Plan 9's Universal Serial Bus

Francisco J. Ballesteros
nemo@lsub.org

ABSTRACT

The Universal Serial Bus is a complex and, therefore, a popular bus on personal computers and other devices. Many devices including disks, keyboards, mice, and network cards are attached to computers using it. The bus is a tree with hubs as nodes and devices as leafs and uses polling from the root of the tree, which is the bus controller. It permits hot plugging and removal of devices at any time. This paper describes the system software used on Plan 9 to drive the bus and its devices. How to use the software is not described here, but in the Plan 9 user's manual.

1. Introduction

The Universal Serial Bus (or USB) [1] is a standard bus developed by a set of corporations including Intel, Compaq, Microsoft, Digital, IBM, and Northern Telecom. It started in 1994 with version 1 supporting two transfer modes: Low speed transfers at 1.5 Mb/s and full speed transfers at 12 Mb/s. Version 2 was introduced on 2000. It defined, so-called, high speed transfers capable of 12 Mb/s. This is the version in use today and is what the software described on this paper can drive. Version 3 of the standard was released in 2008 introducing 5 Gb/s transfers (called superspeed transfers). However, as of today, there is no hardware in the market supporting this version of the standard.

The bus structure is certainly complex, when compared to other buses, mostly because of the requirements on the software. It is a made of a tree of devices with a host controller at the root, hubs implementing branches, and devices attached to leaves of the tree (to USB ports). There can be up to 127 devices attached to the bus including hubs.

USB devices are standarized into classes of devices, further divided into subclasses, and sets of devices speaking a particular protocol. Together, class, subclass, and protocol identify a device type, codified as a number known as a ''CSP''. In practice some devices belong to a ''vendor specific class'' that may contain any type of device, rendering the CSP useless. In this case vendor and product identifiers are the only choice to determine which one is the device at hand.

A device may be in fact a combination of different devices packaged together. For example, keyboard and mouse combos are packaged into a single device attached to a single USB port. In part because of this, in part to try to simplify the interaction of the software with devices, a device includes different addressable entities called *endpoints*, grouped into *interfaces*. Each interface is an administrative entity that has its own CSP and includes one or more endpoints. In our example, a keyboard and mouse combo may provide two interfaces (one for the mouse and one for the keyboard).

Initially, all devices include a zero endpoint used for configuration purposes. The setup endpoint is available as long as the device is attached. Other endpoints may be configured later by the software, as dictated by the device. These are grouped into

interfaces, which correspond to a function performed by the device. Each interface has also an associated CSP that identifies its type. In few words, an endpoint is an artifact used for I/O and has an associated CSP indicating its purpose.

As of today, the Plan 9 USB software supports version 2 of the bus including several drivers for disks, keyboards, mice, serial lines, ethernet cards, and KNX devices. It is likely that more drivers will be added in the future. The software is responsible for enumerating devices on the bus, configuring them, and providing interfaces to use them and perform actual I/O.

The enumeration process consists on detecting devices attached to the bus and assigning addresses to them. A newly attached device uses a well-known configuration address to permit the software performing the enumeration to reach the new device. A consequence is that only one device can be attached and configured at a time. Once the new device has been given an address, another port may be permitted to attach another device, which starts using the configuration address. Enumeration has to take into account that hot-plugging is supported by the bus so that devices may be attached and removed at any time.

Device configuration may not be trivial for some devices. This means that it is better to keep as much of the USB software as feasible outside of the kernel. At least, the part responsible for configuring devices. Configuration is generic in principle, because devices include data to describe themselves. However, for many devices it is necessary to perform specific configuration tasks, which may be complex as well. Once a device is configured, one or several data pipes (endpoints) are available for use to operate on the device.

For efficiency reasons it is desirable to keep the mechanism used for I/O within the kernel. The idea is that after a device driver has configured a device the kernel provides the actual mechanism for performing I/O and programs may perform `read` and `write` system calls to obtain and to send data. For some devices, which require following specific interaction protocols, this may not be possible and a driver must sit between the application and the I/O mechanism provided by the kernel.

Device combos have implications for device drivers. A single device driver must be responsible for the entire device, but several drivers may be required to deal with the different functions of the device. That is the case of `kb`, which handles mouse and keyboard combos and thus corresponds to two different drivers.

Last but not least, several USB devices may be required for use during the boot process. For example, keyboards and mice. This implies including them into the kernel as boot files. Embedding all these drivers into the kernel as separate programs replicates multiple times almost the same code for the C library and the 9P library, among other utilities. Some mechanism is necessary to avoid this duplication.

These are the requirements on the software to drive USB. In what follows we describe this software after a brief overview of the USB hardware and its protocol.

2. USB hardware and protocol

The purpose of USB is to perform transfers between the host controllers and devices attached to the bus (see figure 1). All details are described in the specification [1]. Here we introduce only the most relevant ones for understanding the rest of this paper. As the Plan 9 software does, we also deviate from the standard in the description, for simplicity.

There are three different controllers in use today. Two of them implement USB version 1: the Universal Host Controller Interface, or UHCI [4], and the Open Host Controller Interface, or OHCI [2]. The former was developed by Intel and the later was developed by Compaq, Microsoft, and National Semiconductor. Both are in use on current hardware.

The main difference between UHCI and OHCI is that the former does not provide any indication whatsoever regarding which transfer is responsible for an interrupt, while the later includes a more sensible interface to the software.

USB version 2 is implemented by the Enhanced Host Controller Interface or EHCI [3] (which like UHCI does not provide any mean to know which transfer is responsible for an interrupt). The EHCI controller is usually packaged along with one or more version 1 controllers, called companion controllers. This means that supporting USB 2.0 requires drivers for all three controllers. In version 3 an Extensible Host Controller Interface, or XHCI, includes a companion version 2 controller (therefore it can be expected that driving the four controllers will be necessary to drive USB 3.0). Its specification is not yet available.

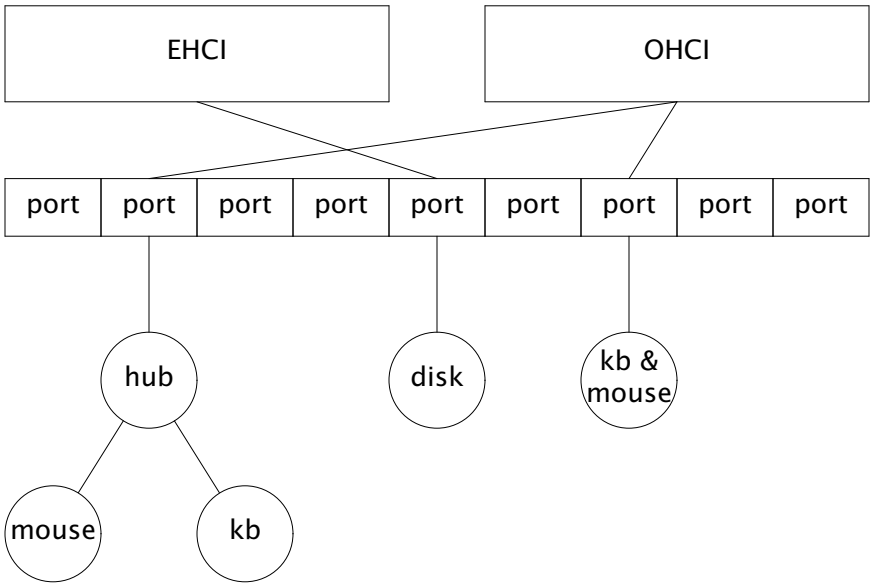

Figure 1. An example Universal Serial Bus.

Packaged with the controllers are included a series of USB ports, where devices and hubs may be attached (Hubs attach to a port and provide extra ports on the bus.) In theory the root of the device tree is a hub, called a root hub. In practice the root hub must be implemented by software. Most other systems do so. We tried not to do it to avoid a significant amount of software.

On a 2.0 USB system ports on the root hub (that is, included in the host controller) may be independently routed to either the 2.0 controller (EHCI) or to a companion USB 1.0 controller. This is done depending on the device speed. Full and low speed devices are routed to the companion controller and high speed devices are kept routed to the EHCI. For example, in the figure 1 a 1.0 hub is attached to the 2nd port. The companion OHCI is responsible for this USB 1.0 subset of the USB 2.0 bus.

Data transfers are implemented by the host controller, which is able to poll devices on the bus using TDMA and a communication protocol spoken on the USB wires. This happens both for input and for output. There are four types of transfers used to talk to endpoints, but it must be noted that an endpoint is capable of only one transfer type:

- *Control transfers* are RPCs to the device. Among other things, they permit configuring the device and recovering from soft errors.
- *Bulk transfers* are unidirectional transfers intended to send or receive a significant amount of data (e.g., 512 Kbytes).

- *Interrupt transfers* are not really interrupts, but unidirectional and small data transfers (e.g., 8 bytes). They are asynchronous in nature but are not so in practice (devices are polled). Ironically, to achieve the asynchronous character of the transfer the device has to be polled often, meaning that these transfers are considered synchronous by USB controllers.
- *Isochronous transfers* are unidirectional transfers that must be performed on a timely basis. For example, audio output. They sacrifice error checking in favor of timeliness.

There are forbidden combinations, but in general, for each transfer type we have full, low, and high speed variants of the transfer type. Full and low speed ones are supported by all three controllers; High speed ones are supported only by EHCI. Bulk, interrupt, and isochronous transfers correspond one to one to transfers understood by the controller. Control transfers do not.

A control transfer is actually a sequence of two or more transfers. First, a *setup* transfer asks the device to perform a control request, perhaps requiring a data transfer from the host to the device or from the device to the host. Second, if the request requires exchanging data between the controller and the device, a second transfer exchanges data (this may be more than one transfer if not all data fits in a USB packet). Third, an empty data transfer from the device to the controller reports the status for the entire control transfer.

USB devices must understand a set of standard control requests, described in chapter 9 of [1]. However, many devices implement non-standard requests (or perhaps standarized requests that are specific of the device class). The most popular control requests are those that retrieve *descriptors* from devices. These are standarized binary descriptions for devices and device features. Some descriptors have to be implemented by all devices while others are supported or not depending on the device. Obtaining device descriptors is necessary to learn how many endpoints there are, what their types are, and how to build a request to activate them and enable I/O.

On the bus a transfer requires multiple packets. Chapter 8 of [1] describes the protocol. Most details are uninteresting but the addressing and the data acknowledge mechanism are important for the software.

Bus addresses are made out of a device address and an endpoint address. That is, the addressable entities are the endpoints and not the devices. Device addresses are assigned by the software. Endpoint addresses are dictated by the hardware.

All devices have initially (after attachment to the bus) an endpoint with address zero, known as the device's setup endpoint. It is always an endpoint using control transfers and thus it is also known as the control endpoint. Its main purpose is to configure the device.

Depending on the type of device other endpoints will be available once the device has been configured. For example, a mouse is likely to have an interrupt input endpoint to report mouse events, separate from the control endpoint. In the same way, a disk drive usually has two bulk endpoints (one to write data to the device and one to read data from it) although it may have a single input/output bulk endpoint. Addressing is important here because information supplied by the device may specify that a given endpoint address is the one to use for a particular task. Therefore, endpoint addresses must be exposed to the USB software.

Data may be lost during transfers. As an acknowledge mechanism to recover from this situation, successive packets sent from the controller to a given endpoint use one out of two different values alternatively (the same happens on the other direction, from the device to the controller). This is called the *data toggle* bit and toggles between two values called *data0* and *data1*. How this is codified varies from one controller to another (and from one transfer type to another). Also, some high speed transfers use a

different acknowledgment mechanism to be able to send multiple transfers on a single bus time frame.

The important point for drivers is that due to errors the controller and the device may be out of sync. At this point the device will simply refuse any transfer with the wrong toggle. To continue operating the device, data toggles must be synchronized again. Errors causing a cease of device I/O are called device *stalls*. Recovering from a stall is called *unstalling* the device. But note that what stalls is really an endpoint and not necessarily the entire device.

3. Plan 9's USB overview

The Plan 9 USB software is organized as depicted in figure 2. The kernel includes the *usb*(3) device driver, known as `#u`, in charge of providing I/O to endpoints present on the bus. A suite of user programs provide for everything else. One program, `usbd`, is in charge of bus enumeration. Remaining programs are USB device drivers. Both `usbd` and device drivers employ a library, called the USB library, for common tasks and data structures.

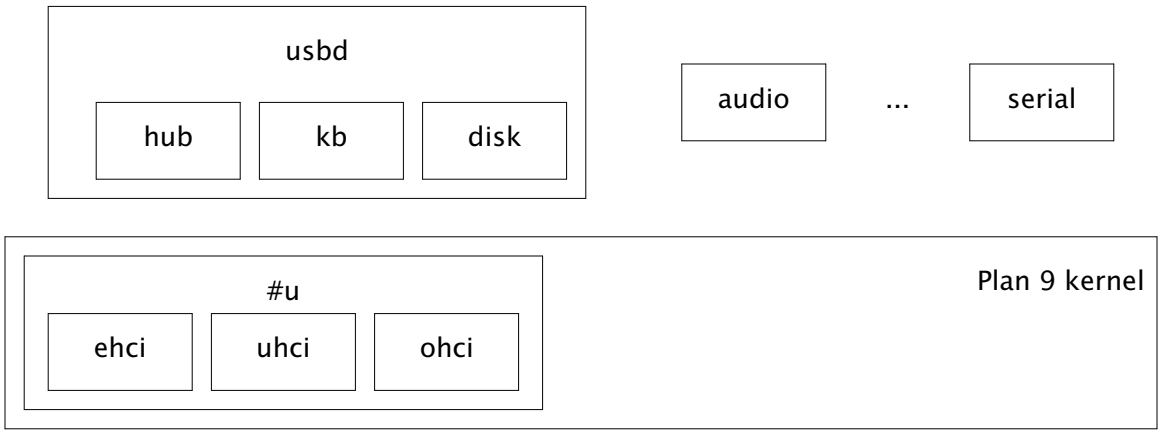

Figure 2. Organization of the USB software.

As figure 2 shows, drivers may be embedded into `usbd` (like `kb` and `disk` in the figure) or kept as separate programs (like `audio` and `serial` in the figure.) The `hub` driver is built into `usbd` and may not be started separately, because hubs play an important role in bus enumeration.

The `#u` device is responsible for initializing the host controllers and abstracting I/O mechanisms used on the bus. Its external interface deviates from the standard in an attempt to simplify things for users and device drivers. The only objects supplied by `#u` are *endpoints*. An endpoint represents a communication channel to a device in the bus (actually, to a bus address).

Figure 3 shows an example file tree provided by `#u`. Each endpoint is represented by a directory that includes two files: `data` and `ctl`, similar to a network connection. The former is used to perform actual I/O and the later is used to issue control requests for the endpoint (not to be confused with USB control transfers). Endpoints are named `ep`*N*`.`*M*, where *N* is the device address and *M* is the endpoint number. Endpoints with name `ep`*N*`.0` are therefore control endpoints.

At boot time `#u` provides one endpoint per root hub. Of course there is no such thing as root hubs, but the user program `usbd` uses these initial endpoints to enumerate the bus. Requests sent through them to query status of ports and to enable them are intercepted by the software and implemented by relying on the host controller interface.

Apart from this feature, there is no software implementation for USB root hubs on Plan 9.

For each new device discovered on the bus `usbd` creates another endpoint in `#u` for its *setup* endpoint. This is used by `usbd` to perform part of the configuration for the device and by the device driver to complete the configuration and, perhaps, to issue other requests to the device. As a result of the configuration, other endpoints are created for the device.

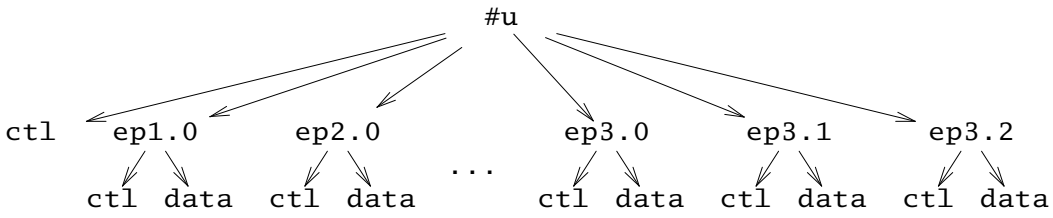

Figure 3: Example USB file tree.

3.1. Kernel drivers

Usb(3) is the driver responsible for providing I/O access to the bus, by providing the endpoint interface described above. Internally, it is built out of a generic device driver that provides the file interface and three different controller drivers that are linked into the former. The interface of the generic part to the rest of the kernel is similar to that in other devices, a `Devtab` structure. The interface between the generic device driver and the three controller-specific drivers deserves some comments.

In general, all three controllers perform the same tasks. They provide a register interface to query and change the status of USB ports on the root hub and use a memory mapped interface with data structures used to perform I/O for all supported transfer types. These data structures may be fancy and include binary trees.

Although much of the code could be shared between controllers (as done in other systems) the particularities of each controller get in the way. For example, how to indicate which data toggle to use for a transfer depends on the transfer type and also on the controller used. Who controls the toggles (the hardware or the software) also depends on the transfer type and on the controller. And the same can be said of many other details. Factoring out the shared code between different controllers would make it necessary to dive often into controller specific code (and data).

Instead of doing so, the Plan 9 `#u` device driver contains only a portable representation for endpoints to support the interface provided to user programs. It is up to each controller to look into the portable representation to configure controller specific data structures to match the portable representation. But for intercepting requests intended for root hubs, all the implementation for I/O is kept on the controller specific part of the code.

This is the portable representation for an endpoint, as kept in `#u`. All endpoints are kept in a global array and the index for each one is kept in the `idx` field. Most of the fields correspond to configuration information retrieved from the device by a user level program and supplied later to the kernel. Other files are used for bookkeeping and to synchronize access to the endpoint.

```
struct Ep
{
        Ref;                    /* one per fid (and per dev ep for ep0s) */

        /* const once inited. */
        int     idx;            /* index in global eps array */
        int     nb;             /* endpoint number in device */
        Hci*    hp;             /* HCI it belongs to */
        Udev*   dev;            /* device for the endpoint */
        Ep*     ep0;            /* control endpoint for its device */

        /* configuration */
        QLock;                  /* protect fields below */
        char*   name;           /* for ep file names at #u/ */
        int     inuse;          /* endpoint is open */
        int     mode;           /* OREAD, OWRITE, or ORDWR */
        int     clrhalt;        /* true if halt was cleared on ep. */
        int     debug;          /* per endpoint debug flag */
        char*   info;           /* for humans to read */
        long    maxpkt;         /* maximum packet size */
        int     ttype;          /* transfer type */
        ulong   load;           /* in µs, for a transfer of maxpkt bytes */
        void*   aux;            /* for controller specific info */
        int     rhrepl;         /* fake root hub replies */
        int     toggle[2];      /* saved toggles (while ep is not in use) */
        long    pollival;       /* poll interval ([µ]frames; intr/iso) */
        long    hz;             /* poll frequency (iso) */
        long    samplesz;       /* sample size (iso) */
        int     ntds;           /* nb. of Tds per µframe */
};
```

The setup endpoint for a device is used to talk to the device and therefore represents the entire device. There is no separate abstraction to represent a USB device. Instead, all endpoints keep in the `ep0` field a link to the control endpoint representing the device. A shared data structure maintains per-device information and can be found linked at the `dev` field of any endpoint. It is declared as follows.

```
struct Udev
{
        int     nb;             /* USB device number */
        int     state;          /* state for the device */
        int     ishub;          /* hubs can allocate devices */
        int     isroot;         /* is a root hub */
        int     speed;          /* Full/Low/High/No -speed */
        int     hub;            /* dev number for the parent hub */
        int     port;           /* port number in the parent hub */
        Ep*     eps[Ndeveps];   /* end points for this device (cached) */
};
```

A device is mostly an address for the device, kept in `nb`, and a series of endpoints, kept in `eps`. Each endpoint has its own address, which matches the index in the per-device endpoint array.

Provided an endpoint, the driver for a controller is responsible for performing I/O on it. To do so, and to provide access to the ports in the controller, the driver must implement the following interface. Currently there are three implementations for EHCI, UHCI, and OHCI.

```
struct Hciimpl
{
        void    *aux;                           /* for controller info */
        void    (*init)(Hci*);                  /* init. controller */
        void    (*dump)(Hci*);                  /* debug */
        void    (*interrupt)(Ureg*, void*);     /* service interrupt */
        void    (*epopen)(Ep*);                 /* prepare ep. for I/O */
        void    (*epclose)(Ep*);                /* terminate I/O on ep. */
        long    (*epread)(Ep*,void*,long);      /* transmit data for ep */
        long    (*epwrite)(Ep*,void*,long);     /* receive data for ep */
        char*   (*seprintep)(char*,char*,Ep*); /* debug */
        int     (*portenable)(Hci*, int, int); /* enable/disable port */
        int     (*portreset)(Hci*, int, int);  /* set/clear port reset */
        int     (*portstatus)(Hci*, int);      /* get port status */
        void    (*debug)(Hci*, int);           /* set/clear debug flag */
};
```

`Init` prepares the controller for operation. `Dump`, `seprintep`, and `debug` are used to control debug diagnostics. `Interrupt` is obviously the interrupt handler for the controller. All other functions are the interesting ones. They implement I/O according for endpoints described by the (portable) data structure shown above and are described in the following sections.

3.2. Hub ports

Ports on root hubs are handled by `portenable`, `portreset`, and `portstatus`. The common `usbread` and `usbwrite` functions intercept requests directed to root hubs to query or adjust the status for their ports. Instead of sending messages through the USB bus, `usbread` and `usbwrite` rely on port handling functions provided by the controller driver. By doing so, `usbd` may remain (mostly) unaware of the difference between root hubs and other hubs.

Instead of performing a full software emulation for root hubs, `#u` includes just a few USB requests (those calling the functions described here). `Usbd` tries not to use other hub features to avoid the need for a full emulation. However, some features are required to configure hubs for operation and thus are used on actual (non-root) hubs. This introduces into `usbd` a few places where the code takes different paths for root and secondary hubs, but the alternative would be to implement a full emulation of root hubs.

Other details necessary in practice to operate on root hubs (e.g., port power configuration) are dealt with in the controller initialization code. All other USB software remains unaware of them.

3.3. Input/Output

The suite provided by `epopen`, `epclose`, `epread`, and `epwrite` provides I/O through USB endpoints. The first two functions prepare the endpoint data structures for I/O and release them, respectively. To avoid resource consumption, an endpoint is kept open only while necessary (while the endpoint data file is open). At all other times, the controller driver keeps no state at all for the endpoint.

During `epopen` the generic description of the endpoint provided by the `Ep` data structure is consulted to configure the actual data structures used by the hardware. Therefore, there is no configuration interface between the generic and the controller specific software (other than the agreed-upon endpoint data structure).

One problem introduced by this scheme is that the physical device may retain configuration, such as protocol data toggles, between successive opens of the same endpoint. But closing an endpoint and opening it again must continue I/O from where it was

left at. This issue is addressed by arranging for epclose to save this information in the portable description of the endpoint, and by making epopen consult the saved state. The Ep structure has an aux field for the controller driver to use. But note that unless the endpoint is in use this field would be null, therefore, saved state is kept within Ep itself.

Epread and Epwrite perform input and output from the bus. In general, they switch on the endpoint transfer type and prepare a transfer of that type, but that is not the case for control transfers.

USB control transfers may require different transfers depending on the request, as said before. To make all control transfers look similar for the user, the interface provided to the user is a single write with the request (and any data sent to the device), perhaps followed by a single read to retrieve data requested by the previous write. The entire transfer is performed during the call to write. This permits retrieving the error status, which is sent by the device at the end of the data transfer, when the request is made. Any data retrieved from the device is kept in memory and given to the user if read is called next (before another write).

Processes using the endpoint must in any case coordinate their requests and thus, performing control transfers in this way does not introduce more race conditions than those existing in the user code.

As an aid, processes not coordinating are kept apart by the DMEXCL permission enforced by #u for endpoint data files. For those who care to look before using, the endpoint control file reports whether the endpoint is in use or not.

Epread and epwrite are responsible for timing out unresponsive devices, raising an error in that case. Arguably, they time out only control and bulk requests. In general these requests complete soon after being issued; Interrupt and isochronous requests do not. Timing out these requests in the kernel in a controlled way helps canceling requests only when the device seems to be not responding. User timeouts might occur with bad timing and confuse the device, which may be simply sending negative acknowledgements while busy doing other things.

Recently we have found some devices where bulk requests may not complete until further device activity. This suggests that only control requests should be timed out even though most devices need timeouts on bulk requests to operate properly with unresponsive devices.

Device drivers like audio and printers may exit after configuring their devices. Using the endpoint data files suffices for them at this point. The problem is that Plan 9 software expects to find these files on conventional names at /dev. As an aid, #u implements a control request capable of giving a second name (at #u) to an endpoint data file.

4. Bus enumeration and hot plugging

Bus enumeration is performed by a single process executing in usbd. This process starts by looking at endpoints existing at boot time. As said before a control endpoint is used to represent a device. The kernel device creates one such endpoint per root hub. From there on, usbd asks for the status for each port in the hub. When a device is connected to a port usbd asks the kernel to create a new endpoint for its control endpoint and, using it, retrieves device descriptors and performs several configuration requests. After configuration, if the device is known to be a hub usbd adds the endpoint to the list of hubs to poll, and polls it. Otherwise, if instructed to do so by its configuration file, usbd starts a new process to execute the device driver for the attached device.

Only one process must use a device at a time. Therefore it is important for usbd and the device driver to coordinate regarding access to USB devices. This problem has been avoided by preventing both usbd and a device driver to own the same device. Figure 4 depicts the typical scenario. In the figure time flows top-down and most of the time will be spent in the small dotted segments, which represent the device while in use. The figure shows only the attachment/detachment process.

Initially, the device is attached to the bus. After that, a port poll done by usbd will see that a device is connected to a port. After configuring the port, by issuing a reset signal and enabling the port, usbd allocates an endpoint (by issuing a control request to the kernel). This becomes the control endpoint for the device and represents the entire device from now on.

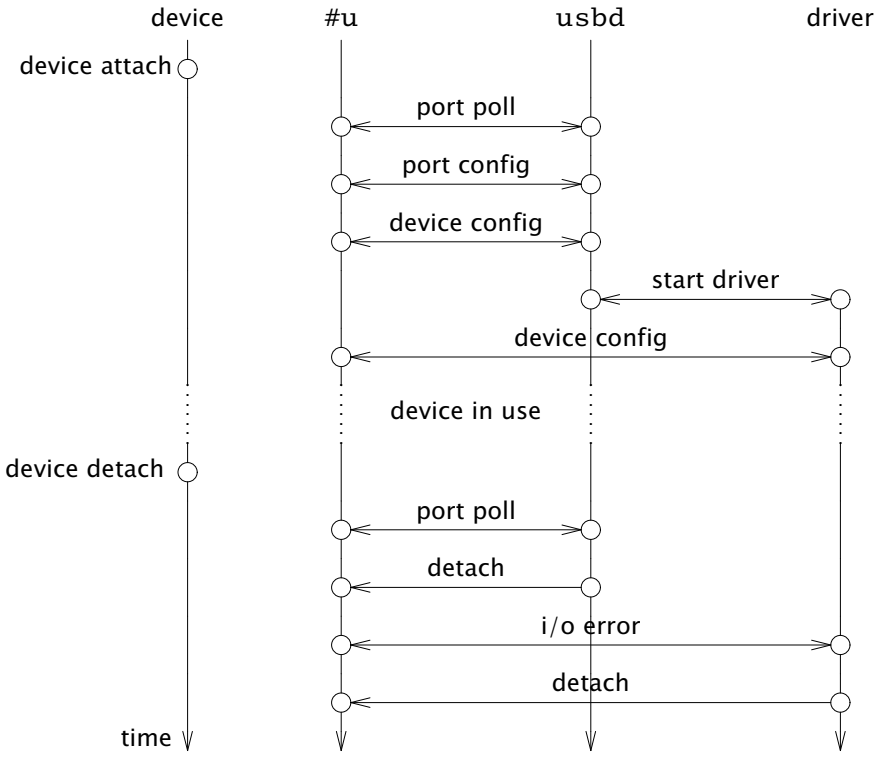

Figure 4. Device attachment and detachment. Time flows down.

Then, usbd configures the device (assigns an address, reads device descriptors reporting the type of device, and activates a configuration for the device). As part of the configuration usbd supplies to the kernel some information about the device according to its descriptors. Most of this information is not used by kernel. It is intended to provide enough information for drivers and users to locate devices, and to learn the relevant data about an endpoint of interest. CSPs, vendor, and product identifiers, and device strings are handled this way. As an example, this is the information supplied for an endpoint:

```
; cat /dev/usb/ep3.0/ctl
enabled control rw speed full maxpkt 64 ival 0 samplesz 0 hz 0 hub 1 port 3 busy
storage csp 0x500608 vid 0x951 did 0x1613 Kingston 'DT 101 II'
;
```

Once the device is configured, usbd is able to start a new process to execute the device driver if instructed to do so. Either case, usbd will *not* touch the device from now on. The driver started is free to open as many endpoints as needed, and to create new endpoints for the device, should it require that, without interference from usbd.

At some point the physical device may be detached. Now one of two things will happen. Either usbd notices that first, as the result of polling the port used by the device; or the device driver learns it first because of I/O errors while using the device. The process noticing the event first, whichever it is, issues a detach control request to the kernel to notify the detachment of the device.

Upon detachment, no further I/O is allowed and all requests (but for a few control requests) return a ''device is detached'' error diagnostic. Thus, if usbd notices first it causes the detachment of the device in the kernel. Any outstanding I/O request is canceled and the driver is notified that the device is detached. If the device driver notices first then it will detach the device and exit on its own (and at some point usbd will see the port as detached).

Another device may be plugged into the port before the next poll done by usbd. This can be detected because the port will not be enabled (although it will have a device connected). In this case we pretend that the port suffered two events: a detach followed by an attach. By doing it this way we can avoid the need for using interrupt endpoints provided by most hubs (real non-root hubs) to report status changes for ports. This saves the software to process such endpoints and also saves their software emulation in the case of root hubs.

Device drivers providing a file system interface are very important regarding hot-plugging of devices. These drivers must not simply exit when a device is detached. When the device driver is embedded into usbd it may be that no process is currently executing the driver's code, because the file system implementation is shared in this case. Thus, upon detachment the file interface is removed from the file tree and no further action is taken.

When the device driver runs as a separate process it should stay running but respond with I/O errors to any further I/O attempt. However, the file tree must be kept alive and responding to permit any client to continue working, if only to exit cleanly. Note that in other case, should the device be bound at /dev, the namespace would become broken whenever /dev is used. When the user notices that the device is no longer working the file system will be unmounted and at that point the driver exits.

5. USB device drivers

A USB device driver on Plan 9 is a user program responsible for configuring and operating a USB device. When usbd notices a new device on the bus it inspects its static configuration to see if a device driver must be started. Should that be the case, usbd creates a new process to execute the driver (either by calling exec for external drivers or by calling the driver's init function for those linked within usbd).

External or standalone drivers must take the burden of locating the devices to drive. By convention a driver will manage all devices known that are not managed by an already executing driver. All the information needed for this purpose (including if the device is in use by another driver) is available by reading the control files of existing control endpoints. That is, files named /dev/usb/ep0.*/ctl.

In practice, a device driver may be started to serve an interface (as part of a driver for the entire device). Thus device drivers must pay attention not only to the device CSP but also to CSPs for device interfaces. On Plan 9 all this information is retrieved by reading control files and most drivers may forget this detail.

Many things done by USB drivers are done by several (when not all) drivers. The corresponding code is kept in a USB library, which is also used by `usbd`. The main data structure in the library is `Dev`. It represents an endpoint provided by `#u`.

After a driver (or `usbd`) identifies a device of interest (i.e., an endpoint) it calls the library to create a `Dev` data structure for it, and spawns a new process using it as an argument, to handle the device. For drivers built into `usbd` it is `usbd` who cares of what has been said so far. For other drivers convenience routines are present in the USB library that can do the job.

The driver entry point is not `main`, but a function called `init` that receives a `Dev` data structure representing the device control endpoint. It is done this way to permit the same code to be used both for the embedded version of the driver and for a standalone version executed as an independent program.

The `Dev` structure refers to an endpoint directory at `#u` and includes file descriptors for the endpoint control and data files:

```
struct Dev
{
        Ref;
        char*     dir;              /* path for the endpoint dir */
        int       id;               /* usb id for device or ep. number */
        int       dfd;              /* descriptor for the data file */
        int       cfd;              /* descriptor for the control file */
        int       maxpkt;           /* cached from usb description */
        Ref       nerrs;            /* number of errors in requests */
        Usbdev*   usb;              /* USB description */
        void*     aux;              /* for the device driver */
        void      (*free)(void*);   /* idem. to release aux */
};
```

Initially only the control file is open. `Usbd` keeps the data file (for the control endpoint) open while configuring the device but closes all descriptors before starting the device driver. The device driver keeps both the control and the data files open most of the time.

This data structure is reference counted to permit an endpoint to go only when no part of the driver is using it. This is important specially for drivers providing a file system interface, which might have outstanding requests while trying to shutdown the device.

All the relevant USB device descriptors must be read from the device to determine which interfaces are present and which endpoints should be used for I/O. This information is gathered by the USB library and placed into a `Usbdev` structure pointed to by `Dev.usb`. Therefore, few device drivers require reading descriptors themselves.

A driver must inspect the USB configuration information to locate interfaces with CSPs that correspond to the functionality provided, and also to locate the endpoints to be used for I/O. Despite help from the USB library all drivers must do this before allocating other endpoints on the device.

In some cases this is not enough. Some devices have descriptors that are specific for them, and are not parsed by the USB library. The library places such unpacked descriptors within the `Usbdev` structure, and the driver is responsible for parsing them if necessary. Chapter 9 of [1] is a good reference for common USB descriptors. Specification documents for particular device classes (or for particular devices) usually describe all device-specific descriptors necessary to drive the hardware.

In many cases drivers may remain unaware of most standard control requests, because the library has functions that do most of the work (configuring the device and unstalling endpoints upon errors). Device specific requests on the other hand must

always be issued by the driver, who knows when and how they must be done. The only thing the library can do is to provide a general purpose `usbcmd` utility function and some symbols to help building such requests.

5.1. Embedded drivers

For most drivers there is almost no difference between the embedded version and the standalone one. As said above, either `usbd` or a `main` function for the driver takes care of locating devices of interest and calling the driver's `init` function for each device found. A side effect of preparing a driver for embedding is that its `main` function may be borrowed almost entirely from another driver.

All other code is kept into a USB driver library that is linked to `usbd` (and also to the standalone version of the driver). This library contains drivers and is not to be confused with the USB library that is a convenience for writing drivers.

An important point for embedded drivers is that they must be careful not to rely on static storage and must be programmed to be reentrant. This is necessary because several instances of the driver may be created at different times to handle different devices, and they should rely just on the `Dev` structure for the control endpoint to operate the device (Of course further endpoints and other data structures can be created but none of them may be static storage).

Reference counting is also important for embedded drivers. A driver should go when the `Dev` reference supplied to its entry point goes down to zero. Initially, the driver's `init` function is given a `Dev` that counts one reference. When the reference goes away the endpoint is closed and the device released. If this happens due to an error, the driver issues a `detach` control request to the endpoint before releasing the reference, causing the endpoint to be collected when the last reference goes away. Drivers may install their own auxiliary structure and a free routine into `Dev`, as an aid to the implementation and also to support clean exits.

5.2. File system interface

The file interface provided by some USB drivers is important because it is the conventional interface for the corresponding devices on Plan 9. But it is also important because it is a key piece of the hot plugging mechanism.

If a device is removed and the file system interface for its driver simply responds with I/O errors to any further request, or aborts, `/dev` may become broken in the name space where this happens.

As devices come and go it is important to be able to see the file interface for devices adjust accordingly. For example, a directory named `/dev/sdU3.0`, corresponding to a disk for the device number 3, is expected to be there (albeit in a detached state) upon disk disconnection; as long as there is someone using the file interface. Only actual data transfers are reported as failed with I/O errors. Directories still work. When the last user of this file is gone, the directory silently disappears. Otherwise, inserting and removing a disk several times would lead to multiple (unused) directories and device numbers would become large numbers hard to remember.

The venerable 9P library is not used by USB software because of the requirements mentioned before. Instead, a small and specialized USB file system library is included in the standard USB library.

The interface for a file server is defined by the following data structure:

```
struct Usbfs
{
        char      name[Namesz];
        uvlong    qid;
        Dev*      dev;
        void*     aux;

        int       (*walk)(Usbfs *fs, Fid *f, char *name);
        void      (*clone)(Usbfs *fs, Fid *of, Fid *nf);
        void      (*clunk)(Usbfs *fs, Fid *f);
        int       (*open)(Usbfs *fs, Fid *f, int mode);
        long      (*read)(Usbfs *fs, Fid *f, void *data, long count, vlong offset);
        long      (*write)(Usbfs *fs, Fid*f, void *data, long count, vlong offset);
        int       (*stat)(Usbfs *fs, Qid q, Dir *d);
        void      (*end)(Usbfs *fs);
};
```

It can be seen how it includes both a `name` for the file tree and a `Dev` reference, pointing to the device behind the file system interface. Only operations named in `Usbfs` have to be implemented (some of them may left null if not of interest). They work as expected given their names.

Of these operations, `open`, `read`, and `write` are executed concurrently by the library using different processes. Thus an implementation may block if necessary without blocking the entire interface for the device (or worse, the entire set of drivers linked into `usbd`). Remaining operations must terminate promptly and are supported by a single, per client, process started by the library.

Despite concurrency there is no flush mechanism. Upon a flush the ongoing operation is left alone (only its result is ignored). This is not a problem in the case of USB drivers. Some flushes happen after device errors (e.g., because an application or a user interrupts a request). They find the device in a detached state as explained before. As a side effect, the detach operation interrupts any further I/O, making flush unnecessary in this case. Other flushes are legitimate interrupt requests issued by the user, and we simple let the outstanding operations complete and be ignored when they do.

The file system interface is designed to stack file systems. The library provides a file system implementation called `usbdir` that implements a single directory and accepts calls to `usbadd` and `usbdel` functions to plug other file systems into this directory. The `name` field in `Usbfs` is used as the name of the file tree when stacked into this directory, and the `qid` field is used to multiplex the Qid space among different file trees. This is the mechanism used to provide the same file interface, some times within an entire tree provided by `usbd`, some times as a solitary tree provided by the device.

6. Status and future work

The software described in this paper is operational. However, there are several issues that must be addressed. Input Isochronous streams for OHCI controllers are not yet implemented. As said before, bulk transfers should perhaps not be timed out by the kernel. The ethernet driver seems to have problems and must be fixed. Also, due to hardware unavailability, version 3.0 of the bus is not supported. In general, the source must be reviewed to undergo some cleaning.

In the future these issues will be addressed and more drivers will be written for other devices present on the bus.

There is plenty of room for performance tuning and optimization. The only optimization introduced so far is using embedded buffers for transfer descriptors. Data structures used to maintain controller specific transfer descriptors include a few bytes of

data storage. For transfers small enough (which are common) no data buffer is allocated and the embedded buffer is used instead.

7. Acknowledgments

The USB software described here is the result of a collective and secret effort. We are grateful to the anonymous authors of the previous USB software (Charles Forsyth and others). We are grateful to Plan 9 developers for the system and to 9fans for testing the software. Geoff Collyer proved to be invaluable to make bugs show up, so they could be debugged. Gorka Guardiola implemented the USB serial device driver and support for USB KNX devices. Cinap Lenrek implemented the old ethernet driver and fixed the new one to make it work. Russ Cox suggested the mechanism to provide named files and avoid keeping a driver running on simple cases. Enrique Soriano added support for a USB toy, a demon with wings that I expect to use as a mail biff. Erik Quanstrom hunted a bug in the USB disk. Thank you all.

References

1. Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC and Philips, USB 2.0 Specification, 2000.
2. Compaq, Microsoft and N. Semiconductor, OpenHCI - Open Host Controller Interface Specification for USB, 1995.
3. Intel, Enhanced Host Controller Interface Specification for Universal Serial Bus, Revision 1.0, 2002.
4. Intel, Universal Host Controller Interface Design Guide, Revision 1.1, 1996.