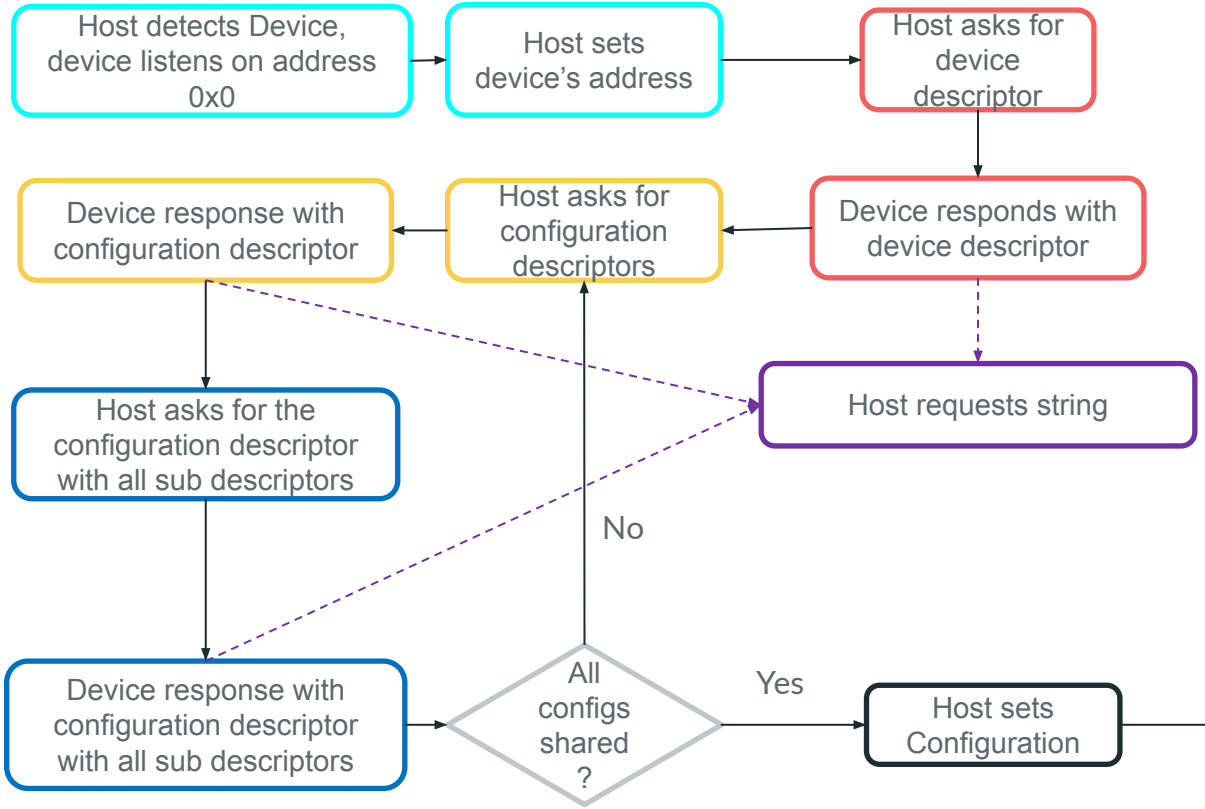
Realizing Descriptors

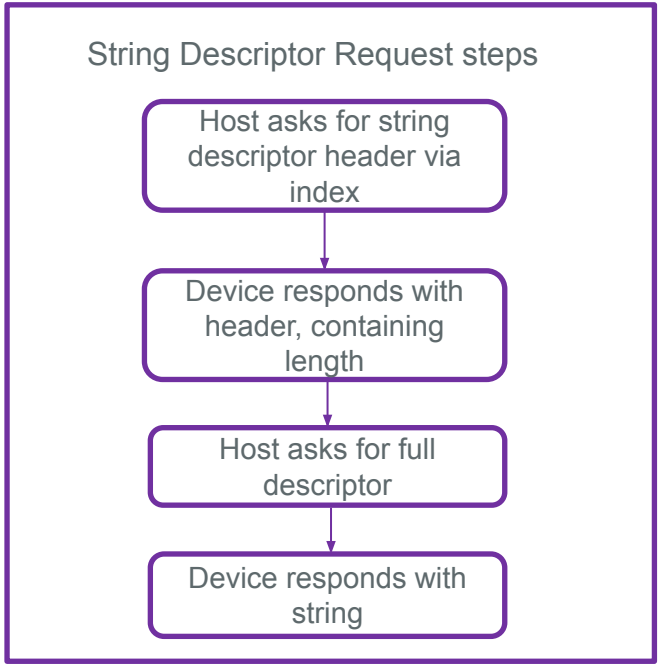
```
// After device and counting configuration indexes and strings
for (configuration& config : *m_configs) {
     auto total_length = static_cast<u16>(constants::config_desc_size);
     for (auto const& iface : config.interfaces) {
       interface::descriptor_count deltas = iface->write_descriptors(
         { .interface = cur_iface_idx, .string = cur_str_idx },
         [&total_length](scatter_span<hal::byte const> p_data) {
           total_length += scatter_span_size(p_data);
         });
       cur_iface_idx += deltas.interface;
       cur_str_idx += deltas.string;
     }
     config.num_interfaces() = cur_iface_idx;
     config.total_length() = total_length;
   }
...
```

Writing descriptors

```
m_ctrl_ep->on_receive([&waiting_for_data](on_receive_tag) { waiting_for_data = false; });
m_ctrl_ep->connect(true);
do {
   while (waiting_for_data) {
      continue;
   }
waiting_for_data = true;
// snip
setup_packet req = pkt_from_span(raw_bytes);
if (req.get_recipient() != setup_packet::recipient::device) {
      throw hal::not_connected(this);
}
handle_standard_device_request(req);
if (static_cast<standard_request_types>(req.request) ==
   standard_request_types::set_configuration) {
 finished_enumeration = true;
 m_ctrl_ep->on_receive([this](on_receive_tag) { m_has_setup_packet = true; });
}

}while (!finished_enumeration);
```

handle_standard_device_request





Questions?