* Introduction

The name "usbmon" in lowercase refers to a facility in kernel which is used to collect traces of I/O on the USB bus. This function is analogous to a packet socket used by network monitoring tools such as tcpdump(1) or Ethereal. Similarly, it is expected that a tool such as usbdump or USBMon (with uppercase letters) is used to examine raw traces produced by usbmon.

The usbmon reports requests made by peripheral-specific drivers to Host Controller Drivers (HCD). So, if HCD is buggy, the traces reported by usbmon may not correspond to bus transactions precisely. This is the same situation as with tcpdump.

* How to use usbmon to collect raw text traces

Unlike the packet socket, usbmon has an interface which provides traces in a text format. This is used for two purposes. First, it serves as a common trace exchange format for tools while more sophisticated formats are finalized. Second, humans can read it in case tools are not available.

To collect a raw text trace, execute following steps.

1. Prepare

Mount debugfs (it has to be enabled in your kernel configuration), and load the usbmon module (if built as module). The second step is skipped if usbmon is built into the kernel.

```
# mount -t debugfs none_debugs /sys/kernel/debug
# modprobe usbmon
#
```

Verify that bus sockets are present.

```
# ls /sys/kernel/debug/usb/usbmon
0s  0u  1s  1t  1u  2s  2t  2u  3s  3t  3u  4s  4t  4u
#
```

Now you can choose to either use the socket '0u' (to capture packets on all buses), and skip to step #3, or find the bus used by your device with step #2. This allows to filter away annoying devices that talk continuously.

2. Find which bus connects to the desired device

Run "cat /proc/bus/usb/devices", and find the T-line which corresponds to the device. Usually you do it by looking for the vendor string. If you have many similar devices, unplug one and compare two /proc/bus/usb/devices outputs. The T-line will have a bus number. Example:

```
T:  Bus=03 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#=  2 Spd=12  MxCh= 0
D:  Ver= 1.10 Cls=00(>ifc ) Sub=00 Prot=00 MxPS= 8 #Cfgs=  1
P:  Vendor=0557 ProdID=2004 Rev= 1.00
S:  Manufacturer=ATEN
S:  Product=UC100KM V2.00
```

Bus=03 means it's bus 3.

3. Start 'cat'

```
# cat /sys/kernel/debug/usb/usbmon/3u > /tmp/1.mon.out
```

to listen on a single bus, otherwise, to listen on all buses, type:

```
# cat /sys/kernel/debug/usb/usbmon/0u > /tmp/1.mon.out
```

This process will be reading until killed. Naturally, the output can be redirected to a desirable location. This is preferred, because it is going to be quite long.

4. Perform the desired operation on the USB bus

This is where you do something that creates the traffic: plug in a flash key, copy files, control a webcam, etc.

5. Kill cat

Usually it's done with a keyboard interrupt (Control-C).

At this point the output file (/tmp/1.mon.out in this example) can be saved, sent by e-mail, or inspected with a text editor. In the last case make sure that the file size is not excessive for your favourite editor.

* Raw text data format

Two formats are supported currently: the original, or '1t' format, and the '1u' format. The '1t' format is deprecated in kernel 2.6.21. The '1u' format adds a few fields, such as ISO frame descriptors, interval, etc. It produces slightly longer lines, but otherwise is a perfect superset of '1t' format.

If it is desired to recognize one from the other in a program, look at the "address" word (see below), where '1u' format adds a bus number. If 2 colons are present, it's the '1t' format, otherwise '1u'.

Any text format data consists of a stream of events, such as URB submission, URB callback, submission error. Every event is a text line, which consists of whitespace separated words. The number or position of words may depend on the event type, but there is a set of words, common for all types.

Here is the list of words, from left to right:

- URB Tag. This is used to identify URBs, and is normally an in-kernel address of the URB structure in hexadecimal, but can be a sequence number or any other unique string, within reason.

- Timestamp in microseconds, a decimal number. The timestamp's resolution depends on available clock, and so it can be much worse than a microsecond (if the implementation uses jiffies, for example).

- Event Type. This type refers to the format of the event, not URB type. Available types are: S - submission, C - callback, E - submission error.

- "Address" word (formerly a "pipe"). It consists of four fields, separated by colons: URB type and direction, Bus number, Device address, Endpoint number. Type and direction are encoded with two bytes in the following manner:
```
    Ci Co   Control input and output
    Zi Zo   Isochronous input and output
    Ii Io   Interrupt input and output
    Bi Bo   Bulk input and output
```
  Bus number, Device address, and Endpoint are decimal numbers, but they may have leading zeros, for the sake of human readers.

- URB Status word. This is either a letter, or several numbers separated by colons: URB status, interval, start frame, and error count. Unlike the "address" word, all fields save the status are optional. Interval is printed only for interrupt and isochronous URBs. Start frame is printed only for isochronous URBs. Error count is printed only for isochronous callback events.

  The status field is a decimal number, sometimes negative, which represents a "status" field of the URB. This field makes no sense for submissions, but is present anyway to help scripts with parsing. When an error occurs, the field contains the error code.

  In case of a submission of a Control packet, this field contains a Setup Tag instead of an group of numbers. It is easy to tell whether the Setup Tag is present because it is never a number. Thus if scripts find a set of numbers in this word, they proceed to read Data Length (except for isochronous URBs). If they find something else, like a letter, they read the setup packet before reading the Data Length or isochronous descriptors.

- Setup packet, if present, consists of 5 words: one of each for bmRequestType, bRequest, wValue, wIndex, wLength, as specified by the USB Specification 2.0. These words are safe to decode if Setup Tag was 's'. Otherwise, the setup packet was present, but not captured, and the fields contain filler.

- Number of isochronous frame descriptors and descriptors themselves. If an Isochronous transfer event has a set of descriptors, a total number of them in an URB is printed first, then a word per descriptor, up to a total of 5. The word consists of 3 colon-separated decimal numbers for status, offset, and length respectively. For submissions, initial length is reported. For callbacks, actual length is reported.

- Data Length. For submissions, this is the requested length. For callbacks, this is the actual length.

- Data tag. The usbmon may not always capture data, even if length is nonzero. The data words are present only if this tag is '='.

- Data words follow, in big endian hexadecimal format. Notice that they are not machine words, but really just a byte stream split into words to make it easier to read. Thus, the last word may contain from one to four bytes. The length of collected data is limited and can be less than the data length report in Data Length word.

Here is an example of code to read the data stream in a well known programming language:

```
class ParsedLine {
        int data_len;           /* Available length of data */
        byte data[];

        void parseData(StringTokenizer st) {
                int availwords = st.countTokens();
                data = new byte[availwords * 4];
                data_len = 0;
                while (st.hasMoreTokens()) {
                        String data_str = st.nextToken();
                        int len = data_str.length() / 2;
                        int i;
                        int b;  // byte is signed, apparently?! XXX
                        for (i = 0; i < len; i++) {
                                // data[data_len] = Byte.parseByte(
                                //     data_str.substring(i*2, i*2 + 2),
                                //     16);
                                b = Integer.parseInt(
                                     data_str.substring(i*2, i*2 + 2),
                                     16);
                                if (b >= 128)
                                        b *= -1;
                                data[data_len] = (byte) b;
                                data_len++;
                        }
                }
        }
}
```

Examples:

An input control transfer to get a port status.

```
d5ea89a0 3575914555 S Ci:1:001:0 s a3 00 0000 0003 0004 4 <
d5ea89a0 3575914560 C Ci:1:001:0 0 4 = 01050000
```

An output bulk transfer to send a SCSI command 0x5E in a 31-byte Bulk wrapper to a storage device at address 5:

```
dd65f0e8 4128379752 S Bo:1:005:2 -115 31 = 55534243 5e000000 00000000 00000600
00000000 00000000 00000000 000000
dd65f0e8 4128379808 C Bo:1:005:2 0 31 >
```

* Raw binary format and API

The overall architecture of the API is about the same as the one above, only the events are delivered in binary format. Each event is sent in the following structure (its name is made up, so that we can refer to it):

```
struct usbmon_packet {
        u64 id;                 /*  0: URB ID - from submission to callback */
        unsigned char type;     /*  8: Same as text; extensible. */
        unsigned char xfer_type; /*    ISO (0), Intr, Control, Bulk (3) */
        unsigned char epnum;    /*     Endpoint number and transfer direction */
        unsigned char devnum;   /*     Device address */
```

```
        u16 busnum;             /* 12: Bus number */
        char flag_setup;        /* 14: Same as text */
        char flag_data;         /* 15: Same as text; Binary zero is OK. */
        s64 ts_sec;             /* 16: gettimeofday */
        s32 ts_usec;            /* 24: gettimeofday */
        int status;             /* 28: */
        unsigned int length;    /* 32: Length of data (submitted or actual) */
        unsigned int len_cap;   /* 36: Delivered length */
        union {                 /* 40: */
                unsigned char setup[SETUP_LEN]; /* Only for Control S-type */
                struct iso_rec {                /* Only for ISO */
                        int error_count;
                        int numdesc;
                } iso;
        } s;
        int interval;           /* 48: Only for Interrupt and ISO */
        int start_frame;        /* 52: For ISO */
        unsigned int xfer_flags; /* 56: copy of URB's transfer_flags */
        unsigned int ndesc;     /* 60: Actual number of ISO descriptors */
};                              /* 64 total length */
```

These events can be received from a character device by reading with read(2), with an ioctl(2), or by accessing the buffer with mmap. However, read(2) only returns first 48 bytes for compatibility reasons.

The character device is usually called /dev/usbmonN, where N is the USB bus number. Number zero (/dev/usbmon0) is special and means "all buses". Note that specific naming policy is set by your Linux distribution.

If you create /dev/usbmon0 by hand, make sure that it is owned by root and has mode 0600. Otherwise, unpriviledged users will be able to snoop keyboard traffic.

The following ioctl calls are available, with MON_IOC_MAGIC 0x92:

 MON_IOCQ_URB_LEN, defined as _IO(MON_IOC_MAGIC, 1)

This call returns the length of data in the next event. Note that majority of events contain no data, so if this call returns zero, it does not mean that no events are available.

 MON_IOCG_STATS, defined as _IOR(MON_IOC_MAGIC, 3, struct mon_bin_stats)

The argument is a pointer to the following structure:

```
struct mon_bin_stats {
        u32 queued;
        u32 dropped;
};
```

The member "queued" refers to the number of events currently queued in the buffer (and not to the number of events processed since the last reset).

The member "dropped" is the number of events lost since the last call to MON_IOCG_STATS.

MON_IOCT_RING_SIZE, defined as _IO(MON_IOC_MAGIC, 4)

This call sets the buffer size. The argument is the size in bytes. The size may be rounded down to the next chunk (or page). If the requested size is out of [unspecified] bounds for this kernel, the call fails with -EINVAL.

MON_IOCQ_RING_SIZE, defined as _IO(MON_IOC_MAGIC, 5)

This call returns the current size of the buffer in bytes.

MON_IOCX_GET, defined as _IOW(MON_IOC_MAGIC, 6, struct mon_get_arg)
MON_IOCX_GETX, defined as _IOW(MON_IOC_MAGIC, 10, struct mon_get_arg)

These calls wait for events to arrive if none were in the kernel buffer, then return the first event. The argument is a pointer to the following structure:

```
struct mon_get_arg {
	struct usbmon_packet *hdr;
	void *data;
	size_t alloc;		/* Length of data (can be zero) */
};
```

Before the call, hdr, data, and alloc should be filled. Upon return, the area pointed by hdr contains the next event structure, and the data buffer contains the data, if any. The event is removed from the kernel buffer.

The MON_IOCX_GET copies 48 bytes to hdr area, MON_IOCX_GETX copies 64 bytes.

MON_IOCX_MFETCH, defined as _IOWR(MON_IOC_MAGIC, 7, struct mon_mfetch_arg)

This ioctl is primarily used when the application accesses the buffer with mmap(2). Its argument is a pointer to the following structure:

```
struct mon_mfetch_arg {
	uint32_t *offvec;	/* Vector of events fetched */
	uint32_t nfetch;	/* Number of events to fetch (out: fetched) */
	uint32_t nflush;	/* Number of events to flush */
};
```

The ioctl operates in 3 stages.

First, it removes and discards up to nflush events from the kernel buffer. The actual number of events discarded is returned in nflush.

Second, it waits for an event to be present in the buffer, unless the pseudo-device is open with O_NONBLOCK.

Third, it extracts up to nfetch offsets into the mmap buffer, and stores them into the offvec. The actual number of event offsets is stored into the nfetch.

MON_IOCH_MFLUSH, defined as _IO(MON_IOC_MAGIC, 8)

This call removes a number of events from the kernel buffer. Its argument is the number of events to remove. If the buffer contains fewer events than requested, all events present are removed, and no error is reported. This works when no events are available too.

FIONBIO

The ioctl FIONBIO may be implemented in the future, if there's a need.

In addition to ioctl(2) and read(2), the special file of binary API can be polled with select(2) and poll(2). But lseek(2) does not work.

* Memory-mapped access of the kernel buffer for the binary API

The basic idea is simple:

To prepare, map the buffer by getting the current size, then using mmap(2). Then, execute a loop similar to the one written in pseudo-code below:

```
   struct mon_mfetch_arg fetch;
   struct usbmon_packet *hdr;
   int nflush = 0;
   for (;;) {
      fetch.offvec = vec; // Has N 32-bit words
      fetch.nfetch = N;   // Or less than N
      fetch.nflush = nflush;
      ioctl(fd, MON_IOCX_MFETCH, &fetch);   // Process errors, too
      nflush = fetch.nfetch;       // This many packets to flush when done
      for (i = 0; i < nflush; i++) {
         hdr = (struct ubsmon_packet *) &mmap_area[vec[i]];
         if (hdr->type == '@')     // Filler packet
            continue;
         caddr_t data = &mmap_area[vec[i]] + 64;
         process_packet(hdr, data);
      }
   }
```

Thus, the main idea is to execute only one ioctl per N events.

Although the buffer is circular, the returned headers and data do not cross the end of the buffer, so the above pseudo-code does not need any gathering.