DIY USB

IN THIS DOCUMENT

The USB (Universal Serial Bus) standard has been with us for many years, but making USB devices is still a daunting task - the USB specification comprises thousands of pages spread over tens of documents, and although good books have been written on the subject, they are rarely shorter. In addition, the API offered for programming USB devices is often complex and intricate.

This article gives the reader a handle on programming their own software-based USB devices. It is not limited to standard class devices, but also presents a way to implement any device, whether they comply with a standard class or not.

1 The USB way of thinking

To understand USB, one has to understand a dozen terms on which the USB world is built. USB separates the *host* from the *device*: there is one host, connected to multiple devices. All traffic is initiated by the host, and the host schedules traffic on the USB bus.

A device is a physical box at the end of the USB cable, that identifies itself to the host by passing it a *Device Descriptor* and a *Configuration Descriptor*. These descriptors are binary data that describe the capabilities of the USB device. In particular, the Configuration Descriptor describes one or more *interfaces*, where each interface is a specific function of the device. A device may have multiple interfaces, for example a USB device that comprises a keyboard with built-in speaker will offer an interface for playing audio, and an interface for key presses.

Each interface comprises a series of *endpoints* that are the communication channels between the host and the device. Endpoints are numbered between 0 and 15, and may be *IN* endpoints or *OUT* endpoints. These terms are relative to the host: OUT endpoints transport data to the device and IN endpoints transport data to the host. Endpoints may be of four types:

- Bulk endpoints transport data whenever required and reliably; bulk data is acknowledged and therefore fault tolerant.
- Isochronous endpoints are for transporting real-time data. Isochronous endpoints have a fixed bandwidth allocated to them; the host allocates this band-

Publication Date: 2014/7/8 Document Number: XM001713A

width and will not allow an isochronous endpoint to be created if there is no bandwidth available. In contrast, bulk endpoints have no guaranteed bandwidth.

- Interrupt endpoints are polled occasionally by the host, and enable a device to report status changes.
- The control endpoint (endpoint 0) is used to perform general operations, such as obtaining descriptors, or performing a control-operation such as “Change the volume” or “Set the baud rate” on any of the interfaces.

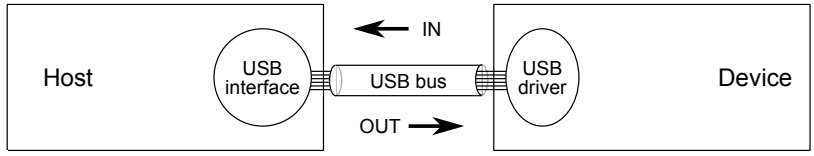


Figure 1: Traffic over the USB bus.

USB traffic is organised in *frames*. Frames are marked by the host sending a *start of frame* (or *SOF*) every 125 us (for high speed USB) or every 1 ms (for Full Speed USB). Isochronous endpoints are allocated a transfer in every frame. Interrupt endpoints are polled once every so many frames, and bulk transfers may happen any time when the bus is not in use.

As an example, the aforementioned keyboard with built-in speaker has at least two endpoints, an isochronous OUT endpoint to transfer audio data to the speaker, and an interrupt IN endpoint to poll the keyboard. Suppose that the speaker is a mono-speaker with a 48 kHz sample rate, then the host will send 6 samples of data every 125 us (6 samples/0.000125 seconds = 48,000 samples/second), and if a sample occupies 16 bits, then the host will reserve enough bandwidth to send a 96-bit OUT packet every 125 us frame. This will consume around 0.5% of the USB bandwidth, the remainder 99.5% is free for other interfaces, or other USB devices on the same bus.

2 Specifying and discovering device capabilities

The host initiates all USB traffic. When a device is plugged in, the host first requests the *device-descriptor*. This descriptor contains two sets of information that inform the host of the basic capabilities of the device: the device class and the VID/PID.

The class and subclass can be used to specify a device with generic capabilities. A USB speaker advertises itself as being of class *Audio-2.0*. A keyboard advertises itself as a *HID* (Human Interface Device) class device. The previous example of a device with both a speaker and a keyboard advertises itself as a *Composite device* class.

USB devices that comply with a specific USB class enable cross-vendor and cross-platform compatible USB devices. The USB specification specifies hundreds of device classes that enable the generic implementation of, for example, Ethernet dongles, mixing desks, or flash disks, and enable operating systems to provide generic drivers for these classes.

There are cases where the USB device does not fit a specific class, or where the class specification is too constrained for a particular device. In that case, the class of the device must be described as *Vendor specific*. The operating system shall then use the *VID* (Vendor ID) and *PID* (Product ID) in order to find a vendor-specific driver.

When the device descriptor has been dealt with, the O/S assigns the USB device a number, informs the USB device of the number (it is being *enumerated*), and the O/S requests the *Configuration Descriptor*. The configuration descriptor specifies each interface in detail. In the case of the earlier example the configuration descriptor will specify two interfaces: one interface of class USB-Audio-2.0 with a single channel output endpoint running at 48 kHz only, the other interface of class HID that specifies a single keyboard with a specific keymap.

There are cases where the USB device does not have any operating system support, and it should interact with a user program directly. In that case, a generic driver such as the *libusb* driver that allows an application program to communicate with any USB device can be used. Typically, the device will be advertised as vendor-specific, and the user program can through the libusb interface detect a device with VID and PID that it wants to interact with, claim an interface, open an endpoint, and send IN and OUT requests to that endpoint.

3 What to do with your data

The enumeration of the device typically requires static descriptors to be sent to the host. The difficult bit is to create the descriptors; serving them is simple as that is the only task required of the device at the time. However, after enumeration, data may arrive or be requested on all endpoints in quick succession. This requires an interface between the software that deals with the function of the USB device (e.g. playing audio, or monitoring key strokes on the keyboard), and the low level USB protocol. Prior to designing the interface, let's look at how to handle data on various types of endpoints.

3.1 Bulk endpoints

Bulk endpoints are the easiest to deal with. Since each data transfer is acknowledged, it is possible to send a *negative acknowledge* or *NAK* stating that the device is not yet ready to deal with the endpoint. For example, if software is dealing with some other part of the device, or if data is simply not yet available (for example a read from FLASH memory is not yet completed) the low level USB driver can send a NAK.

However, sending NAKs has a downside. The only sensible option for the host is to retry the request, thereby potentially creating a long series of requests that are aborted by NAKs. This wastes USB bandwidth that could have been used by other endpoints or devices. In addition the host software is blocked until the device answers. Hence, NAKs should be a last resort; it may be more appropriate to send partial data then to NAK an IN request. In the case of an OUT request there is little that can be done; if there is no room to accept the data then a NAK is the only answer. However, it may be more appropriate to introduce a high level protocol that will not allow an OUT request until there is space.

3.2 Isochronous endpoints

Isochronous endpoints are more difficult to deal with because these endpoints are not acknowledged. The transmitter (in either direction) assumes that the data arrives. Since there is no acknowledgement on an isochronous endpoint, there is no possibility to send a NAK. Hence, if the device is not ready, the only course of action is to drop the data from an OUT packet, or to send no data for an IN packet.

Although this may seem harsh at first, remember that the purpose of an isochronous endpoint is to transmit real-time data in a *guaranteed* time slice of the USB bus. If the device does not have room to store the OUT data, then data is probably not dealt with in real-time, hence dropping is a sensible course of action. If there is no data available to answer an IN request, then the device has not collected enough data. A sensible course of action is to transmit whatever data is present; possibly no data at all.

Assuming that the data can be processed or produced in real-time, it is easy to compute the buffer requirements for an isochronous endpoint.

- For an OUT endpoint, The worst possible case is that the host posts one OUT request right at the end of a USB frame, and then immediately after the SOF it posts a second OUT request. This means that two OUT requests, carrying 250 us of data, are received in quick succession. Hence the buffering scheme must be able to buffer at least 250 us worth of data. As long as the program does not consume data from this buffer until the SOF following the first packet, then the buffer will never empty, providing a continuous data stream from host to device.

- For an IN endpoint, the worst case is similar. The host could perform two IN transfers in short succession just before and immediately after a SOF. This means that the IN buffer needs to be at least 250 us too, and the buffer should contain at least 125 us at the start of each frame.

3.3 Interrupt endpoints

Interrupt endpoints enquire about current state. This may be data that is not too time critical (such as a key press), or it may be time critical data (such as the X and Y location of a mouse or other pointing device). In the first case a few ms delay between typing the key and reporting it won't hurt. However, when reporting mouse locations, irregular reporting may lead to unintended results.

3.4 Errors

It is worth comparing bulk and isochronous transfers from a perspective of how to cope with errors. In bulk transfers, the data itself is critical; The host and device can retry and slow down, as long as the data is transferred correctly and this transfer must be acknowledged. For an isochronous transfer it is the timing that is critical: either side can throw data away, as long as the real-time characteristics of data further along in the stream are adhered to (of course, the decision to drop data should not be taken lightly as it will have an impact on the fidelity of, for example, a video or audio stream).

The data-centric versus time-centric approach has a knock-on effect on the consequences of bit-errors. All USB traffic is protected by means of a CRC for error detection. A corrupted bulk transfer must be retried until the data is transferred without error. In contrast a corrupted isochronous transfer will simply be dropped. The transmitting side will be unaware that data was dropped. The receiving side may know that the transfer was dropped (if the header with the endpoint was not corrupted), but even then it cannot be certain how many bytes the transfer contained. When streaming real-time video or audio this is important, since there will be an unknown gap in the stream that has to be filled with best effort.

4 Programming USB Devices

Having seen how to deal with different types of endpoints, we can develop a programming model for software based USB devices. It is helpful to keep in mind how USB operates:

- There are one or more endpoints, for one or more interfaces, on which traffic may arrive or depart at any time.
- Transfers on isochronous endpoints are time critical.
- There is at most one transfer happening at a time.

The first two bullet points suggest a multi-threaded programming structure (shown in Figure 2, Figure 2); especially if more than a single interface is concerned, or if isochronous endpoints are being used. The basic software architecture assumes that there is some sort of USB device library, and that for each endpoint we implement a thread that deals with USB transfers on that endpoint. Other parts of the system, not directly connected to the USB device library, are implemented using additional threads.

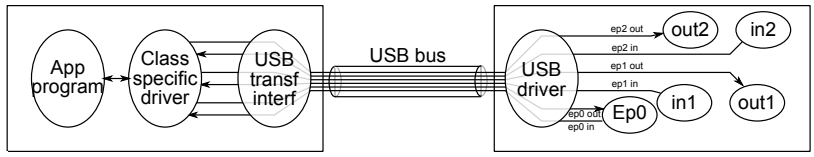


Figure 2: Software architecture for handling multiple endpoints

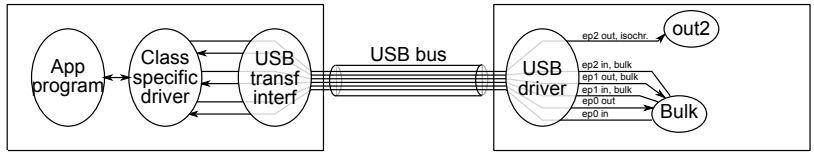
Note that one thread per endpoint may not be required, and may not be the most elegant method. Given that only one transaction happens at a time (the third bullet point) we can create a version of the system that relies on fewer threads in the system. Suppose for example that we want to implement a synchronous protocol over two endpoints where the host will always transmit data over a bulk OUT endpoint, prior to receiving data on an associated IN endpoint. This protocol requires only a single thread that handles OUT and IN transactions in order on that endpoint.

This optimisation is not without risk. Using a single thread per endpoint naturally caters for the situation where the host program was aborted and restarted between the OUT and IN transaction. In this case, the sequence of transactions seen on the

device will be ..., OUT, IN, OUT, IN, OUT, **OUT**, IN, ..., and the thread dealing with OUT transactions must swallow the extra OUT. When optimised away to a program that sequentially consumes OUT and IN in order, this program must be written so that at any time it may expects the protocol to reset.

The third bullet point enables a further optimisation. A single thread can deal with *all* bulk traffic on *all* interfaces. This architecture is shown in Figure 3, Figure 3. The single thread receives a request (IN or OUT) on any endpoint, dealing with that request, whereupon it moves on to the next request, possibly on a different endpoint. If the next request arrives before the last request has been dealt with completely, then the USB device library sends NAKs, temporarily holding up the host. This optimisation has one disadvantage, which is that the single thread must keep state for each endpoint, and is effectively context switching on each request. We will show an example of this later.

Figure 3: Optimising multiple endpoints into a single thread



The same optimisation cannot be applied to isochronous endpoints. If we had a single thread dealing with all isochronous data, then this would involve FIFOs for each endpoints from which the thread will read data or post data. These FIFOs will increase latency which is often undesirable.

The rest of this article discusses two examples of the software architecture and optimisations: one example that uses vendor specific drivers and mostly bulk endpoints (JTAG over USB), and one that shows a standard USB class with mostly isochronous endpoints (Audio over USB).

4.1 JTAG over USB

For debugging programs on embedded processors, it is common to use a protocol such as JTAG for accessing the internal state of the processor, and to use a program such as gdb to run on a PC to interpret and modify state, set breakpoints, single step, and so on. USB can be used to provide a cross platform portable transport layer between the PC and JTAG wires.

These devices are often referred to as JTAG keys. In addition to JTAG, they often contain a UART for text I/O from the embedded program. JTAG keys do not follow any standard USB class, and hence the descriptor labels them as vendor-specific and it is up to us to define an endpoint structure that is fit for purpose. One endpoint structure would use six endpoints:

- two endpoints that control the USB device itself (endpoint 0 IN and OUT, required by USB)
- an IN and OUT endpoint for JTAG traffic.

- an IN and OUT endpoint for UART traffic

Since there is no USB standard we can define the protocol for the JTAG traffic, and choose a set of commands such as “send a clock with TMS high”, or “read the program counter”. On the host side, our program can use libusb to search for a device with our VID and PID, claim the interface, and then use the libusb interface to send IN and OUT transactions to both the JTAG and UART endpoints.

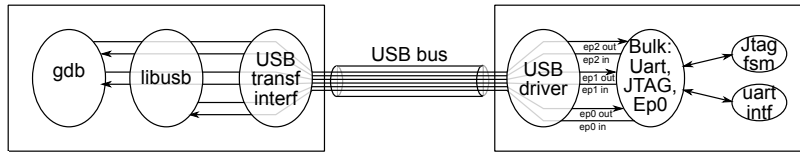


Figure 4: JTAG over USB

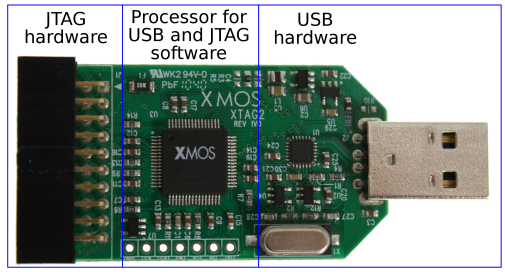


Figure 5: JTAG over USB hardware using standard 20-pin connector

A suitable software architecture for the device end is shown in Figure 4, Figure 4. Given that all endpoints are for bulk traffic, they can all be mapped onto a single thread, and have two separate threads to deal with the state machines for JTAG traffic and UART traffic. A sample implementation is shown in Figure 5, Figure 5.

4.2 Audio over USB

As an example of a standard USB device, let’s discuss Audio over USB. The Audio-2.0 Class standard allows interoperability of devices on platforms: a consumer can buy a USB microphone or USB speakers, and plug it into any computer that supports Audio over USB. More complex devices are also supported: the descriptor has a syntax for describing mixers, volume controls, equalizers, clocks, resampling, MIDI, and many other functions, although not all of those functions are recognised by all operating systems.

On the host side, all USB traffic carrying audio samples is directed to the USB-Audio driver, which interacts through some general kernel sound interface with the program using audio, such as Skype. Other data, such as MIDI can be handled through a separate interface by a separate driver.

Since the device is designed to be Audio-2.0 Class compliant, there is no flexibility in the protocol. If the application has to support MIDI, stereo in, and stereo out with a clock controlled by the device, then the standard dictates that there shall be seven endpoints:

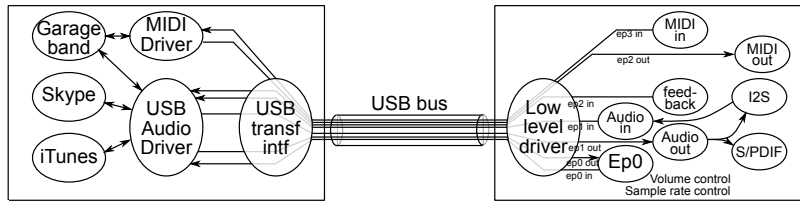


Figure 6: Diagram showing Audio over USB software.

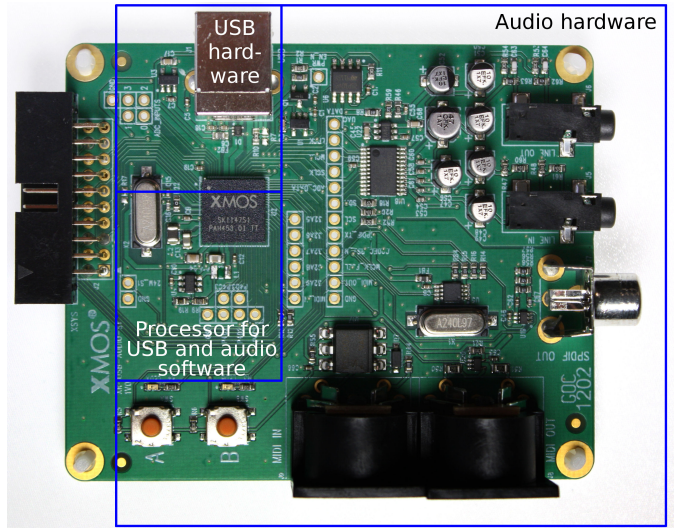


Figure 7: Audio over USB hardware.

- two endpoints that control the USB device itself (endpoint 0 IN and OUT, required by USB)
- an isochronous IN endpoint for the I2S Analogue to Digital Converter (ADC).
- an isochronous OUT endpoint for the I2S Digital to Analogue Converter (DAC).
- an isochronous IN endpoint for feedback on the clock speed.
- A bulk IN endpoint and bulk OUT endpoint for MIDI.

The endpoints for the ADC and DAC have one IN and OUT transaction every microframe, every 125 us. Assuming that the DAC and ADC operate with a 96 kHz sample rate, that means that 12 samples are sent in each direction every 125 us. Note that there are two independent oscillators: the device controls the 96 kHz sample rate, and the host controls the 125 us microframe rate. As these clocks are independent, they will drift relative to each other, and there won't always be 12 samples in each transfer. The vast majority of the transfers will have 12 samples, but sometimes there will be 13 or 11 samples. The third isochronous endpoint is used by the device to inform the host of the current speed. It is sampled once every

few ms and reports the current sample rate in terms of samples per microframe. The MIDI endpoints carry MIDI data as and when available.

The software architecture for this device is shown in Figure 6, Figure 6. Unlike the previous example, there is little that can be optimised. The endpoint structure is dictated by the class specification, and with three isochronous endpoints, it is advisable to have three processes ready to accept and provide data on these endpoints. The only optimisation that is feasible is for Endpoint 0 and the MIDI endpoints to be handled by a single thread. A sample implementation is shown in Figure 7, Figure 7 .

5 Summary

USB devices are composed of a multitude of interfaces that run concurrently, and a number of endpoints that are either bulk or isochronous. Bulk endpoints are for reliable data transport between host and device, whereas isochronous endpoints are for real-time data transport.

When programming USB devices endpoints it is easiest to see those endpoints as individual software threads. Some of those can be mapped onto a single thread, but the programmer has to understand the consequences. In particular, mapping multiple isochronous endpoints onto a single software thread will introduce an (unpredictable) latency in the real time stream.