FreeBSD Journal
Production Deployments

USB in User Space

A short introduction to USB tooling and how to write your own custom USB tools or userland drivers

Rick Parrish

This article starts with an overview of the USB tooling that ships with FreeBSD. For developers, the article goes on to describe the libusb20(3) access library. This library provides a developer friendly way to interact with the generic USB kernel module without stuffing bytes into ioctl(2) calls.

USB tools on FreeBSD

Here's an overview of some of the tools available without the need to write any code — with handbook references by chapter. For a general overview of USB on FreeBSD, take a look at **chapter 13 of the arch Handbook**.

USB Storage and OTG
Storage (chapter 21) is a whole other beast that I won't discuss here. Ditto for OTG or **gadget mode (chapter 29)**.

USB HID (keyboard and mouse)
The tool **usbhidaction** can monitor the HID interface looking for key presses from the remote. A typical use-case is to configure your keyboard so the special "Fn" keys cause actions that match the OEM silkscreen. Actions like

mute, increase or decrease volume, screen brightness, turn wifi on or off and such. As a companion to usbhidaction, look at **usbhidctl**.

USB Tethering (**chapter 35**)
Any time I need laptop internet access but WiFi isn't cooperating, I can fall back to USB tethering to the internet data provided by a smart phone. Connect smart phone to laptop. Turn on USB tethering in the phone. Inspecting dmesg(8) should show a new device 'ue0'. The only trick is to run dhclient on the new interface. It's quite painless — run 'dhclient ue0' as root. Here's a devd(8) rule to make it automatic.

```
notify 20 {
    match "system" "ETHERNET";
    match "type" "IFATTACH";
    match "subsystem" "ue[0-9]+";
    action "/sbin/dhclient $subsystem";
};
```

ls(1)
In case you weren't aware, USB device nodes appear under /dev/ugenX.Y and under /dev/usb/X.Y.Z where X, Y and Z are decimal digits. I think ls(1) is most useful for examining group and user access for a node. The /dev/ugenX.Y device nodes refer to a whole USB device. Example:

```
$ ls -la /dev/ugen*
lrwxr-xr-x 1 root wheel 9 Jul 27 19:59 /dev/ugen0.1 -> usb/0.1.0
lrwxr-xr-x 1 root wheel 9 Jul 27 19:59 /dev/ugen0.2 -> usb/0.2.0
lrwxr-xr-x 1 root wheel 9 Jul 27 19:59 /dev/ugen0.3 -> usb/0.3.0
lrwxr-xr-x 1 root wheel 9 Jul 27 19:59 /dev/ugen1.1 -> usb/1.1.0
lrwxr-xr-x 1 root wheel 9 Jul 27 19:59 /dev/ugen1.2 -> usb/1.2.0
lrwxr-xr-x 1 root wheel 9 Jul 27 19:59 /dev/ugen1.3 -> usb/1.3.0
lrwxr-xr-x 1 root wheel 9 Jul 27 19:59 /dev/ugen1.4 -> usb/1.4.0
lrwxr-xr-x 1 root wheel 9 Jul 27 19:59 /dev/ugen2.1 -> usb/2.1.0
lrwxr-xr-x 1 root wheel 9 Jul 27 19:59 /dev/ugen2.2 -> usb/2.2.0
```

Notice the /dev/ugenX.Y device nodes are all symbolic links to /dev/usb/X.Y.0 device nodes (eg. device nodes where the third digit is always zero). Here is an example listing of /dev/usb/X.Y.Z device nodes:

```
$ ls -la /dev/usb
dr-xr-xr-x 2 root wheel 512 Jul 27 19:59 .
dr-xr-xr-x 13 root wheel 512 Jul 27 14:59 ..
crw------- 1 root operator 0x2f Jul 27 19:59 0.1.0
crw------- 1 root operator 0x45 Jul 27 19:59 0.1.1
crw------- 1 root operator 0x80 Jul 27 19:59 0.2.0
crw------- 1 root operator 0x84 Jul 27 19:59 0.2.4
crw------- 1 root operator 0x85 Jul 27 19:59 0.2.5
crw------- 1 root operator 0x86 Jul 27 19:59 0.2.6
crw------- 1 root operator 0x87 Jul 27 19:59 0.2.7
crw------- 1 root operator 0x88 Jul 27 19:59 0.2.8
crw------- 1 root operator 0x89 Jul 27 19:59 0.2.9
crw------- 1 root operator 0x8e Jul 27 19:59 0.3.0
crw------- 1 root operator 0x90 Jul 27 19:59 0.3.1
crw------- 1 root operator 0x91 Jul 27 19:59 0.3.2
```

... clipped for space ...

```
crw------- 1 root operator 0x33 Jul 27 19:59 2.1.0
crw------- 1 root operator 0x43 Jul 27 19:59 2.1.1
crw------- 1 root operator 0x82 Jul 27 19:59 2.2.0
crw------- 1 root operator 0x8a Jul 27 19:59 2.2.1
```

Many of these /dev/usb device nodes do not end with a zero. The nodes ending in a non-zero are additional endpoint / interface combinations offered by the same device. In the above example under /dev/usb, 2.1.0 and 2.1.1 share the same device. Likewise, 2.2.0 and 2.2.1 are the same device.

dmesg and usbconfig

Run usbconfig(8) to see your connected USB devices. The output is much more interesting if run from root. When run as an ordinary user, it will tell you which devices are accessible. If a desired device isn't shown, you might need

to create a devd rule to grant yourself access when the device is attached. A Linux equivalent might be lsusb.

dmesg(8) will log USB device attach and detach events. Shortly after plugging in a device, you can examine the tail of dmesg(8) to see what device node was assigned. Here's an example where I unplugged and then re-inserted two different USB adapters.

```
ugen0.2: <Realtek 802.11ac WLAN Adapter> at usbus0 (disconnected)
ugen0.2: <Realtek 802.11ac WLAN Adapter> at usbus0
ugen0.5: <Logitech USB Receiver> at usbus0 (disconnected)
ugen0.5: <Logitech USB Receiver> at usbus0
```

Most USB devices are supported by FreeBSD kernel modules that map host level operations to something the USB connected device understands. Examples are reading mouse button clicks or passing IP packets to/from a WiFi adapter.

Some devices are exotic enough that a kernel module isn't available or the device manufacturer instead provides a userland library that avoids the need for a kernel module.

Today, the most portable — and most popular – way to directly talk to a USB device from a user process is through the libusb(3) access library which is available on numerous OSes. FreeBSD includes its own implementation.

Under the hood of the FreeBSD implementation of the libusb(3) API is a native FreeBSD access library API that offers some advantages over the libusb API. This article looks at the native access library API unique to FreeBSD.

In the text that follows, we'll look at the native USB API — libusb20(3) — and write our own implementation of usbconfig(8) / lsusb using strictly FreeBSD native bits. We need no new kernel modules outside of what comes standard

with FreeBSD. The USB API provides everything we need to locate a desired USB device, open it like you might open a file or tty device and send commands to control the device.

The first task in talking to any USB device is to identify the device from the list of currently connected USB devices. At a shell prompt, you can run 'usbconfig list' from a login with root level access to see what's there. Example output:

```
ugen0.1: <XHCI root HUB Intel> at usbus0, cfg=0 md=HOST spd=SUPER
      (5.0Gbps) pwr=SAVE (0mA)
ugen1.1: <EHCI root HUB Intel> at usbus1, cfg=0 md=HOST spd=HIGH
      (480Mbps) pwr=SAVE (0mA)
ugen2.1: <EHCI root HUB Intel> at usbus2, cfg=0 md=HOST spd=HIGH
      (480Mbps) pwr=SAVE (0mA)
ugen0.2: <AC600 wireless Realtek RTL8811AU [Archer T2U Nano] TP-Link>
      at usbus0, cfg=0 md=HOST spd=HIGH (480Mbps) pwr=ON (500mA)
ugen2.2: <Integrated Rate Matching Hub Intel Corp.> at usbus2, cfg=0
      md=HOST spd=HIGH (480Mbps) pwr=SAVE (0mA)
ugen1.2: <Integrated Rate Matching Hub Intel Corp.> at usbus1, cfg=0
      md=HOST spd=HIGH (480Mbps) pwr=SAVE (0mA)
ugen0.3: <Nano Receiver Logitech, Inc.> at usbus0, cfg=0 md=HOST
      spd=FULL (12Mbps) pwr=ON (98mA)
```

USB devices are mostly identified by a pair of 16 bit identities — a vendor ID or VID and a product ID or PID. To see the VID and PID of each device, run 'usbconfig dump_device_desc'. VID and PID are normally displayed as four digit hexadecimal values. If you have two or more devices with the same VID and PID, they can often be distinguished by a serial number. In the USB world, a serial number is an alphanumeric string (not an integer). It is not limited to decimal digits. If you have two or more devices with the same VID, PID, and serial number, they might still be distinguishable by USB bus.
A program coded to talk to a specific device can enumerate all devices looking for candidates with a desired VID and PID. The program can present the list of matches to the user or maybe pick the first one.

Interfaces and Configurations

USB devices present one or more interfaces. This mostly allows one USB device to serve multiple functions. When dealing with bulk and isochronous transfers, you'll need to specify which interface. The usbconfig(8) utility can list all interfaces advertised by a given device.

```
usbconfig -d ugen0.1 dump_curr_config_desc
```

For this article, we'll stick to the default configuration but some devices support multiple configurations. usbconfig(8) can reveal all configurations.

```
usbconfig -d ugen0.1 dump_all_config_desc
```

Notice that I'm not giving much detail on interfaces and configurations. Their meaning and number varies from one device to the next. There are too many possibilities to delve into them here. Just know they exist.

devd(8)

The point of this article is to write user — not kernel — code to access the device without the need for elevated (root) access.

A surgically placed chmod(1) or chown(1) on the device node removes the need for root.

A small devd(8) rule makes that step automatic.

One cheat starts with the device in question unplugged. Plug it into an open USB slot, and examine dmesg(8) to see what device node was assigned to it. Given the VID and PID, a simple devd rule adjusts permissions for you each time the device is re-attached to the host. Use ls(1) to verify ugen device node permissions.

As an aid, we can create one devd(8) rule whose only purpose is logging.

```
notify 100 {
       match "system"          "USB";
       match "subsystem"       "DEVICE";
       match "type"            "ATTACH";
       match "vendor"          "0x1234";
       match "product"         "0x5678";
       action "logger $cdev device ";
};
```

The *match "vendor"* clause matches a four digit VID. The 0x prefix tells us the value is hexadecimal. The *match "product"* clauses matches a four digit PID, also with a 0x hexadecimal prefix. The above snippet logs the device node when the device is attached. Seeing such a log message confirms the rule matching clauses.

USB devices generally support multiple interfaces. If we're able to successfully log the device as attached, we can add a rule that acts on the interface(s) to modify group and user permissions. You should manually try the commands before including them in a devd(8) rule. chown(1) can assign both group and user like so:

```
chown ordinary:wheel /dev/ugen1.2
chown ordinary:wheel /dev/usb/1.2.*
```

This assumes the desired outcome is for user 'ordinary' who happens to be a member of the 'wheel' group. Edit this to suit your needs. The actual devd(8) rule action clause is a little more complicated as I want to touch the /dev/ugenX.Y node and also the associated /dev/usbX.Y.Z nodes. Here's the whole snippet:

```
notify 100 {
       match "system"          "USB";
       match "subsystem"       "INTERFACE";
       match "type"            "ATTACH";
```

```
        match "vendor"          "0x1234";
        match "product"         "0x5678";
        action "cdev=$cdev; usb=$(expr ${cdev} : 'ugen\([0-
9]\.[0-9]\)'); chown -h ordinary:wheel /dev/${cdev}; chown -h
ordinary:wheel /dev/usb/${usb}.*";
};
```

Be sure to use ls(1) to verify ownership was applied correctly.

usbipd(8)

The usbipd(8) daemon and companion usbip(8) command allow connecting to a USB device on a remote host. This tool is compatible with a well-known USB-over-IP implementation on Linux. Your FreeBSD host can be client or server - or both.

I tested this over localhost by picking a device that appears as ugen0.3 on my host. Here are the steps with results.

```
git clone https://github.com/hackermaskee/freebsd-usbip
cd freebsd-usbip
make kmod
make
[sudo|doas|mdo]make install

[sudo|doas|mdo]vi /etc/rc.conf
usbipd_devices="ugen0.3"
usbipd_enable="YES"

[sudo|doas|mdo]kldload vhci
[sudo|doas|mdo]service usbipd start
[sudo|doas|mdo]usbip attach -r loocalhost -b ugen0.3
```

At this point, usbconfig lists the device twice like so:

```
$ usbconfig

ugen3.2: <RTL2838 DVB-T Realtek Semiconductor Corp.> at usbus3,
cfg=0 md=HOST spd=HIGH (480Mbps) pwr=ON (500mA)

ugen0.3: <RTL2838 DVB-T Realtek Semiconductor Corp.> at usbus0,
cfg=0 md=HOST spd=HIGH (480Mbps) pwr=ON (500mA)
```

The ugen3.2 device node is the replicated device to ugen0.3. Accessing either device (in this case by using rtl_test) works equally well. It's worth noting the new device node also honored my devd rule.

libusb20(3)

libusb20(3) is FreeBSD's own access library for processes to talk directly to USB devices instead of indirectly through some domain specific kernel module. A generic USB kernel module exposes an ioctl(2) interface that makes common USB operations available to user space. The libusb20(3) access library provides a developer friendly way to interact with the generic USB module without stuffing bytes into ioctl(2) calls.
Managing a USB device involves some of the following:

1. devd(8)
2. device enumeration
3. transfers
    - control transfers
    - bulk transfers
    - interrupt transfers
    - isochronous transfers

We've already covered devd(8).

Enumeration

The ability to enumerate USB devices is necessary to find the specific device for your application. Here's some code fragments to do just that. I assume you can read and write C or C++. Your source file(s) need these include statements.

```
#include <libusb20.h>
#include <libusb20_desc.h>
#include <dev/usb/usb_ioctl.h>
```

Enumeration involves the following functions:

- libusb20_be_alloc_default
- libusb20_be_device_foreach
- libsub20_dev_get_device_desc
- libusb20_be_free

libusb20(3) has the idea of a backend or BE for locating devices. For clarity, this has nothing to do with a ZFS BE. Different things altogether. In theory you could write your own backend to create mock USB devices for testing or for something that resembles USB over TCP. To date, I've not seen anyone implement USB over IP this way. The default backend is all we need here — but a description of a mock software only USB device appears at the end of this article.

Calling libusb20_device_foreach(3) allows enumerating the device list one node at a time. libusb20_dev_get_device_desc(3) can offer additional details about a device node that may be useful for enumeration.
This code fragment assumes an external function called "candidate" that returns true for a matching VID and PID and false otherwise.

```
bool (*candidate)(uint16_t vid, uint16_t pid);
```

It also assumes a function called "acceptable" that returns true if the usb_device_info data is acceptable to the application.

```
bool (*acceptable)(struct usb_device_info *info);
```

As to just what might be acceptable is up to you.

```
libusb20_device *handle = nullptr;

libusb20_device *walker = nullptr;
auto be = libusb20_be_alloc_default();
if (be == nullptr)
{
   printf(stderr, "libusb20_be_alloc()\n");
   return false;
}
// True means we found a suitable device and can stop looking.
bool okay = false;
walker = libusb20_be_device_foreach(be, walker);
while (walker != NULL)
{
   auto ddp = libusb20_dev_get_device_desc(walker);
   // matching vid and pid?
   if ( candidate(ddp->idVendor, ddp->idProduct) )
   {
       // yes: open it to get more info.
       if (libusb20_dev_open(walker, reserve) ==
          LIBUSB20_SUCCESS)
       {
           struct usb_device_info info{};
           libusb20_dev_get_info(walker, &info);
           // We can examine the usb_device_info contents to
           // further distinguish this device from others.
           okay  = acceptable(walker, info);
           if (okay)
```

```
            {
                handle = walker;
                libusb20_be_dequeue_device(be, walker);
                break;
            }
            // not the device we want so keep looking.
            libusb20_dev_close(walker);
        }
    }
    walker = libusb20_be_device_foreach(be, walker);
}
libusb20_be_free(be);
return okay;
```

The above code could be modified to return a list of usb_device_info objects presented to user as a pick-list. As written, the code returns a device pointer to the first candidate device it considers acceptable.

Take note of that *reserve* parameter to libusb20_dev_open(3). Its purpose will be explained when we get into transfers.

Once you find a suitable device, call libusb20_be_dequeue_device(3) to remove the device from the device list. This is a way to reserve the device although this isn't as useful as I'd like. If you re-enumerate the same device list, it won't be there. However, calling libusb20_be_device_foreach(3) in the same or a different process creates a new list with the device present. This won't give you exclusive access to the device.

Now that we have a device pointer, the next step is to open the device with libusb20_dev_open(3). We use the same device pointer going forward. There is no separate device handle. This does not grant exclusive access. One fix is to place an advisory lock on the device but this only works if all parties honor the advisory lock. Personally I think it's cool to allow two process to share a USB device (like a temperature / humitity sensor) but this is a rare use case. There's talk of extending the libusb20(3) ABI to accept an exclusive-use flag but that has not happened yet.

With an open device, we can now do some interesting things.

One is to retrieve device info. Another is to perform one of four types of transfers. So what are these four types of transfers?

The ***control transfer*** is the most common. These are short device specific commands. Each command is an eight byte header followed by an optional payload. Some send data to the device. Others retrieve data from the device. For simplicity, control transfers are usually done synchronously.
Interrupt transfers are a type of low-bandwidth, sporadic transfer. Keyboards and mice are two major examples that make use of interrupt transfers. Since these are normally handled by kernel modules, this article won't discuss interrupt transfers.

Isochronous and ***bulk transfers*** are very similar where isochronous transfers offer some extra guarantees on delivery (and fail if those guarantees can't be met). I mostly work with bulk transfers but the mechanism is so similar that someone needing to do isochronous transfers can use the bulk examples. Before I describe transfers, here's a loose analogy between a "ugen" or generic USB device and a /dev/tty serial port. On a serial port, you must set baud rate, bits per character, start & stop bits, and parity before sending and receiving character data. That step is analogous to USB control transfers that command a USB device. The sending and receiving of serial port character data is analogous to a USB bulk or isochronous transfer.

Control Transfers
Control transfers are often used to initialize a device or place it into a specific operating mode.

A control transfer can be one of two directions — IN or OUT. There's a fixed portion (LIBUSB20_CONTROL_SETUP_DECODED) and an optional section for data. The libusb20_dev_request_sync function includes a parameter to capture the number of bytes transferrred. This is the number of "extra" bytes that is in-addition to the bytes displaced by the setup header.

```
struct LIBUSB20_CONTROL_SETUP_DECODED setup;
```

This macro initializes the above struct to sane starting values.

```
LIBUSB20_INIT(LIBUSB20_CONTROL_SETUP, &setup);
```

Member bmRequestType is generally one of two values — for input or output.

```
setup.bmRequestType = LIBUSB20_REQUEST_TYPE_VENDOR |
LIBUSB20_ENDPOINT_OUT;

setup.bmRequestType = LIBUSB20_REQUEST_TYPE_VENDOR |
LIBUSB20_ENDPOINT_IN;

setup.bRequest = request;
setup.wIndex = index
setup.wValue = value;
setup.wLength = length;
```

The exact values of each of the above four parameters is device specific. wLength is the number of extra bytes. For an LIBUSB20_ENDPOINT_OUT request type, it is the number of extra data bytes to be written to the USB device. For web developers, this is loosely analogous to an HTTP/POST. Of course, your "posting" to a USB device, not an HTTP web server.

```
int status = libusb20_dev_request_sync(dev, &setup, data,
     &actual, TIMEOUT, 0);
```

dev is the device pointer. *setup* is an instance of LIBUSB20_CONTROL_SETUP_DECODED. *data* below is the pointer to the optional, extra data. *actual* captures the number of extra bytes transferred. For LIBUSB20_ENDPOINT_IN, *actual* may be less than or equal to the value specified in setup.wLength. For web developers, this is loosely analogous to an HTTP/GET.

libusb20_dev_request_sync blocks until the control transfer is complete. The bulk transfer examples below won't be synchronous so you can issue

overlapping requests. I avoid overlapping control transfers because the next control transfer often depends upon the success of (and data returned by) the previous control transfer.

Interrupt Transfers
Most devices that support interrupt transfers have supporting kernel modules so a need for programmatic user level access is rare.

Bulk Transfers
Bulk transfers allow sending or receiving chunks of data much like you might transfer data over a socket or file handle.

Some devices place a restriction on the size of bulk / isochronous transfers. Some chipsets I've supported impose a requirement that transfer sizes be multiples of 512 bytes.

For streaming data (like audio or video), you'll want to strike a balance between latency and overhead. If you transfer one megabyte of data as single 512 byte chunks, you're sending 2048 requests. You could also send the same data in a single request. Larger chunk sizes net lower CPU overhead but also impose a lag penalty if you decide you want to stop streaming or change content mid-stream. Scaling chunk sizes to match a desired frame rate (like 30 fps) allows pausing transfers sooner than a chunk size of — say – one minute.

Suppose you are streaming stereo audio samples at 192 ksps. Each sample is four bytes (16 bits per channel). That's 192k samples times 4 bytes per sample or 768 kilobytes per second. At 30 fps, one frame is 25600 bytes. Note that this just happens to be a multiple of 512. If your frame size isn't a multiple of 512, you'll need to round up or down. This implies your effective frame rate won't be exactly 30 fps. This is fine for most situations.

To stream continuously, you need to manage multiple outstanding transfers. You want one transfer that is in-flight and at least two more transfers waiting. Once a transfer completes, the buffer can be reused — and re-issued. Using the example above, you'd need at least three transfers of 25600 bytes each.

To stream in both directions, you'll need that many transfers for each direction.

I have found it most useful to break down streaming into four steps:

1. Reserve,
2. Start,
3. Stop,
4. Release.

Reserve allocates a list of transfer buffers. For bookkeeping purposes, I create a transfer context to keep transfer related bits together.

```
// USB transfer context.
struct Transfer
{
   void(unsigned char *data, unsigned size) *callback;
   unsigned char *buffer;
   struct libusb20_transfer *transfer;
   uint32_t size;
   uint32_t timeout;
};
struct libusb20_transfer *transfer = libusb20_tr_get_pointer(dev, i);
Transfer *context = new Transfer(back, transfer, bucket, 3 * timeout);
result = libusb20_tr_open(context->transfer, context->size, 1, 0x81);
// each buffer is ~30 milliseconds so one whole second is more than enough
      time to complete.
libusb20_tr_setup_bulk(context->transfer, context->buffer, context->size,
      context->timeout);
libusb20_tr_set_callback(context->transfer, bulk);
libusb20_tr_set_priv_sc1(context->transfer, context);
```

Note the above context tracks only one transfer. For three transfers, I need three Transfer objects. Remember that *reserve* parameter to libusb20_dev_open(3)? That tells the host kernel how many transfers are expected. For a device that performs only bulk and control transfers —

reserve is the number of bulk plus the number of control transfers. For this article, I use 2 control transfers and 3 bulk transfers so the *reserve* parameter is 5.

libusb20_tr_open(3) accepts a frame count and endpoint. I'm issuing each transfer as one frame. There are 16 possible IN and 16 possible OUT endpoints. The example given – 0x81 – is a bulk IN endpoint. Much like configurations and interfaces, endpoint designations will vary by device. Note that, in USB terminology — a set of transfers to a specific endpoint is also called a pipe. Tip: if control transfers work fine but libusb20_tr_open(3) fails, check your configuration index.

libusb20_tr_set_priv_sc1(3) allows us to associate a bare pointer with the transfer. I set this pointer to the context object managing the transfer. Its usefulness will become apparent soon.

Start calls libusb20_tr_start(3) once for each transfer buffer.
Stop calls libusb20_tr_stop(3) once for each transfer buffer.
Release frees the memory and resources associated with each transfer.

In the reserve stages, I like to allocate one large chunk as a single call to malloc instead of multiple smaller chunks. That simplifies the release stage to one call to free.

libusb20_tr_set_callback(3) sets the function pointer of the completion routine. This routine fills the transfer buffers with new data for outbound streams and copies data from the transfer buffers for inbound streams. It also re-issues the transfer to keep the stream going.

The completion routine looks much like the Reserve and Start steps combined.

```
uint8_t status = libusb20_tr_get_status(transfer);
uint32_t actual = libusb20_tr_get_actual_length(transfer);
uint32_t size = libusb20_tr_get_max_total_length(transfer);
Transfer *context = (Transfer *)libusb20_tr_get_priv_sc1(transfer);
```

```
// copying data to/from the buffer step goes here.
libusb20_tr_setup_bulk(transfer, context->buffer, context->size,
      context->timeout);
libusb20_tr_set_callback(transfer, bulk);
libusb20_tr_set_priv_sc1(transfer, context);
libusb20_tr_submit(transfer);
```

We can examine *status* to know if there were any errors. Under ideal conditions, *actual* should match *size*. After copying data to/from the buffer, the remaining functions are called. Note how libusb20_tr_get_priv_sc1(3) allows us to recover the context pointer from the transfer.

You can do the Reserve step and then do Start / Stop steps as many times as you like. You do not need to perform the Release step unless you want to exit or wish to change the geometry of the transfer buffers (more buffers, fewer buffers, larger buffers, smaller buffers). If your new geometry exceeds the *reserve* parameter in libusb20_dev_open(3), you must close and re-open the device with the new *reserve* value.

Be sure to do the Stop step BEFORE the Release step. The heap will become very angry if outstanding I/Os modify the data buffers that you freed — becoming a "use after free" bug.

Isochronous Transfers

I've not had the need to deal with isochronous trasnfers so I must leave you hanging. My understanding is that the setup effort is very similar to bulk transfers. See libusb20_tr_setup_isoc(3).

Event Pump

The last piece is the event pump. This is similar to the event pump in an X.org application. Instead of pumping X windows events, you are pump USB events. This thread of execution ultimately calls the completion routines. Without the event pump, your code won't know your transfers have completed. The pump code is very small. Most programs place this pump in a background thread. This example polls 30 times a second.

```
while (flag)
{
   int milliseconds = 1000/30;
   if (libusb20_dev_process(device) == 0)
   {
       libusb20_dev_wait_process(device, milliseconds);
   }
}
```

Measuring Latency

Measuring latency is useful for determining how well the process is at keeping up with incoming data. The libusb20_tr_pending(3) function asks the USB stack if it thinks a transfer is complete. Counting the number of completed (but not yet processed) buffers gives us an idea of how far behind the process is at keeping pace. If all buffers are complete, the process has likely missed data since there are no empty transfers to capture data. A short term fix is to the increase the number of buffers. A better fix might be to put effort into the processing code for faster per-transfer turn-around times.

Clearing Halts

A device can flag an endpoint as halted. While halts are generally not a problem for control transfers, they literally bring bulk and isochronous transfers to a halt. The underlying cause can be a loose cable or some other weirdness. Clearing the halt allows the host to resume transfers to/from that endpoint. The libusb20_tr_clear_stall_sync(3) function requires a transfer pointer. Rather than recycle one of my bulk transfer pointers, I reserve one extra transfer pointer in libusb20_dev_open(3). That way I always have a spare transfer pointer specifically for clearing halts.

As an example, your application needs N bulk transfer pointers so you call libusb20_dev_open with a max transfers count of N + 1, here are the steps to clear a halt on endpoint 0x81.

```
const uint8_t ep = 0x81;
auto transfer = libusb_tr_get_pointer(handle, N);
```

```
auto transfer = libusb_tr_open(handle, 0, 1, ep);
if (transfer != nullptr)
{
   libusb20_tr_clear_stall_sync(transfer);
   libusb_tr_close(handle);
}
```

Note the parameters to libusb20_tr_open — particularly the size of zero and count of 1. That's all you need to clear a halted endpoint.

Backend Hacking

Here's a dependency injection hack for using the USB backend mechanism to inject a pseudo device that exists entirely in software. All interaction occurs through the libusb20(3) (and libusb) library. First, create your own implementation of libusb20_be_alloc_default(3).

Your implementation will use dlsym(3) to locate and call the real libusb20_be_alloc_default(3).

It will return a modified copy of the backend structure returned by libusb20_be_alloc_default(3). This modified structure contains function pointers. You can splice in your own code to intercept calls to the default backend. From there you can do things like intercept calls to libusb20_dev_open(3) to return a pseudo device that responds to control and bulk or isochronous transfers.

This allows for some very fine-grained unit testing of your libusb20(3) and libusb(3) client code without the need for actual hardware. Coding your own backend requires these two header files — in this order:

```
#include <sys/queue.h>
#include <libusb20_int.h>
```

DTrace

While coding around the access library makes debugging much easier, sometimes you need some visibility into the raw ioctls. For example, a call to libusb20_tr_open(3) fails but the library returns a very generic

LIBUSB20_ERROR_INVALID_PARAM. The following dtrace snippet will tell you the actual return value. Knowing this value, you can examine the ugen driver kernel source for clues as to why it failed.

```
dtrace -n 'fbt:kernel:usb_ioctl:return /execname ==
"my_usb_code"/ { printf("%d\n", (int)arg1); }'
```

libusb20(3) advantages over libusb(3) and WinUSB

I like how simple and direct the libusb20(3) ABI is. It offers just enough complexity to get the job done while keeping your code readable.
The big advantage is software license. Licensing and copyright matter. The real libusb(3) library found on Linux and Windows is LGPL. There are many situations where LGPL is perfectly acceptable.

libusb20(3) (and the libusb(3) compatible shim) – being a part of FreeBSD – is BSD-2-Clause. That's a very permissive license. The same is true for the FreeBSD version of pthreads. On FreeBSD, it is easier to avoid compliance issues around (L)GPL and linking exceptions. Enough about licenses and copyright.

One chief technical advantage is I can find a device and pull string descriptors from the device even if the device is owned by another process. In Linux and Windows libusb(3) implementations, the device disappears from the device list. This has historically been confusing to end users when their favorite software tells them their shiny USB connected gadget isn't there — only because another process got to it first.

A utility like usbconfig(8) could not be implemented using libusb(3) — at least not on non-FreeBSD platforms because of how libusb(3) implements enumeration on those platforms. You can't describe what you can't see.
I also include Microsoft's WinUSB for this comparison. Issuing transfers in WinUSB is very similar so there's not much difference at this level. However, Windows USB device enumeration is horrific. It takes hundreds of lines of code compared to a paragraph of code for FreeBSD or Linux. Once upon a time, there were also breaking WinUSB (via SetupAPI) changes between Windows versions (between 7 and 8). I personally have not yet seen a breaking change with libusb20(3).

libusb20(3) disadvantages to libusb(3) and WinUSB

As hinted at earlier, there are distadvantages too.

One chief disadvantage is a lack of exclusive device access. On Windows and Linux, if two processes enumerate the same device and attempt to open it — the first one wins. On FreeBSD, they both win which can cause some confusion. This is a noteworthy problem for ports that assume Windows or Linux behavior.

An advisory lock provides a fix for exclusivity but this only works if adopted by all parties.

The biggest disadvantage is a lack of portability. You won't see libusb20(3) on anything but FreeBSD and FreeBSD derivative operating systems. For my use cases, that is perfectly fine.

If you have a USB gadget you'd like to support on FreeBSD, consider using libusb20(3) to write some code to exercise the device. Debugging from userland will be easier than debugging a kernel module. Once you have working code, you're done. If you insist on a kernel module, use the userland code as a basis for your new kernel module that exercises the device in the same manner.

Here's a libusb20 implementation of a bare minimum usb-list program.

```
#include <assert.h>
#include <unistd.h>
#include <stdio.h>
#include <libusb20_desc.h>
#include <libusb20.h>
#include <sys/queue.h>
#include <libusb20_int.h>
#include <dev/usb/usb_ioctl.h>

int main(int, const char *[])
{
   auto be = libusb20_be_alloc_default();
   struct libusb20_device *device = nullptr;
```

```
    while ( (device = libusb20_be_device_foreach(be, device)) != nullptr)
    {
        int error = libusb20_dev_open(device, 0);
        if (error == LIBUSB20_SUCCESS)
        {
            struct usb_device_info info{};
            libusb20_dev_get_info(device, &info);
            fprintf(stdout, "%s %04hX (%s) %04hX (%s) %s [%d]\n",
                    libusb20_dev_get_desc(device),
                    info.udi_vendorNo,
                    info.udi_vendor,
                    info.udi_productNo,
                    info.udi_product,
                    info.udi_serial,
                    info.udi_config_index);
            libusb20_dev_close(device);
        }
    }
    libusb20_be_free(be);
    return 0;
}
```

Here's a loose equivalent using libusb that should be portable to other host OSes.

```
#include <assert.h>
#include <unistd.h>
#include <stdio.h>
#include <libusb.h>

int main(int, const char *[])
{
    libusb_context *context = nullptr;
    libusb_device **list = nullptr;
    unsigned index = 0;
    libusb_device *device = nullptr;
```

```
int result = libusb_init(&context);
if (result == LIBUSB_SUCCESS)
{
    if (libusb_get_device_list(context, &list) >= 0)
    {
        while ( (device = list[index++]) != nullptr )
        {
            libusb_device_handle *handle = nullptr;
            struct libusb_device_descriptor desc{};
            result = libusb_get_device_descriptor(device, &desc);
            if (result == LIBUSB_SUCCESS)
            {
                result = libusb_open(device, &handle);
                if (result == LIBUSB_SUCCESS)
                {
                    int config = 0;
                    uint8_t make[128]{};
                    uint8_t product[128]{};
                    uint8_t serial[128]{};
                    result = libusb_get_string_descriptor_ascii(handle,
                        desc.iManufacturer, make, sizeof make);
                    if (result <= 0)
                        sprintf((char *)make, "vendor %04hX", desc.idVendor);
                    result = libusb_get_string_descriptor_ascii(handle,
                        desc.iProduct, product, sizeof product);
                    if (result <= 0)
                        sprintf((char *)product, "product %04hX", desc.idProduct);
                    libusb_get_string_descriptor_ascii(handle, desc.iSerialNumber,
                        serial, sizeof serial);
                    libusb_get_configuration(handle, &config);
                    fprintf(stdout, "ugen%d.%d %04hX (%s) %04hX (%s) %s [%d]\n",
                    libusb_get_bus_number(device),
                            libusb_get_device_address(device),
                            desc.idVendor,
                            make,
                            desc.idProduct,
```

```
                                product,
                                serial,
                                config);
                        libusb_close(handle);
                    }
                }
            }
            libusb_free_device_list(list, 0);
        }
        libusb_exit(context);
    }
    return 0;
}
```

Additional Reading

DTrace – Chapter 28 (FreeBSD Handbook)
https://docs.freebsd.org/en/books/handbook/dtrace/

USB Devices – Chapter 13 (Arch handbook)
https://docs.freebsd.org/en/books/arch-handbook/usb/index.html

USB Devices – Chapter 29 (Freebsd handbook)
https://docs.freebsd.org/en/books/handbook/usb-device-mode/index.html

---

Rick Parrish is a long-time C++ developer whose interest in USB devices led him to discover FreeBSD libusb20 access library. Finding gaps in the library's documentation, he has submitted small improvements to the FreeBSD documentation team.

Copyright © 2026 held by owner/author. Publication rights licensed to the FreeBSD Journal. This work is licensed under a **Creative Commons Attribution International 4.0 License**.

3980 Broadway Street
STE #103-107
Boulder CO 80304
1-720-207-5142
info@freebsdfoundation.org

