Linux Usb Documentation

The kernel development community

Jun 10, 2024

CONTENTS

1 Linux ACM driver v0.16 1

2 Authorizing (or not) your USB devices to connect to the system 5

3 ChipIdea Highspeed Dual Role Controller Driver 9

4 DWC3 driver 13

5 EHCI driver 15

6 How FunctionFS works 19

7 Linux USB gadget configured through configfs 21

8 Linux USB HID gadget driver 29

9 Multifunction Composite Gadget 39

10 Linux USB Printer Gadget Driver 43

11 Linux Gadget Serial Driver v2.0 53

12 Linux UVC Gadget Driver 59

13 Gadget Testing 67

14 Infinity Usb Unlimited Readme 89

15 Mass Storage Gadget (MSG) 91

16 USB 7-Segment Numeric Display 95

17 mtouchusb driver 97

18 OHCI 99

19 USB Raw Gadget 101

20 USB/IP protocol 103

21 usbmon 113

22 USB serial 121

23 USB references **131**

24 Linux CDC ACM inf **133**

25 Linux inf **137**

26 USB devfs drop permissions source **139**

27 Credits **143**

CHAPTER

ONE

LINUX ACM DRIVER V0.16

Sponsored by SuSE

1.1 0. Disclaimer

Should you need to contact me, the author, you can do so either by e-mail - mail your message to <vojtech@suse.cz>, or by paper mail: Vojtech Pavlik, Ucitelska 1576, Prague 8, 182 00 Czech Republic

For your convenience, the GNU General Public License version 2 is included in the package: See the file COPYING.

1.2 1. Usage

The drivers/usb/class/cdc-acm.c drivers works with USB modems and USB ISDN terminal adapters that conform to the Universal Serial Bus Communication Device Class Abstract Control Model (USB CDC ACM) specification.

Many modems do, here is a list of those I know of:

- 3Com OfficeConnect 56k
- 3Com Voice FaxModem Pro
- 3Com Sportster
- MultiTech MultiModem 56k
- Zoom 2986L FaxModem

- Compaq 56k FaxModem
- ELSA Microlink 56k

I know of one ISDN TA that does work with the acm driver:

- 3Com USR ISDN Pro TA

Some cell phones also connect via USB. I know the following phones work:

- SonyEricsson K800i

Unfortunately many modems and most ISDN TAs use proprietary interfaces and thus won't work with this drivers. Check for ACM compliance before buying.

To use the modems you need these modules loaded:

```
usbcore.ko
uhci-hcd.ko ohci-hcd.ko or ehci-hcd.ko
cdc-acm.ko
```

After that, the modem[s] should be accessible. You should be able to use minicom, ppp and mgetty with them.

1.3 2. Verifying that it works

The first step would be to check /sys/kernel/debug/usb/devices, it should look like this:

```
T:  Bus=01 Lev=00 Prnt=00 Port=00 Cnt=00 Dev#=  1 Spd=12  MxCh= 2
B:  Alloc=  0/900 us ( 0%), #Int=  0, #Iso=  0
D:  Ver= 1.00 Cls=09(hub  ) Sub=00 Prot=00 MxPS= 8 #Cfgs=  1
P:  Vendor=0000 ProdID=0000 Rev= 0.00
S:  Product=USB UHCI Root Hub
S:  SerialNumber=6800
C:* #Ifs= 1 Cfg#= 1 Atr=40 MxPwr=  0mA
I:  If#= 0 Alt= 0 #EPs= 1 Cls=09(hub  ) Sub=00 Prot=00 Driver=hub
E:  Ad=81(I) Atr=03(Int.) MxPS=   8 Ivl=255ms
T:  Bus=01 Lev=01 Prnt=01 Port=01 Cnt=01 Dev#=  2 Spd=12  MxCh= 0
D:  Ver= 1.00 Cls=02(comm.) Sub=00 Prot=00 MxPS= 8 #Cfgs=  2
P:  Vendor=04c1 ProdID=008f Rev= 2.07
S:  Manufacturer=3Com Inc.
S:  Product=3Com U.S. Robotics Pro ISDN TA
S:  SerialNumber=UFT53A49BVT7
C:  #Ifs= 1 Cfg#= 1 Atr=60 MxPwr=  0mA
I:  If#= 0 Alt= 0 #EPs= 3 Cls=ff(vend.) Sub=ff Prot=ff Driver=acm
E:  Ad=85(I) Atr=02(Bulk) MxPS=  64 Ivl=  0ms
E:  Ad=04(O) Atr=02(Bulk) MxPS=  64 Ivl=  0ms
E:  Ad=81(I) Atr=03(Int.) MxPS=  16 Ivl=128ms
C:* #Ifs= 2 Cfg#= 2 Atr=60 MxPwr=  0mA
I:  If#= 0 Alt= 0 #EPs= 1 Cls=02(comm.) Sub=02 Prot=01 Driver=acm
E:  Ad=81(I) Atr=03(Int.) MxPS=  16 Ivl=128ms
I:  If#= 1 Alt= 0 #EPs= 2 Cls=0a(data ) Sub=00 Prot=00 Driver=acm
E:  Ad=85(I) Atr=02(Bulk) MxPS=  64 Ivl=  0ms
E:  Ad=04(O) Atr=02(Bulk) MxPS=  64 Ivl=  0ms
```

The presence of these three lines (and the Cls= ‘comm’ and ‘data’ classes) is important, it means it’s an ACM device. The Driver=acm means the acm driver is used for the device. If you see only Cls=ff(vend.) then you’re out of luck, you have a device with vendor specific-interface:

```
D:  Ver= 1.00 Cls=02(comm.) Sub=00 Prot=00 MxPS= 8 #Cfgs=  2
I:  If#= 0 Alt= 0 #EPs= 1 Cls=02(comm.) Sub=02 Prot=01 Driver=acm
I:  If#= 1 Alt= 0 #EPs= 2 Cls=0a(data ) Sub=00 Prot=00 Driver=acm
```

In the system log you should see:

```
usb.c: USB new device connect, assigned device number 2
usb.c: kmalloc IF c7691fa0, numif 1
usb.c: kmalloc IF c7b5f3e0, numif 2
usb.c: skipped 4 class/vendor specific interface descriptors
usb.c: new device strings: Mfr=1, Product=2, SerialNumber=3
usb.c: USB device number 2 default language ID 0x409
Manufacturer: 3Com Inc.
Product: 3Com U.S. Robotics Pro ISDN TA
SerialNumber: UFT53A49BVT7
acm.c: probing config 1
acm.c: probing config 2
ttyACM0: USB ACM device
acm.c: acm_control_msg: rq: 0x22 val: 0x0 len: 0x0 result: 0
acm.c: acm_control_msg: rq: 0x20 val: 0x0 len: 0x7 result: 7
usb.c: acm driver claimed interface c7b5f3e0
usb.c: acm driver claimed interface c7b5f3f8
usb.c: acm driver claimed interface c7691fa0
```

If all this seems to be OK, fire up minicom and set it to talk to the ttyACM device and try typing ‘at’. If it responds with ‘OK’, then everything is working.

CHAPTER

TWO

AUTHORIZING (OR NOT) YOUR USB DEVICES TO CONNECT TO THE SYSTEM

This feature allows you to control if a USB device can be used (or not) in a system. This feature will allow you to implement a lock-down of USB devices, fully controlled by user space.

As of now, when a USB device is connected it is configured and its interfaces are immediately made available to the users. With this modification, only if root authorizes the device to be configured will then it be possible to use it.

2.1 Usage

Authorize a device to connect:

```
$ echo 1 > /sys/bus/usb/devices/DEVICE/authorized
```

De-authorize a device:

```
$ echo 0 > /sys/bus/usb/devices/DEVICE/authorized
```

Set new devices connected to hostX to be deauthorized by default (ie: lock down):

```
$ echo 0 > /sys/bus/usb/devices/usbX/authorized_default
```

Remove the lock down:

```
$ echo 1 > /sys/bus/usb/devices/usbX/authorized_default
```

By default, all USB devices are authorized. Writing “2” to the authorized_default attribute causes the kernel to authorize by default only devices connected to internal USB ports.

2.1.1 Example system lockdown (lame)

Imagine you want to implement a lockdown so only devices of type XYZ can be connected (for example, it is a kiosk machine with a visible USB port):

```
boot up
rc.local ->

 for host in /sys/bus/usb/devices/usb*
 do
    echo 0 > $host/authorized_default
 done
```

Hookup an script to udev, for new USB devices:

```
if device_is_my_type $DEV
then
  echo 1 > $device_path/authorized
done
```

Now, device_is_my_type() is where the juice for a lockdown is. Just checking if the class, type and protocol match something is the worse security verification you can make (or the best, for someone willing to break it). If you need something secure, use crypto and Certificate Authentication or stuff like that. Something simple for an storage key could be:

```
function device_is_my_type()
{
  echo 1 > authorized          # temporarily authorize it
                               # FIXME: make sure none can mount it
  mount DEVICENODE /mntpoint
  sum=$(md5sum /mntpoint/.signature)
  if [ $sum = $(cat /etc/lockdown/keysum) ]
  then
       echo "We are good, connected"
       umount /mntpoint
       # Other stuff so others can use it
  else
       echo 0 > authorized
  fi
}
```

Of course, this is lame, you'd want to do a real certificate verification stuff with PKI, so you don't depend on a shared secret, etc, but you get the idea. Anybody with access to a device gadget kit can fake descriptors and device info. Don't trust that. You are welcome.

2.1.2 Interface authorization

There is a similar approach to allow or deny specific USB interfaces. That allows to block only a subset of an USB device.

Authorize an interface:

```
$ echo 1 > /sys/bus/usb/devices/INTERFACE/authorized
```

Deauthorize an interface:

```
$ echo 0 > /sys/bus/usb/devices/INTERFACE/authorized
```

The default value for new interfaces on a particular USB bus can be changed, too.

Allow interfaces per default:

```
$ echo 1 > /sys/bus/usb/devices/usbX/interface_authorized_default
```

Deny interfaces per default:

```
$ echo 0 > /sys/bus/usb/devices/usbX/interface_authorized_default
```

Per default the interface_authorized_default bit is 1. So all interfaces would authorized per default.

Note:
If a deauthorized interface will be authorized so the driver probing must be triggered manually by writing INTERFACE to /sys/bus/usb/drivers_probe

For drivers that need multiple interfaces all needed interfaces should be authorized first. After that the drivers should be probed. This avoids side effects.

CHAPTER

THREE

CHIPIDEA HIGHSPEED DUAL ROLE CONTROLLER DRIVER

3.1 1. How to test OTG FSM(HNP and SRP)

To show how to demo OTG HNP and SRP functions via sys input files with 2 Freescale i.MX6Q sabre SD boards.

3.2 1.1 How to enable OTG FSM

3.2.1 1.1.1 Select CONFIG_USB_OTG_FSM in menuconfig, rebuild kernel

Image and modules. If you want to check some internal variables for otg fsm, mount debugfs, there are 2 files which can show otg fsm variables and some controller registers value:

```
cat /sys/kernel/debug/ci_hdrc.0/otg
cat /sys/kernel/debug/ci_hdrc.0/registers
```

3.2.2 1.1.2 Add below entries in your dts file for your controller node

```
otg-rev = <0x0200>;
adp-disable;
```

3.3 1.2 Test operations

1) Power up 2 Freescale i.MX6Q sabre SD boards with gadget class driver loaded (e.g. g_mass_storage).
2) Connect 2 boards with usb cable: one end is micro A plug, the other end is micro B plug.

   The A-device (with micro A plug inserted) should enumerate B-device.
3) Role switch

   On B-device:

   ```
   echo 1 > /sys/bus/platform/devices/ci_hdrc.0/inputs/b_bus_req
   ```

   B-device should take host role and enumerate A-device.

4) A-device switch back to host.

   On B-device:

```
echo 0 > /sys/bus/platform/devices/ci_hdrc.0/inputs/b_bus_req
```

   or, by introducing HNP polling, B-Host can know when A-peripheral wishes to be in the host role, so this role switch also can be triggered in A-peripheral side by answering the polling from B-Host. This can be done on A-device:

```
echo 1 > /sys/bus/platform/devices/ci_hdrc.0/inputs/a_bus_req
```

   A-device should switch back to host and enumerate B-device.

5) Remove B-device (unplug micro B plug) and insert again in 10 seconds; A-device should enumerate B-device again.

6) Remove B-device (unplug micro B plug) and insert again after 10 seconds; A-device should NOT enumerate B-device.

   if A-device wants to use bus:

   On A-device:

```
echo 0 > /sys/bus/platform/devices/ci_hdrc.0/inputs/a_bus_drop
echo 1 > /sys/bus/platform/devices/ci_hdrc.0/inputs/a_bus_req
```

   if B-device wants to use bus:

   On B-device:

```
echo 1 > /sys/bus/platform/devices/ci_hdrc.0/inputs/b_bus_req
```

7) A-device power down the bus.

   On A-device:

```
echo 1 > /sys/bus/platform/devices/ci_hdrc.0/inputs/a_bus_drop
```

   A-device should disconnect with B-device and power down the bus.

8) B-device does data pulse for SRP.

   On B-device:

```
echo 1 > /sys/bus/platform/devices/ci_hdrc.0/inputs/b_bus_req
```

   A-device should resume usb bus and enumerate B-device.

3.4 1.3 Reference document

“On-The-Go and Embedded Host Supplement to the USB Revision 2.0 Specification July 27, 2012 Revision 2.0 version 1.1a”

3.5 2. How to enable USB as system wakeup source

Below is the example for how to enable USB as system wakeup source on an imx6 platform.

2.1 Enable core’s wakeup:

```
echo enabled > /sys/bus/platform/devices/ci_hdrc.0/power/wakeup
```

2.2 Enable glue layer’s wakeup:

```
echo enabled > /sys/bus/platform/devices/2184000.usb/power/wakeup
```

2.3 Enable PHY’s wakeup (optional):

```
echo enabled > /sys/bus/platform/devices/20c9000.usbphy/power/wakeup
```

2.4 Enable roothub’s wakeup:

```
echo enabled > /sys/bus/usb/devices/usb1/power/wakeup
```

2.5 Enable related device’s wakeup:

```
echo enabled > /sys/bus/usb/devices/1-1/power/wakeup
```

If the system has only one usb port, and you want usb wakeup at this port, you can use the below script to enable usb wakeup:

```
for i in $(find /sys -name wakeup | grep usb);do echo enabled > $i;done;
```

CHAPTER

FOUR

DWC3 DRIVER

4.1 TODO

Please pick something while reading :)

- Convert interrupt handler to per-ep-thread-irq

  As it turns out some DWC3-commands ~1ms to complete. Currently we spin until the command completes which is bad.

  Implementation idea:

  - dwc core implements a demultiplexing irq chip for interrupts per endpoint. The interrupt numbers are allocated during probe and belong to the device. If MSI provides per-endpoint interrupt this dummy interrupt chip can be replaced with "real" interrupts.
  - interrupts are requested / allocated on usb_ep_enable() and removed on usb_ep_disable(). Worst case are 32 interrupts, the lower limit is two for ep0/1.
  - dwc3_send_gadget_ep_cmd() will sleep in wait_for_completion_timeout() until the command completes.
  - the interrupt handler is split into the following pieces:
    * primary handler of the device goes through every event and calls generic_handle_irq() for event it. On return from generic_handle_irq() in acknowledges the event counter so interrupt goes away (eventually).
    * threaded handler of the device none
    * primary handler of the EP-interrupt reads the event and tries to process it. Everything that requires sleeping is handed over to the Thread. The event is saved in an per-endpoint data-structure. We probably have to pay attention not to process events once we handed something to thread so we don't process event X prio Y where X > Y.
    * threaded handler of the EP-interrupt handles the remaining EP work which might sleep such as waiting for command completion.

  Latency:

  There should be no increase in latency since the interrupt-thread has a high priority and will be run before an average task in user land (except the user changed priorities).

CHAPTER

FIVE

EHCI DRIVER

27-Dec-2002

The EHCI driver is used to talk to high speed USB 2.0 devices using USB 2.0-capable host controller hardware. The USB 2.0 standard is compatible with the USB 1.1 standard. It defines three transfer speeds:

- “High Speed” 480 Mbit/sec (60 MByte/sec)
- “Full Speed” 12 Mbit/sec (1.5 MByte/sec)
- “Low Speed” 1.5 Mbit/sec

USB 1.1 only addressed full speed and low speed. High speed devices can be used on USB 1.1 systems, but they slow down to USB 1.1 speeds.

USB 1.1 devices may also be used on USB 2.0 systems. When plugged into an EHCI controller, they are given to a USB 1.1 “companion” controller, which is a OHCI or UHCI controller as normally used with such devices. When USB 1.1 devices plug into USB 2.0 hubs, they interact with the EHCI controller through a “Transaction Translator” (TT) in the hub, which turns low or full speed transactions into high speed “split transactions” that don’t waste transfer bandwidth.

At this writing, this driver has been seen to work with implementations of EHCI from (in alphabetical order): Intel, NEC, Philips, and VIA. Other EHCI implementations are becoming available from other vendors; you should expect this driver to work with them too.

While usb-storage devices have been available since mid-2001 (working quite speedily on the 2.4 version of this driver), hubs have only been available since late 2001, and other kinds of high speed devices appear to be on hold until more systems come with USB 2.0 built-in. Such new systems have been available since early 2002, and became much more typical in the second half of 2002.

Note that USB 2.0 support involves more than just EHCI. It requires other changes to the Linux-USB core APIs, including the hub driver, but those changes haven’t needed to really change the basic “usbcore” APIs exposed to USB device drivers.

- David Brownell <dbrownell@users.sourceforge.net>

5.1 Functionality

This driver is regularly tested on x86 hardware, and has also been used on PPC hardware so big/little endianness issues should be gone. It's believed to do all the right PCI magic so that I/O works even on systems with interesting DMA mapping issues.

5.1.1 Transfer Types

At this writing the driver should comfortably handle all control, bulk, and interrupt transfers, including requests to USB 1.1 devices through transaction translators (TTs) in USB 2.0 hubs. But you may find bugs.

High Speed Isochronous (ISO) transfer support is also functional, but at this writing no Linux drivers have been using that support.

Full Speed Isochronous transfer support, through transaction translators, is not yet available. Note that split transaction support for ISO transfers can't share much code with the code for high speed ISO transfers, since EHCI represents these with a different data structure. So for now, most USB audio and video devices can't be connected to high speed buses.

5.1.2 Driver Behavior

Transfers of all types can be queued. This means that control transfers from a driver on one interface (or through usbfs) won't interfere with ones from another driver, and that interrupt transfers can use periods of one frame without risking data loss due to interrupt processing costs.

The EHCI root hub code hands off USB 1.1 devices to its companion controller. This driver doesn't need to know anything about those drivers; a OHCI or UHCI driver that works already doesn't need to change just because the EHCI driver is also present.

There are some issues with power management; suspend/resume doesn't behave quite right at the moment.

Also, some shortcuts have been taken with the scheduling periodic transactions (interrupt and isochronous transfers). These place some limits on the number of periodic transactions that can be scheduled, and prevent use of polling intervals of less than one frame.

5.2 Use by

Assuming you have an EHCI controller (on a PCI card or motherboard) and have compiled this driver as a module, load this like:

```
# modprobe ehci-hcd
```

and remove it by:

```
# rmmod ehci-hcd
```

You should also have a driver for a "companion controller", such as "ohci-hcd" or "uhci-hcd". In case of any trouble with the EHCI driver, remove its module and then the driver for that

companion controller will take over (at lower speed) all the devices that were previously handled by the EHCI driver.

Module parameters (pass to “modprobe”) include:

log2_irq_thresh (default 0):
Log2 of default interrupt delay, in microframes. The default value is 0, indicating 1 microframe (125 usec). Maximum value is 6, indicating 2^6 = 64 microframes. This controls how often the EHCI controller can issue interrupts.

If you’re using this driver on a 2.5 kernel, and you’ve enabled USB debugging support, you’ll see three files in the “sysfs” directory for any EHCI controller:

“async”
dumps the asynchronous schedule, used for control and bulk transfers. Shows each active qh and the qtds pending, usually one qtd per urb. (Look at it with usb-storage doing disk I/O; watch the request queues!)

“periodic”
dumps the periodic schedule, used for interrupt and isochronous transfers. Doesn’t show qtds.

“registers”
show controller register state, and

The contents of those files can help identify driver problems.

Device drivers shouldn’t care whether they’re running over EHCI or not, but they may want to check for “usb_device->speed == USB_SPEED_HIGH”. High speed devices can do things that full speed (or low speed) ones can’t, such as “high bandwidth” periodic (interrupt or ISO) transfers. Also, some values in device descriptors (such as polling intervals for periodic transfers) use different encodings when operating at high speed.

However, do make a point of testing device drivers through USB 2.0 hubs. Those hubs report some failures, such as disconnections, differently when transaction translators are in use; some drivers have been seen to behave badly when they see different faults than OHCI or UHCI report.

5.3 Performance

USB 2.0 throughput is gated by two main factors: how fast the host controller can process requests, and how fast devices can respond to them. The 480 Mbit/sec “raw transfer rate” is obeyed by all devices, but aggregate throughput is also affected by issues like delays between individual high speed packets, driver intelligence, and of course the overall system load. Latency is also a performance concern.

Bulk transfers are most often used where throughput is an issue. It’s good to keep in mind that bulk transfers are always in 512 byte packets, and at most 13 of those fit into one USB 2.0 microframe. Eight USB 2.0 microframes fit in a USB 1.1 frame; a microframe is 1 msec/8 = 125 usec.

So more than 50 MByte/sec is available for bulk transfers, when both hardware and device driver software allow it. Periodic transfer modes (isochronous and interrupt) allow the larger packet sizes which let you approach the quoted 480 MBit/sec transfer rate.

5.3.1 Hardware Performance

At this writing, individual USB 2.0 devices tend to max out at around 20 MByte/sec transfer rates. This is of course subject to change; and some devices now go faster, while others go slower.

The first NEC implementation of EHCI seems to have a hardware bottleneck at around 28 MByte/sec aggregate transfer rate. While this is clearly enough for a single device at 20 MByte/sec, putting three such devices onto one bus does not get you 60 MByte/sec. The issue appears to be that the controller hardware won't do concurrent USB and PCI access, so that it's only trying six (or maybe seven) USB transactions each microframe rather than thirteen. (Seems like a reasonable trade off for a product that beat all the others to market by over a year!)

It's expected that newer implementations will better this, throwing more silicon real estate at the problem so that new motherboard chip sets will get closer to that 60 MByte/sec target. That includes an updated implementation from NEC, as well as other vendors' silicon.

There's a minimum latency of one microframe (125 usec) for the host to receive interrupts from the EHCI controller indicating completion of requests. That latency is tunable; there's a module option. By default ehci-hcd driver uses the minimum latency, which means that if you issue a control or bulk request you can often expect to learn that it completed in less than 250 usec (depending on transfer size).

5.3.2 Software Performance

To get even 20 MByte/sec transfer rates, Linux-USB device drivers will need to keep the EHCI queue full. That means issuing large requests, or using bulk queuing if a series of small requests needs to be issued. When drivers don't do that, their performance results will show it.

In typical situations, a usb_bulk_msg() loop writing out 4 KB chunks is going to waste more than half the USB 2.0 bandwidth. Delays between the I/O completion and the driver issuing the next request will take longer than the I/O. If that same loop used 16 KB chunks, it'd be better; a sequence of 128 KB chunks would waste a lot less.

But rather than depending on such large I/O buffers to make synchronous I/O be efficient, it's better to just queue up several (bulk) requests to the HC, and wait for them all to complete (or be canceled on error). Such URB queuing should work with all the USB 1.1 HC drivers too.

In the Linux 2.5 kernels, new usb_sg_*() api calls have been defined; they queue all the buffers from a scatterlist. They also use scatterlist DMA mapping (which might apply an IOMMU) and IRQ reduction, all of which will help make high speed transfers run as fast as they can.

TBD:

Interrupt and ISO transfer performance issues. Those periodic transfers are fully scheduled, so the main issue is likely to be how to trigger "high bandwidth" modes.

TBD:

More than standard 80% periodic bandwidth allocation is possible through sysfs uframe_periodic_max parameter. Describe that.

CHAPTER

SIX

HOW FUNCTIONFS WORKS

From kernel point of view it is just a composite function with some unique behaviour. It may be added to an USB configuration only after the user space driver has registered by writing descriptors and strings (the user space program has to provide the same information that kernel level composite functions provide when they are added to the configuration).

This in particular means that the composite initialisation functions may not be in init section (ie. may not use the __init tag).

From user space point of view it is a file system which when mounted provides an “ep0” file. User space driver need to write descriptors and strings to that file. It does not need to worry about endpoints, interfaces or strings numbers but simply provide descriptors such as if the function was the only one (endpoints and strings numbers starting from one and interface numbers starting from zero). The FunctionFS changes them as needed also handling situation when numbers differ in different configurations.

When descriptors and strings are written “ep#” files appear (one for each declared endpoint) which handle communication on a single endpoint. Again, FunctionFS takes care of the real numbers and changing of the configuration (which means that “ep1” file may be really mapped to (say) endpoint 3 (and when configuration changes to (say) endpoint 2)). “ep0” is used for receiving events and handling setup requests.

When all files are closed the function disables itself.

What I also want to mention is that the FunctionFS is designed in such a way that it is possible to mount it several times so in the end a gadget could use several FunctionFS functions. The idea is that each FunctionFS instance is identified by the device name used when mounting.

One can imagine a gadget that has an Ethernet, MTP and HID interfaces where the last two are implemented via FunctionFS. On user space level it would look like this:

```
$ insmod g_ffs.ko idVendor=<ID> iSerialNumber=<string> functions=mtp,hid
$ mkdir /dev/ffs-mtp && mount -t functionfs mtp /dev/ffs-mtp
$ ( cd /dev/ffs-mtp && mtp-daemon ) &
$ mkdir /dev/ffs-hid && mount -t functionfs hid /dev/ffs-hid
$ ( cd /dev/ffs-hid && hid-daemon ) &
```

On kernel level the gadget checks ffs_data->dev_name to identify whether its FunctionFS is designed for MTP (“mtp”) or HID (“hid”).

If no “functions” module parameters is supplied, the driver accepts just one function with any name.

When “functions” module parameter is supplied, only functions with listed names are accepted. In particular, if the “functions” parameter’s value is just a one-element list, then the behaviour

is similar to when there is no “functions” at all; however, only a function with the specified name is accepted.

The gadget is registered only after all the declared function filesystems have been mounted and USB descriptors of all functions have been written to their ep0’s.

Conversely, the gadget is unregistered after the first USB function closes its endpoints.

CHAPTER

SEVEN

LINUX USB GADGET CONFIGURED THROUGH CONFIGFS

25th April 2013

7.1 Overview

A USB Linux Gadget is a device which has a UDC (USB Device Controller) and can be connected to a USB Host to extend it with additional functions like a serial port or a mass storage capability.

A gadget is seen by its host as a set of configurations, each of which contains a number of interfaces which, from the gadget's perspective, are known as functions, each function representing e.g. a serial connection or a SCSI disk.

Linux provides a number of functions for gadgets to use.

Creating a gadget means deciding what configurations there will be and which functions each configuration will provide.

Configfs (please see *Documentation/filesystems/configfs.rst*) lends itself nicely for the purpose of telling the kernel about the above mentioned decision. This document is about how to do it.

It also describes how configfs integration into gadget is designed.

7.2 Requirements

In order for this to work configfs must be available, so CONFIGFS_FS must be 'y' or 'm' in .config. As of this writing USB_LIBCOMPOSITE selects CONFIGFS_FS.

7.3 Usage

(The original post describing the first function made available through configfs can be seen here: http://www.spinics.net/lists/linux-usb/msg76388.html)

```
$ modprobe libcomposite
$ mount none $CONFIGFS_HOME -t configfs
```

where CONFIGFS_HOME is the mount point for configfs

7.3.1 1. Creating the gadgets

For each gadget to be created its corresponding directory must be created:

```
$ mkdir $CONFIGFS_HOME/usb_gadget/<gadget name>
```

e.g.:

```
$ mkdir $CONFIGFS_HOME/usb_gadget/g1

...
...
...

$ cd $CONFIGFS_HOME/usb_gadget/g1
```

Each gadget needs to have its vendor id <VID> and product id <PID> specified:

```
$ echo <VID> > idVendor
$ echo <PID> > idProduct
```

A gadget also needs its serial number, manufacturer and product strings. In order to have a place to store them, a strings subdirectory must be created for each language, e.g.:

```
$ mkdir strings/0x409
```

Then the strings can be specified:

```
$ echo <serial number> > strings/0x409/serialnumber
$ echo <manufacturer> > strings/0x409/manufacturer
$ echo <product> > strings/0x409/product
```

Further custom string descriptors can be created as directories within the language's directory, with the string text being written to the "s" attribute within the string's directory:

$ mkdir strings/0x409/xu.0 $ echo <string text> > strings/0x409/xu.0/s

Where function drivers support it, functions may allow symlinks to these custom string descriptors to associate those strings with class descriptors.

7.3.2 2. Creating the configurations

Each gadget will consist of a number of configurations, their corresponding directories must be created:

$ mkdir configs/<name>.<number>

where <name> can be any string which is legal in a filesystem and the <number> is the configuration's number, e.g.:

```
$ mkdir configs/c.1

...
```

```
...
...
```

Each configuration also needs its strings, so a subdirectory must be created for each language, e.g.:

```
$ mkdir configs/c.1/strings/0x409
```

Then the configuration string can be specified:

```
$ echo <configuration> > configs/c.1/strings/0x409/configuration
```

Some attributes can also be set for a configuration, e.g.:

```
$ echo 120 > configs/c.1/MaxPower
```

7.3.3 3. Creating the functions

The gadget will provide some functions, for each function its corresponding directory must be created:

```
$ mkdir functions/<name>.<instance name>
```

where <name> corresponds to one of allowed function names and instance name is an arbitrary string allowed in a filesystem, e.g.:

```
$ mkdir functions/ncm.usb0 # usb_f_ncm.ko gets loaded with request_module()

...
...
...
```

Each function provides its specific set of attributes, with either read-only or read-write access. Where applicable they need to be written to as appropriate. Please refer to Documentation/ABI/testing/configfs-usb-gadget for more information.

7.3.4 4. Associating the functions with their configurations

At this moment a number of gadgets is created, each of which has a number of configurations specified and a number of functions available. What remains is specifying which function is available in which configuration (the same function can be used in multiple configurations). This is achieved with creating symbolic links:

```
$ ln -s functions/<name>.<instance name> configs/<name>.<number>
```

e.g.:

```
$ ln -s functions/ncm.usb0 configs/c.1

...
```

```
...
...
```

7.3.5 5. Enabling the gadget

All the above steps serve the purpose of composing the gadget of configurations and functions.

An example directory structure might look like this:

```
.
./strings
./strings/0x409
./strings/0x409/serialnumber
./strings/0x409/product
./strings/0x409/manufacturer
./configs
./configs/c.1
./configs/c.1/ncm.usb0 -> ../../../../usb_gadget/g1/functions/ncm.usb0
./configs/c.1/strings
./configs/c.1/strings/0x409
./configs/c.1/strings/0x409/configuration
./configs/c.1/bmAttributes
./configs/c.1/MaxPower
./functions
./functions/ncm.usb0
./functions/ncm.usb0/ifname
./functions/ncm.usb0/qmult
./functions/ncm.usb0/host_addr
./functions/ncm.usb0/dev_addr
./UDC
./bcdUSB
./bcdDevice
./idProduct
./idVendor
./bMaxPacketSize0
./bDeviceProtocol
./bDeviceSubClass
./bDeviceClass
```

Such a gadget must be finally enabled so that the USB host can enumerate it.

In order to enable the gadget it must be bound to a UDC (USB Device Controller):

```
$ echo <udc name> > UDC
```

where <udc name> is one of those found in /sys/class/udc/* e.g.:

```
$ echo s3c-hsotg > UDC
```

7.3.6 6. Disabling the gadget

```
$ echo "" > UDC
```

7.3.7 7. Cleaning up

Remove functions from configurations:

```
$ rm configs/<config name>.<number>/<function>
```

where <config name>.<number> specify the configuration and <function> is a symlink to a function being removed from the configuration, e.g.:

```
$ rm configs/c.1/ncm.usb0

...
...
...
```

Remove strings directories in configurations:

    $ rmdir configs/<config name>.<number>/strings/<lang>

e.g.:

```
$ rmdir configs/c.1/strings/0x409

...
...
...
```

and remove the configurations:

```
$ rmdir configs/<config name>.<number>
```

e.g.:

```
rmdir configs/c.1

...
...
...
```

Remove functions (function modules are not unloaded, though):

    $ rmdir functions/<name>.<instance name>

e.g.:

```
$ rmdir functions/ncm.usb0

...
```

```
...
...
```

Remove strings directories in the gadget:

```
$ rmdir strings/<lang>
```

e.g.:

```
$ rmdir strings/0x409
```

and finally remove the gadget:

```
$ cd ..
$ rmdir <gadget name>
```

e.g.:

```
$ rmdir g1
```

7.4 Implementation design

Below the idea of how configfs works is presented. In configfs there are items and groups, both represented as directories. The difference between an item and a group is that a group can contain other groups. In the picture below only an item is shown. Both items and groups can have attributes, which are represented as files. The user can create and remove directories, but cannot remove files, which can be read-only or read-write, depending on what they represent.

The filesystem part of configfs operates on config_items/groups and configfs_attributes which are generic and of the same type for all configured elements. However, they are embedded in usage-specific larger structures. In the picture below there is a “cs” which contains a config_item and an “sa” which contains a configfs_attribute.

The filesystem view would be like this:

```
./
./cs        (directory)
   |
   +--sa    (file)
   |
   .
   .
   .
```

Whenever a user reads/writes the “sa” file, a function is called which accepts a struct config_item and a struct configfs_attribute. In the said function the “cs” and “sa” are retrieved using the well known container_of technique and an appropriate sa’s function (show or store) is called and passed the “cs” and a character buffer. The “show” is for displaying the file’s contents (copy data from the cs to the buffer), while the “store” is for modifying the file’s contents (copy data from the buffer to the cs), but it is up to the implementer of the two functions to decide what they actually do.

```
typedef struct configured_structure cs;
typedef struct specific_attribute sa;

                                    sa
                        +-----------------------------------+
        cs              |  (*show)(cs *, buffer);           |
+-----------------+     |  (*store)(cs *, buffer, length);  |
|                 |     |                                   |
| +-------------+ |     |       +------------------+        |
| | struct      |-|-----|------>|struct            |        |
| | config_item | |     |       |configfs_attribute|        |
| +-------------+ |     |       +------------------+        |
|                 |     +-----------------------------------+
| data to be set  |                 .
|                 |                 .
+-----------------+                 .
```

The file names are decided by the config item/group designer, while the directories in general can be named at will. A group can have a number of its default sub-groups created automatically.

For more information on configfs please see *Documentation/filesystems/configfs.rst*.

The concepts described above translate to USB gadgets like this:

1. A gadget has its config group, which has some attributes (idVendor, idProduct etc) and default sub-groups (configs, functions, strings). Writing to the attributes causes the information to be stored in appropriate locations. In the configs, functions and strings sub-groups a user can create their sub-groups to represent configurations, functions, and groups of strings in a given language.

2. The user creates configurations and functions, in the configurations creates symbolic links to functions. This information is used when the gadget's UDC attribute is written to, which means binding the gadget to the UDC. The code in drivers/usb/gadget/configfs.c iterates over all configurations, and in each configuration it iterates over all functions and binds them. This way the whole gadget is bound.

3. The file drivers/usb/gadget/configfs.c contains code for

   - gadget's config_group
   - gadget's default groups (configs, functions, strings)
   - associating functions with configurations (symlinks)

4. Each USB function naturally has its own view of what it wants configured, so config_groups for particular functions are defined in the functions implementation files drivers/usb/gadget/f_*.c.

5. Function's code is written in such a way that it uses

usb_get_function_instance(), which, in turn, calls request_module. So, provided that modprobe works, modules for particular functions are loaded automatically. Please note that the converse is not true: after a gadget is disabled and torn down, the modules remain loaded.

CHAPTER

EIGHT

LINUX USB HID GADGET DRIVER

8.1 Introduction

The HID Gadget driver provides emulation of USB Human Interface Devices (HID). The basic HID handling is done in the kernel, and HID reports can be sent/received through I/O on the /dev/hidgX character devices.

For more details about HID, see the developer page on https://www.usb.org/developers/hidpage/

8.2 Configuration

g_hid is a platform driver, so to use it you need to add struct platform_device(s) to your platform code defining the HID function descriptors you want to use - E.G. something like:

```
#include <linux/platform_device.h>
#include <linux/usb/g_hid.h>

/* hid descriptor for a keyboard */
static struct hidg_func_descriptor my_hid_data = {
      .subclass               = 0, /* No subclass */
      .protocol               = 1, /* Keyboard */
      .report_length          = 8,
      .report_desc_length     = 63,
      .report_desc            = {
              0x05, 0x01,     /* USAGE_PAGE (Generic Desktop)          */
              0x09, 0x06,     /* USAGE (Keyboard)                      */
              0xa1, 0x01,     /* COLLECTION (Application)              */
              0x05, 0x07,     /*   USAGE_PAGE (Keyboard)               */
              0x19, 0xe0,     /*   USAGE_MINIMUM (Keyboard LeftControl) */
              0x29, 0xe7,     /*   USAGE_MAXIMUM (Keyboard Right GUI)   */
              0x15, 0x00,     /*   LOGICAL_MINIMUM (0)                 */
              0x25, 0x01,     /*   LOGICAL_MAXIMUM (1)                 */
              0x75, 0x01,     /*   REPORT_SIZE (1)                     */
              0x95, 0x08,     /*   REPORT_COUNT (8)                    */
              0x81, 0x02,     /*   INPUT (Data,Var,Abs)                */
              0x95, 0x01,     /*   REPORT_COUNT (1)                    */
              0x75, 0x08,     /*   REPORT_SIZE (8)                     */
              0x81, 0x03,     /*   INPUT (Cnst,Var,Abs)                */
```

```
                0x95, 0x05,     /*   REPORT_COUNT (5)                     */
                0x75, 0x01,     /*   REPORT_SIZE (1)                      */
                0x05, 0x08,     /*   USAGE_PAGE (LEDs)                    */
                0x19, 0x01,     /*   USAGE_MINIMUM (Num Lock)             */
                0x29, 0x05,     /*   USAGE_MAXIMUM (Kana)                 */
                0x91, 0x02,     /*   OUTPUT (Data,Var,Abs)                */
                0x95, 0x01,     /*   REPORT_COUNT (1)                     */
                0x75, 0x03,     /*   REPORT_SIZE (3)                      */
                0x91, 0x03,     /*   OUTPUT (Cnst,Var,Abs)                */
                0x95, 0x06,     /*   REPORT_COUNT (6)                     */
                0x75, 0x08,     /*   REPORT_SIZE (8)                      */
                0x15, 0x00,     /*   LOGICAL_MINIMUM (0)                  */
                0x25, 0x65,     /*   LOGICAL_MAXIMUM (101)                */
                0x05, 0x07,     /*   USAGE_PAGE (Keyboard)                */
                0x19, 0x00,     /*   USAGE_MINIMUM (Reserved)             */
                0x29, 0x65,     /*   USAGE_MAXIMUM (Keyboard Application) */
                0x81, 0x00,     /*   INPUT (Data,Ary,Abs)                 */
                0xc0            /* END_COLLECTION                         */
        }
};

static struct platform_device my_hid = {
        .name                   = "hidg",
        .id                     = 0,
        .num_resources          = 0,
        .resource               = 0,
        .dev.platform_data      = &my_hid_data,
};
```

You can add as many HID functions as you want, only limited by the amount of interrupt endpoints your gadget driver supports.

8.3 Configuration with configfs

Instead of adding fake platform devices and drivers in order to pass some data to the kernel, if HID is a part of a gadget composed with configfs the hidg_func_descriptor.report_desc is passed to the kernel by writing the appropriate stream of bytes to a configfs attribute.

8.4 Send and receive HID reports

HID reports can be sent/received using read/write on the /dev/hidgX character devices. See below for an example program to do this.

hid_gadget_test is a small interactive program to test the HID gadget driver. To use, point it at a hidg device and set the device type (keyboard / mouse / joystick) - E.G.:

```
# hid_gadget_test /dev/hidg0 keyboard
```

You are now in the prompt of hid_gadget_test. You can type any combination of options and values. Available options and values are listed at program start. In keyboard mode you can send up to six values.

For example type: g i s t r --left-shift

Hit return and the corresponding report will be sent by the HID gadget.

Another interesting example is the caps lock test. Type --caps-lock and hit return. A report is then sent by the gadget and you should receive the host answer, corresponding to the caps lock LED status:

```
--caps-lock
recv report:2
```

With this command:

```
# hid_gadget_test /dev/hidg1 mouse
```

You can test the mouse emulation. Values are two signed numbers.

Sample code:

```
  /* hid_gadget_test */

  #include <pthread.h>
  #include <string.h>
  #include <stdio.h>
  #include <ctype.h>
  #include <fcntl.h>
  #include <errno.h>
  #include <stdio.h>
  #include <stdlib.h>
  #include <unistd.h>

  #define BUF_LEN 512

  struct options {
      const char    *opt;
      unsigned char val;
};

static struct options kmod[] = {
      {.opt = "--left-ctrl",          .val = 0x01},
      {.opt = "--right-ctrl",         .val = 0x10},
      {.opt = "--left-shift",         .val = 0x02},
      {.opt = "--right-shift",        .val = 0x20},
      {.opt = "--left-alt",           .val = 0x04},
      {.opt = "--right-alt",          .val = 0x40},
      {.opt = "--left-meta",          .val = 0x08},
      {.opt = "--right-meta",         .val = 0x80},
      {.opt = NULL}
};
```

```
static struct options kval[] = {
      {.opt = "--return",     .val = 0x28},
      {.opt = "--esc",        .val = 0x29},
      {.opt = "--bckspc",     .val = 0x2a},
      {.opt = "--tab",        .val = 0x2b},
      {.opt = "--spacebar",   .val = 0x2c},
      {.opt = "--caps-lock",  .val = 0x39},
      {.opt = "--f1",         .val = 0x3a},
      {.opt = "--f2",         .val = 0x3b},
      {.opt = "--f3",         .val = 0x3c},
      {.opt = "--f4",         .val = 0x3d},
      {.opt = "--f5",         .val = 0x3e},
      {.opt = "--f6",         .val = 0x3f},
      {.opt = "--f7",         .val = 0x40},
      {.opt = "--f8",         .val = 0x41},
      {.opt = "--f9",         .val = 0x42},
      {.opt = "--f10",        .val = 0x43},
      {.opt = "--f11",        .val = 0x44},
      {.opt = "--f12",        .val = 0x45},
      {.opt = "--insert",     .val = 0x49},
      {.opt = "--home",       .val = 0x4a},
      {.opt = "--pageup",     .val = 0x4b},
      {.opt = "--del",        .val = 0x4c},
      {.opt = "--end",        .val = 0x4d},
      {.opt = "--pagedown",   .val = 0x4e},
      {.opt = "--right",      .val = 0x4f},
      {.opt = "--left",       .val = 0x50},
      {.opt = "--down",       .val = 0x51},
      {.opt = "--kp-enter",   .val = 0x58},
      {.opt = "--up",         .val = 0x52},
      {.opt = "--num-lock",   .val = 0x53},
      {.opt = NULL}
};

int keyboard_fill_report(char report[8], char buf[BUF_LEN], int *hold)
{
      char *tok = strtok(buf, " ");
      int key = 0;
      int i = 0;

      for (; tok != NULL; tok = strtok(NULL, " ")) {

              if (strcmp(tok, "--quit") == 0)
                      return -1;

              if (strcmp(tok, "--hold") == 0) {
                      *hold = 1;
                      continue;
              }
```

```
		if (key < 6) {
			for (i = 0; kval[i].opt != NULL; i++)
				if (strcmp(tok, kval[i].opt) == 0) {
					report[2 + key++] = kval[i].val;
					break;
				}
			if (kval[i].opt != NULL)
				continue;
		}

		if (key < 6)
			if (islower(tok[0])) {
				report[2 + key++] = (tok[0] - ('a' - 0x04));
				continue;
			}

		for (i = 0; kmod[i].opt != NULL; i++)
			if (strcmp(tok, kmod[i].opt) == 0) {
				report[0] = report[0] | kmod[i].val;
				break;
			}
		if (kmod[i].opt != NULL)
			continue;

		if (key < 6)
			fprintf(stderr, "unknown option: %s\n", tok);
	}
	return 8;
}

static struct options mmod[] = {
	{.opt = "--b1", .val = 0x01},
	{.opt = "--b2", .val = 0x02},
	{.opt = "--b3", .val = 0x04},
	{.opt = NULL}
};

int mouse_fill_report(char report[8], char buf[BUF_LEN], int *hold)
{
	char *tok = strtok(buf, " ");
	int mvt = 0;
	int i = 0;
	for (; tok != NULL; tok = strtok(NULL, " ")) {

		if (strcmp(tok, "--quit") == 0)
			return -1;

		if (strcmp(tok, "--hold") == 0) {
			*hold = 1;
			continue;
```

```
		}

		for (i = 0; mmod[i].opt != NULL; i++)
			if (strcmp(tok, mmod[i].opt) == 0) {
				report[0] = report[0] | mmod[i].val;
				break;
			}
		if (mmod[i].opt != NULL)
			continue;

		if (!(tok[0] == '-' && tok[1] == '-') && mvt < 2) {
			errno = 0;
			report[1 + mvt++] = (char)strtol(tok, NULL, 0);
			if (errno != 0) {
				fprintf(stderr, "Bad value:'%s'\n", tok);
				report[1 + mvt--] = 0;
			}
			continue;
		}

		fprintf(stderr, "unknown option: %s\n", tok);
	}
	return 3;
}

static struct options jmod[] = {
	{.opt = "--b1",         .val = 0x10},
	{.opt = "--b2",         .val = 0x20},
	{.opt = "--b3",         .val = 0x40},
	{.opt = "--b4",         .val = 0x80},
	{.opt = "--hat1",       .val = 0x00},
	{.opt = "--hat2",       .val = 0x01},
	{.opt = "--hat3",       .val = 0x02},
	{.opt = "--hat4",       .val = 0x03},
	{.opt = "--hatneutral", .val = 0x04},
	{.opt = NULL}
};

int joystick_fill_report(char report[8], char buf[BUF_LEN], int *hold)
{
	char *tok = strtok(buf, " ");
	int mvt = 0;
	int i = 0;

	*hold = 1;

	/* set default hat position: neutral */
	report[3] = 0x04;

	for (; tok != NULL; tok = strtok(NULL, " ")) {
```

```
                if (strcmp(tok, "--quit") == 0)
                        return -1;

                for (i = 0; jmod[i].opt != NULL; i++)
                        if (strcmp(tok, jmod[i].opt) == 0) {
                                report[3] = (report[3] & 0xF0) | jmod[i].val;
                                break;
                        }
                if (jmod[i].opt != NULL)
                        continue;

                if (!(tok[0] == '-' && tok[1] == '-') && mvt < 3) {
                        errno = 0;
                        report[mvt++] = (char)strtol(tok, NULL, 0);
                        if (errno != 0) {
                                fprintf(stderr, "Bad value:'%s'\n", tok);
                                report[mvt--] = 0;
                        }
                        continue;
                }

                fprintf(stderr, "unknown option: %s\n", tok);
        }
        return 4;
}

void print_options(char c)
{
        int i = 0;

        if (c == 'k') {
                printf("        keyboard options:\n"
                       "                --hold\n");
                for (i = 0; kmod[i].opt != NULL; i++)
                        printf("\t\t%s\n", kmod[i].opt);
                printf("\n      keyboard values:\n"
                       "                [a-z] or\n");
                for (i = 0; kval[i].opt != NULL; i++)
                        printf("\t\t%-8s%s", kval[i].opt, i % 2 ? "\n" : "");
                printf("\n");
        } else if (c == 'm') {
                printf("        mouse options:\n"
                       "                --hold\n");
                for (i = 0; mmod[i].opt != NULL; i++)
                        printf("\t\t%s\n", mmod[i].opt);
                printf("\n      mouse values:\n"
                       "                Two signed numbers\n"
                       "--quit to close\n");
        } else {
```

```
                printf("        joystick options:\n");
                for (i = 0; jmod[i].opt != NULL; i++)
                        printf("\t\t%s\n", jmod[i].opt);
                printf("\n      joystick values:\n"
                       "                three signed numbers\n"
                       "--quit to close\n");
        }
}

int main(int argc, const char *argv[])
{
        const char *filename = NULL;
        int fd = 0;
        char buf[BUF_LEN];
        int cmd_len;
        char report[8];
        int to_send = 8;
        int hold = 0;
        fd_set rfds;
        int retval, i;

        if (argc < 3) {
                fprintf(stderr, "Usage: %s devname mouse|keyboard|joystick\n",
                        argv[0]);
                return 1;
        }

        if (argv[2][0] != 'k' && argv[2][0] != 'm' && argv[2][0] != 'j')
          return 2;

        filename = argv[1];

        if ((fd = open(filename, O_RDWR, 0666)) == -1) {
                perror(filename);
                return 3;
        }

        print_options(argv[2][0]);

        while (42) {

                FD_ZERO(&rfds);
                FD_SET(STDIN_FILENO, &rfds);
                FD_SET(fd, &rfds);

                retval = select(fd + 1, &rfds, NULL, NULL, NULL);
                if (retval == -1 && errno == EINTR)
                        continue;
                if (retval < 0) {
                        perror("select()");
```

```
                        return 4;
                }

                if (FD_ISSET(fd, &rfds)) {
                        cmd_len = read(fd, buf, BUF_LEN - 1);
                        printf("recv report:");
                        for (i = 0; i < cmd_len; i++)
                                printf(" %02x", buf[i]);
                        printf("\n");
                }

                if (FD_ISSET(STDIN_FILENO, &rfds)) {
                        memset(report, 0x0, sizeof(report));
                        cmd_len = read(STDIN_FILENO, buf, BUF_LEN - 1);

                        if (cmd_len == 0)
                                break;

                        buf[cmd_len - 1] = '\0';
                        hold = 0;

                        memset(report, 0x0, sizeof(report));
                        if (argv[2][0] == 'k')
                                to_send = keyboard_fill_report(report, buf, &
↪hold);
                        else if (argv[2][0] == 'm')
                                to_send = mouse_fill_report(report, buf, &hold);
                        else
                                to_send = joystick_fill_report(report, buf, &
↪hold);

                        if (to_send == -1)
                                break;

                        if (write(fd, report, to_send) != to_send) {
                                perror(filename);
                                return 5;
                        }
                        if (!hold) {
                                memset(report, 0x0, sizeof(report));
                                if (write(fd, report, to_send) != to_send) {
                                        perror(filename);
                                        return 6;
                                }
                        }
                }
        }

        close(fd);
        return 0;
```

```
}
```

CHAPTER

NINE

MULTIFUNCTION COMPOSITE GADGET

9.1 Overview

The Multifunction Composite Gadget (or g_multi) is a composite gadget that makes extensive use of the composite framework to provide a... multifunction gadget.

In its standard configuration it provides a single USB configuration with RNDIS[1] (that is Ethernet), USB CDC[2] ACM (that is serial) and USB Mass Storage functions.

A CDC ECM (Ethernet) function may be turned on via a Kconfig option and RNDIS can be turned off. If they are both enabled the gadget will have two configurations -- one with RNDIS and another with CDC ECM[3].

Please note that if you use non-standard configuration (that is enable CDC ECM) you may need to change vendor and/or product ID.

9.2 Host drivers

To make use of the gadget one needs to make it work on host side -- without that there's no hope of achieving anything with the gadget. As one might expect, things one need to do very from system to system.

9.2.1 Linux host drivers

Since the gadget uses standard composite framework and appears as such to Linux host it does not need any additional drivers on Linux host side. All the functions are handled by respective drivers developed for them.

This is also true for two configuration set-up with RNDIS configuration being the first one. Linux host will use the second configuration with CDC ECM which should work better under Linux.

9.2.2 Windows host drivers

For the gadget to work under Windows two conditions have to be met:

Detecting as composite gadget

First of all, Windows need to detect the gadget as an USB composite gadget which on its own have some conditions[4]. If they are met, Windows lets USB Generic Parent Driver[5] handle the device which then tries to match drivers for each individual interface (sort of, don't get into too many details).

The good news is: you do not have to worry about most of the conditions!

The only thing to worry is that the gadget has to have a single configuration so a dual RNDIS and CDC ECM gadget won't work unless you create a proper INF -- and of course, if you do submit it!

Installing drivers for each function

The other, trickier thing is making Windows install drivers for each individual function.

For mass storage it is trivial since Windows detect it's an interface implementing USB Mass Storage class and selects appropriate driver.

Things are harder with RDNIS and CDC ACM.

RNDIS

To make Windows select RNDIS drivers for the first function in the gadget, one needs to use the [[file:linux.inf]] file provided with this document. It "attaches" Window's RNDIS driver to the first interface of the gadget.

Please note, that while testing we encountered some issues[6] when RNDIS was not the first interface. You do not need to worry abut it unless you are trying to develop your own gadget in which case watch out for this bug.

CDC ACM

Similarly, [[file:linux-cdc-acm.inf]] is provided for CDC ACM.

Customising the gadget

If you intend to hack the g_multi gadget be advised that rearranging functions will obviously change interface numbers for each of the functionality. As an effect provided INFs won't work since they have interface numbers hard-coded in them (it's not hard to change those though[7]).

This also means, that after experimenting with g_multi and changing provided functions one should change gadget's vendor and/or product ID so there will be no collision with other customised gadgets or the original gadget.

Failing to comply may cause brain damage after wondering for hours why things don't work as intended before realising Windows have cached some drivers information (changing USB port may sometimes help plus you might try using USBDeview[8] to remove the phantom device).

INF testing

Provided INF files have been tested on Windows XP SP3, Windows Vista and Windows 7, all 32-bit versions. It should work on 64-bit versions as well. It most likely won't work on Windows prior to Windows XP SP2.

9.2.3 Other systems

At this moment, drivers for any other systems have not been tested. Knowing how MacOS is based on BSD and BSD is an Open Source it is believed that it should (read: "I have no idea whether it will") work out-of-the-box.

For more exotic systems I have even less to say...

Any testing and drivers *are welcome*!

9.3 Authors

This document has been written by Michal Nazarewicz ([[mailto:mina86@mina86.com]]). INF files have been hacked with support of Marek Szyprowski ([[mailto:m.szyprowski@samsung.com]]) and Xiaofan Chen ([[mailto:xiaofanc@gmail.com]]) basing on the MS RNDIS template[9], Microchip's CDC ACM INF file and David Brownell's ([[mailto:dbrownell@users.sourceforge.net]]) original INF files.

9.4 Footnotes

[1] Remote Network Driver Interface Specification, [[https://msdn.microsoft.com/en-us/library/ee484414.aspx]].

[2] Communications Device Class Abstract Control Model, spec for this and other USB classes can be found at [[http://www.usb.org/developers/devclass_docs/]].

[3] CDC Ethernet Control Model.

[4] [[https://msdn.microsoft.com/en-us/library/ff537109(v=VS.85).aspx]]

[5] [[https://msdn.microsoft.com/en-us/library/ff539234(v=VS.85).aspx]]

[6] To put it in some other nice words, Windows failed to respond to any user input.

[7] You may find [[http://www.cygnal.org/ubb/Forum9/HTML/001050.html]] useful.

[8] https://www.nirsoft.net/utils/usb_devices_view.html

[9] [[https://msdn.microsoft.com/en-us/library/ff570620.aspx]]

CHAPTER

TEN

LINUX USB PRINTER GADGET DRIVER

06/04/2007

10.1 General

This driver may be used if you are writing printer firmware using Linux as the embedded OS. This driver has nothing to do with using a printer with your Linux host system.

You will need a USB device controller and a Linux driver for it that accepts a gadget / “device class” driver using the Linux USB Gadget API. After the USB device controller driver is loaded then load the printer gadget driver. This will present a printer interface to the USB Host that your USB Device port is connected to.

This driver is structured for printer firmware that runs in user mode. The user mode printer firmware will read and write data from the kernel mode printer gadget driver using a device file. The printer returns a printer status byte when the USB HOST sends a device request to get the printer status. The user space firmware can read or write this status byte using a device file /dev/g_printer . Both blocking and non-blocking read/write calls are supported.

10.2 Howto Use This Driver

To load the USB device controller driver and the printer gadget driver. The following example uses the Netchip 2280 USB device controller driver:

```
modprobe net2280
modprobe g_printer
```

The follow command line parameter can be used when loading the printer gadget (ex: modprobe g_printer idVendor=0x0525 idProduct=0xa4a8):

idVendor
: This is the Vendor ID used in the device descriptor. The default is the Netchip vendor id 0x0525. YOU MUST CHANGE TO YOUR OWN VENDOR ID BEFORE RELEASING A PRODUCT. If you plan to release a product and don’t already have a Vendor ID please see www.usb.org for details on how to get one.

idProduct
: This is the Product ID used in the device descriptor. The default is 0xa4a8, you should

change this to an ID that's not used by any of your other USB products if you have any. It would be a good idea to start numbering your products starting with say 0x0001.

bcdDevice
This is the version number of your product. It would be a good idea to put your firmware version here.

iManufacturer
A string containing the name of the Vendor.

iProduct
A string containing the Product Name.

iSerialNum
A string containing the Serial Number. This should be changed for each unit of your product.

iPNPstring
The PNP ID string used for this printer. You will want to set either on the command line or hard code the PNP ID string used for your printer product.

qlen
The number of 8k buffers to use per endpoint. The default is 10, you should tune this for your product. You may also want to tune the size of each buffer for your product.

10.3 Using The Example Code

This example code talks to stdout, instead of a print engine.

To compile the test code below:

1) save it to a file called prn_example.c
2) compile the code with the follow command:

```
gcc prn_example.c -o prn_example
```

To read printer data from the host to stdout:

```
# prn_example -read_data
```

To write printer data from a file (data_file) to the host:

```
# cat data_file | prn_example -write_data
```

To get the current printer status for the gadget driver::

```
# prn_example -get_status

Printer status is:
     Printer is NOT Selected
     Paper is Out
     Printer OK
```

To set printer to Selected/On-line:

```
# prn_example -selected
```

To set printer to Not Selected/Off-line:

```
# prn_example -not_selected
```

To set paper status to paper out:

```
# prn_example -paper_out
```

To set paper status to paper loaded:

```
# prn_example -paper_loaded
```

To set error status to printer OK:

```
# prn_example -no_error
```

To set error status to ERROR:

```
# prn_example -error
```

10.4 Example Code

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <linux/poll.h>
#include <sys/ioctl.h>
#include <linux/usb/g_printer.h>

#define PRINTER_FILE                    "/dev/g_printer"
#define BUF_SIZE                        512


/*
 * 'usage()' - Show program usage.
 */

static void
usage(const char *option)               /* I - Option string or NULL */
{
      if (option) {
              fprintf(stderr,"prn_example: Unknown option \"%s\"!\n",
                              option);
      }

      fputs("\n", stderr);
      fputs("Usage: prn_example -[options]\n", stderr);
      fputs("Options:\n", stderr);
```

```
      fputs("\n", stderr);
      fputs("-get_status    Get the current printer status.\n", stderr);
      fputs("-selected      Set the selected status to selected.\n", stderr);
      fputs("-not_selected  Set the selected status to NOT selected.\n",
                      stderr);
      fputs("-error         Set the error status to error.\n", stderr);
      fputs("-no_error      Set the error status to NO error.\n", stderr);
      fputs("-paper_out     Set the paper status to paper out.\n", stderr);
      fputs("-paper_loaded  Set the paper status to paper loaded.\n",
                      stderr);
      fputs("-read_data     Read printer data from driver.\n", stderr);
      fputs("-write_data    Write printer sata to driver.\n", stderr);
      fputs("-NB_read_data  (Non-Blocking) Read printer data from driver.\n",
                      stderr);
      fputs("\n\n", stderr);

      exit(1);
}


static int
read_printer_data()
{
      struct pollfd   fd[1];

      /* Open device file for printer gadget. */
      fd[0].fd = open(PRINTER_FILE, O_RDWR);
      if (fd[0].fd < 0) {
              printf("Error %d opening %s\n", fd[0].fd, PRINTER_FILE);
              close(fd[0].fd);
              return(-1);
      }

      fd[0].events = POLLIN | POLLRDNORM;

      while (1) {
              static char buf[BUF_SIZE];
              int bytes_read;
              int retval;

              /* Wait for up to 1 second for data. */
              retval = poll(fd, 1, 1000);

              if (retval && (fd[0].revents & POLLRDNORM)) {

                      /* Read data from printer gadget driver. */
                      bytes_read = read(fd[0].fd, buf, BUF_SIZE);

                      if (bytes_read < 0) {
                              printf("Error %d reading from %s\n",
```

```
                                                fd[0].fd, PRINTER_FILE);
                                close(fd[0].fd);
                                return(-1);
                        } else if (bytes_read > 0) {
                                /* Write data to standard OUTPUT (stdout). */
                                fwrite(buf, 1, bytes_read, stdout);
                                fflush(stdout);
                        }

                }

        }

        /* Close the device file. */
        close(fd[0].fd);

        return 0;
}


static int
write_printer_data()
{
        struct pollfd   fd[1];

        /* Open device file for printer gadget. */
        fd[0].fd = open (PRINTER_FILE, O_RDWR);
        if (fd[0].fd < 0) {
                printf("Error %d opening %s\n", fd[0].fd, PRINTER_FILE);
                close(fd[0].fd);
                return(-1);
        }

        fd[0].events = POLLOUT | POLLWRNORM;

        while (1) {
                int retval;
                static char buf[BUF_SIZE];
                /* Read data from standard INPUT (stdin). */
                int bytes_read = fread(buf, 1, BUF_SIZE, stdin);

                if (!bytes_read) {
                        break;
                }

                while (bytes_read) {

                        /* Wait for up to 1 second to sent data. */
                        retval = poll(fd, 1, 1000);
```

```
                        /* Write data to printer gadget driver. */
                        if (retval && (fd[0].revents & POLLWRNORM)) {
                                retval = write(fd[0].fd, buf, bytes_read);
                                if (retval < 0) {
                                        printf("Error %d writing to %s\n",
                                                        fd[0].fd,
                                                        PRINTER_FILE);
                                        close(fd[0].fd);
                                        return(-1);
                                } else {
                                        bytes_read -= retval;
                                }

                        }

                }

        }

        /* Wait until the data has been sent. */
        fsync(fd[0].fd);

        /* Close the device file. */
        close(fd[0].fd);

        return 0;
}


static int
read_NB_printer_data()
{
        int             fd;
        static char     buf[BUF_SIZE];
        int             bytes_read;

        /* Open device file for printer gadget. */
        fd = open(PRINTER_FILE, O_RDWR|O_NONBLOCK);
        if (fd < 0) {
                printf("Error %d opening %s\n", fd, PRINTER_FILE);
                close(fd);
                return(-1);
        }

        while (1) {
                /* Read data from printer gadget driver. */
                bytes_read = read(fd, buf, BUF_SIZE);
                if (bytes_read <= 0) {
                        break;
                }
```

```
                /* Write data to standard OUTPUT (stdout). */
                fwrite(buf, 1, bytes_read, stdout);
                fflush(stdout);
        }

        /* Close the device file. */
        close(fd);

        return 0;
}


static int
get_printer_status()
{
        int     retval;
        int     fd;

        /* Open device file for printer gadget. */
        fd = open(PRINTER_FILE, O_RDWR);
        if (fd < 0) {
                printf("Error %d opening %s\n", fd, PRINTER_FILE);
                close(fd);
                return(-1);
        }

        /* Make the IOCTL call. */
        retval = ioctl(fd, GADGET_GET_PRINTER_STATUS);
        if (retval < 0) {
                fprintf(stderr, "ERROR: Failed to set printer status\n");
                return(-1);
        }

        /* Close the device file. */
        close(fd);

        return(retval);
}


static int
set_printer_status(unsigned char buf, int clear_printer_status_bit)
{
        int     retval;
        int     fd;

        retval = get_printer_status();
        if (retval < 0) {
                fprintf(stderr, "ERROR: Failed to get printer status\n");
```

```
                return(-1);
        }

        /* Open device file for printer gadget. */
        fd = open(PRINTER_FILE, O_RDWR);

        if (fd < 0) {
                printf("Error %d opening %s\n", fd, PRINTER_FILE);
                close(fd);
                return(-1);
        }

        if (clear_printer_status_bit) {
                retval &= ~buf;
        } else {
                retval |= buf;
        }

        /* Make the IOCTL call. */
        if (ioctl(fd, GADGET_SET_PRINTER_STATUS, (unsigned char)retval)) {
                fprintf(stderr, "ERROR: Failed to set printer status\n");
                return(-1);
        }

        /* Close the device file. */
        close(fd);

        return 0;
}


static int
display_printer_status()
{
        char    printer_status;

        printer_status = get_printer_status();
        if (printer_status < 0) {
                fprintf(stderr, "ERROR: Failed to get printer status\n");
                return(-1);
        }

        printf("Printer status is:\n");
        if (printer_status & PRINTER_SELECTED) {
                printf("     Printer is Selected\n");
        } else {
                printf("     Printer is NOT Selected\n");
        }
        if (printer_status & PRINTER_PAPER_EMPTY) {
                printf("     Paper is Out\n");
```

```
      } else {
              printf("     Paper is Loaded\n");
      }
      if (printer_status & PRINTER_NOT_ERROR) {
              printf("     Printer OK\n");
      } else {
              printf("     Printer ERROR\n");
      }

      return(0);
}


int
main(int  argc, char *argv[])
{
      int     i;              /* Looping var */
      int     retval = 0;

      /* No Args */
      if (argc == 1) {
              usage(0);
              exit(0);
      }

      for (i = 1; i < argc && !retval; i ++) {

              if (argv[i][0] != '-') {
                      continue;
              }

              if (!strcmp(argv[i], "-get_status")) {
                      if (display_printer_status()) {
                              retval = 1;
                      }

              } else if (!strcmp(argv[i], "-paper_loaded")) {
                      if (set_printer_status(PRINTER_PAPER_EMPTY, 1)) {
                              retval = 1;
                      }

              } else if (!strcmp(argv[i], "-paper_out")) {
                      if (set_printer_status(PRINTER_PAPER_EMPTY, 0)) {
                              retval = 1;
                      }

              } else if (!strcmp(argv[i], "-selected")) {
                      if (set_printer_status(PRINTER_SELECTED, 0)) {
                              retval = 1;
                      }
```

```
                } else if (!strcmp(argv[i], "-not_selected")) {
                        if (set_printer_status(PRINTER_SELECTED, 1)) {
                                retval = 1;
                        }

                } else if (!strcmp(argv[i], "-error")) {
                        if (set_printer_status(PRINTER_NOT_ERROR, 1)) {
                                retval = 1;
                        }

                } else if (!strcmp(argv[i], "-no_error")) {
                        if (set_printer_status(PRINTER_NOT_ERROR, 0)) {
                                retval = 1;
                        }

                } else if (!strcmp(argv[i], "-read_data")) {
                        if (read_printer_data()) {
                                retval = 1;
                        }

                } else if (!strcmp(argv[i], "-write_data")) {
                        if (write_printer_data()) {
                                retval = 1;
                        }

                } else if (!strcmp(argv[i], "-NB_read_data")) {
                        if (read_NB_printer_data()) {
                                retval = 1;
                        }

                } else {
                        usage(argv[i]);
                        retval = 1;
                }
        }

        exit(retval);
}
```

CHAPTER

ELEVEN

LINUX GADGET SERIAL DRIVER V2.0

11/20/2004

(updated 8-May-2008 for v2.3)

11.1 License and Disclaimer

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA.

This document and the gadget serial driver itself are Copyright (C) 2004 by Al Borchers (alborchers@steinerpoint.com).

If you have questions, problems, or suggestions for this driver please contact Al Borchers at alborchers@steinerpoint.com.

11.2 Prerequisites

Versions of the gadget serial driver are available for the 2.4 Linux kernels, but this document assumes you are using version 2.3 or later of the gadget serial driver in a 2.6 Linux kernel.

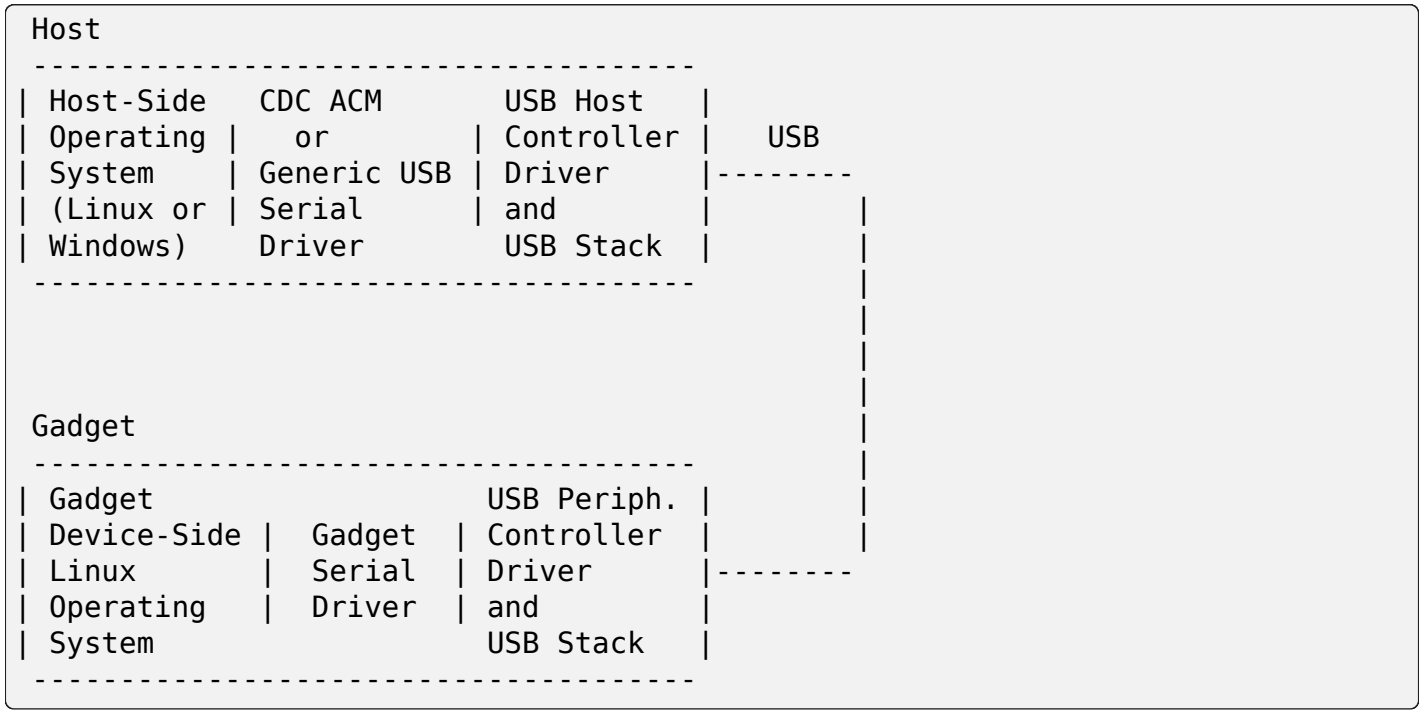
This document assumes that you are familiar with Linux and Windows and know how to configure and build Linux kernels, run standard utilities, use minicom and HyperTerminal, and work with USB and serial devices. It also assumes you configure the Linux gadget and usb drivers as modules.

With version 2.3 of the driver, major and minor device nodes are no longer statically defined. Your Linux based system should mount sysfs in /sys, and use “mdev” (in Busybox) or “udev” to make the /dev nodes matching the sysfs /sys/class/tty files.

11.3 Overview

The gadget serial driver is a Linux USB gadget driver, a USB device side driver. It runs on a Linux system that has USB device side hardware; for example, a PDA, an embedded Linux system, or a PC with a USB development card.

The gadget serial driver talks over USB to either a CDC ACM driver or a generic USB serial driver running on a host PC:



On the device-side Linux system, the gadget serial driver looks like a serial device.

On the host-side system, the gadget serial device looks like a CDC ACM compliant class device or a simple vendor specific device with bulk in and bulk out endpoints, and it is treated similarly to other serial devices.

The host side driver can potentially be any ACM compliant driver or any driver that can talk to a device with a simple bulk in/out interface. Gadget serial has been tested with the Linux ACM driver, the Windows usbser.sys ACM driver, and the Linux USB generic serial driver.

With the gadget serial driver and the host side ACM or generic serial driver running, you should be able to communicate between the host and the gadget side systems as if they were connected by a serial cable.

The gadget serial driver only provides simple unreliable data communication. It does not yet handle flow control or many other features of normal serial devices.

11.4 Installing the Gadget Serial Driver

To use the gadget serial driver you must configure the Linux gadget side kernel for “Support for USB Gadgets”, for a “USB Peripheral Controller” (for example, net2280), and for the “Serial Gadget” driver. All this are listed under “USB Gadget Support” when configuring the kernel. Then rebuild and install the kernel or modules.

Then you must load the gadget serial driver. To load it as an ACM device (recommended for interoperability), do this:

```
modprobe g_serial
```

To load it as a vendor specific bulk in/out device, do this:

```
modprobe g_serial use_acm=0
```

This will also automatically load the underlying gadget peripheral controller driver. This must be done each time you reboot the gadget side Linux system. You can add this to the start up scripts, if desired.

Your system should use mdev (from busybox) or udev to make the device nodes. After this gadget driver has been set up you should then see a /dev/ttyGS0 node:

```
# ls -l /dev/ttyGS0 | cat
crw-rw----    1 root     root     253,   0 May  8 14:10 /dev/ttyGS0
#
```

Note that the major number (253, above) is system-specific. If you need to create /dev nodes by hand, the right numbers to use will be in the /sys/class/tty/ttyGS0/dev file.

When you link this gadget driver early, perhaps even statically, you may want to set up an /etc/inittab entry to run “getty” on it. The /dev/ttyGS0 line should work like most any other serial port.

If gadget serial is loaded as an ACM device you will want to use either the Windows or Linux ACM driver on the host side. If gadget serial is loaded as a bulk in/out device, you will want to use the Linux generic serial driver on the host side. Follow the appropriate instructions below to install the host side driver.

11.5 Installing the Windows Host ACM Driver

To use the Windows ACM driver you must have the “linux-cdc-acm.inf” file (provided along this document) which supports all recent versions of Windows.

When the gadget serial driver is loaded and the USB device connected to the Windows host with a USB cable, Windows should recognize the gadget serial device and ask for a driver. Tell Windows to find the driver in the folder that contains the “linux-cdc-acm.inf” file.

For example, on Windows XP, when the gadget serial device is first plugged in, the “Found New Hardware Wizard” starts up. Select “Install from a list or specific location (Advanced)”, then on the next screen select “Include this location in the search” and enter the path or browse to the folder containing the “linux-cdc-acm.inf” file. Windows will complain that the Gadget Serial

driver has not passed Windows Logo testing, but select “Continue anyway” and finish the driver installation.

On Windows XP, in the “Device Manager” (under “Control Panel”, “System”, “Hardware”) expand the “Ports (COM & LPT)” entry and you should see “Gadget Serial” listed as the driver for one of the COM ports.

To uninstall the Windows XP driver for “Gadget Serial”, right click on the “Gadget Serial” entry in the “Device Manager” and select “Uninstall”.

11.6 Installing the Linux Host ACM Driver

To use the Linux ACM driver you must configure the Linux host side kernel for “Support for Host-side USB” and for “USB Modem (CDC ACM) support”.

Once the gadget serial driver is loaded and the USB device connected to the Linux host with a USB cable, the host system should recognize the gadget serial device. For example, the command:

```
cat /sys/kernel/debug/usb/devices
```

should show something like this::

```
T:  Bus=01 Lev=01 Prnt=01 Port=01 Cnt=02 Dev#=  5 Spd=480 MxCh= 0
D:  Ver= 2.00 Cls=02(comm.) Sub=00 Prot=00 MxPS=64 #Cfgs=  1
P:  Vendor=0525 ProdID=a4a7 Rev= 2.01
S:  Manufacturer=Linux 2.6.8.1 with net2280
S:  Product=Gadget Serial
S:  SerialNumber=0
C:* #Ifs= 2 Cfg#= 2 Atr=c0 MxPwr=  2mA
I:  If#= 0 Alt= 0 #EPs= 1 Cls=02(comm.) Sub=02 Prot=01 Driver=acm
E:  Ad=83(I) Atr=03(Int.) MxPS=   8 Ivl=32ms
I:  If#= 1 Alt= 0 #EPs= 2 Cls=0a(data ) Sub=00 Prot=00 Driver=acm
E:  Ad=81(I) Atr=02(Bulk) MxPS= 512 Ivl=0ms
E:  Ad=02(O) Atr=02(Bulk) MxPS= 512 Ivl=0ms
```

If the host side Linux system is configured properly, the ACM driver should be loaded automatically. The command “lsmod” should show the “acm” module is loaded.

11.7 Installing the Linux Host Generic USB Serial Driver

To use the Linux generic USB serial driver you must configure the Linux host side kernel for “Support for Host-side USB”, for “USB Serial Converter support”, and for the “USB Generic Serial Driver”.

Once the gadget serial driver is loaded and the USB device connected to the Linux host with a USB cable, the host system should recognize the gadget serial device. For example, the command:

```
cat /sys/kernel/debug/usb/devices
```

should show something like this::

```
T:  Bus=01 Lev=01 Prnt=01 Port=01 Cnt=02 Dev#=  6 Spd=480 MxCh= 0
D:  Ver= 2.00 Cls=ff(vend.) Sub=00 Prot=00 MxPS=64 #Cfgs=  1
P:  Vendor=0525 ProdID=a4a6 Rev= 2.01
S:  Manufacturer=Linux 2.6.8.1 with net2280
S:  Product=Gadget Serial
S:  SerialNumber=0
C:* #Ifs= 1 Cfg#= 1 Atr=c0 MxPwr=  2mA
I:  If#= 0 Alt= 0 #EPs= 2 Cls=0a(data ) Sub=00 Prot=00 Driver=serial
E:  Ad=81(I) Atr=02(Bulk) MxPS= 512 Ivl=0ms
E:  Ad=02(O) Atr=02(Bulk) MxPS= 512 Ivl=0ms
```

You must load the usbserial driver and explicitly set its parameters to configure it to recognize the gadget serial device, like this:

```
echo 0x0525 0xA4A6 >/sys/bus/usb-serial/drivers/generic/new_id
```

The legacy way is to use module parameters:

```
modprobe usbserial vendor=0x0525 product=0xA4A6
```

If everything is working, usbserial will print a message in the system log saying something like “Gadget Serial converter now attached to ttyUSB0”.

11.8 Testing with Minicom or HyperTerminal

Once the gadget serial driver and the host driver are both installed, and a USB cable connects the gadget device to the host, you should be able to communicate over USB between the gadget and host systems. You can use minicom or HyperTerminal to try this out.

On the gadget side run “minicom -s” to configure a new minicom session. Under “Serial port setup” set “/dev/ttygserial” as the “Serial Device”. Set baud rate, data bits, parity, and stop bits, to 9600, 8, none, and 1--these settings mostly do not matter. Under “Modem and dialing” erase all the modem and dialing strings.

On a Linux host running the ACM driver, configure minicom similarly but use “/dev/ttyACM0” as the “Serial Device”. (If you have other ACM devices connected, change the device name appropriately.)

On a Linux host running the USB generic serial driver, configure minicom similarly, but use “/dev/ttyUSB0” as the “Serial Device”. (If you have other USB serial devices connected, change the device name appropriately.)

On a Windows host configure a new HyperTerminal session to use the COM port assigned to Gadget Serial. The “Port Settings” will be set automatically when HyperTerminal connects to the gadget serial device, so you can leave them set to the default values--these settings mostly do not matter.

With minicom configured and running on the gadget side and with minicom or HyperTerminal configured and running on the host side, you should be able to send data back and forth between the gadget side and host side systems. Anything you type on the terminal window on the gadget side should appear in the terminal window on the host side and vice versa.

CHAPTER

TWELVE

LINUX UVC GADGET DRIVER

12.1 Overview

The UVC Gadget driver is a driver for hardware on the *device* side of a USB connection. It is intended to run on a Linux system that has USB device-side hardware such as boards with an OTG port.

On the device system, once the driver is bound it appears as a V4L2 device with the output capability.

On the host side (once connected via USB cable), a device running the UVC Gadget driver *and controlled by an appropriate userspace program* should appear as a UVC specification compliant camera, and function appropriately with any program designed to handle them. The userspace program running on the device system can queue image buffers from a variety of sources to be transmitted via the USB connection. Typically this would mean forwarding the buffers from a camera sensor peripheral, but the source of the buffer is entirely dependent on the userspace companion program.

12.2 Configuring the device kernel

The Kconfig options USB_CONFIGFS, USB_LIBCOMPOSITE, USB_CONFIGFS_F_UVC and USB_F_UVC must be selected to enable support for the UVC gadget.

12.3 Configuring the gadget through configfs

The UVC Gadget expects to be configured through configfs using the UVC function. This allows a significant degree of flexibility, as many of a UVC device's settings can be controlled this way.

Not all of the available attributes are described here. For a complete enumeration see Documentation/ABI/testing/configfs-usb-gadget-uvc

12.3.1 Assumptions

This section assumes that you have mounted configfs at */sys/kernel/config* and created a gadget as */sys/kernel/config/usb_gadget/g1*.

12.3.2 The UVC Function

The first step is to create the UVC function:

```
# These variables will be assumed throughout the rest of the document
CONFIGFS="/sys/kernel/config"
GADGET="$CONFIGFS/usb_gadget/g1"
FUNCTION="$GADGET/functions/uvc.0"

mkdir -p $FUNCTION
```

12.3.3 Formats and Frames

You must configure the gadget by telling it which formats you support, as well as the frame sizes and frame intervals that are supported for each format. In the current implementation there is no way for the gadget to refuse to set a format that the host instructs it to set, so it is important that this step is completed *accurately* to ensure that the host never asks for a format that can't be provided.

Formats are created under the streaming/uncompressed and streaming/mjpeg configfs groups, with the framesizes created under the formats in the following structure:

```
uvc.0 +
      |
      + streaming +
                  |
                  + mjpeg +
                  |       |
                  |       + mjpeg +
                  |             |
                  |             + 720p
                  |             |
                  |             + 1080p
                  |
                  + uncompressed +
                                 |
                                 + yuyv +
                                        |
                                        + 720p
                                        |
                                        + 1080p
```

Each frame can then be configured with a width and height, plus the maximum buffer size required to store a single frame, and finally with the supported frame intervals for that format and framesize. Width and height are enumerated in units of pixels, frame interval in units of

100ns. To create the structure above with 2, 15 and 100 fps frameintervals for each framesize for example you might do:

```
create_frame() {
        # Example usage:
        # create_frame <width> <height> <group> <format name>

        WIDTH=$1
        HEIGHT=$2
        FORMAT=$3
        NAME=$4

        wdir=$FUNCTION/streaming/$FORMAT/$NAME/${HEIGHT}p

        mkdir -p $wdir
        echo $WIDTH > $wdir/wWidth
        echo $HEIGHT > $wdir/wHeight
        echo $(( $WIDTH * $HEIGHT * 2 )) > $wdir/dwMaxVideoFrameBufferSize
        cat <<EOF > $wdir/dwFrameInterval
666666
100000
5000000
EOF
}

create_frame 1280 720 mjpeg mjpeg
create_frame 1920 1080 mjpeg mjpeg
create_frame 1280 720 uncompressed yuyv
create_frame 1920 1080 uncompressed yuyv
```

The only uncompressed format currently supported is YUYV, which is detailed at Documentation/userspace-api/media/v4l/pixfmt-packed.yuv.rst.

12.3.4 Color Matching Descriptors

It's possible to specify some colometry information for each format you create. This step is optional, and default information will be included if this step is skipped; those default values follow those defined in the Color Matching Descriptor section of the UVC specification.

To create a Color Matching Descriptor, create a configfs item and set its three attributes to your desired settings and then link to it from the format you wish it to be associated with:

```
# Create a new Color Matching Descriptor

mkdir $FUNCTION/streaming/color_matching/yuyv
pushd $FUNCTION/streaming/color_matching/yuyv

echo 1 > bColorPrimaries
echo 1 > bTransferCharacteristics
echo 4 > bMatrixCoefficients
```

```
popd

# Create a symlink to the Color Matching Descriptor from the format's config␣
↪item
ln -s $FUNCTION/streaming/color_matching/yuyv $FUNCTION/streaming/uncompressed/
↪yuyv
```

For details about the valid values, consult the UVC specification. Note that a default color matching descriptor exists and is used by any format which does not have a link to a different Color Matching Descriptor. It's possible to change the attribute settings for the default descriptor, so bear in mind that if you do that you are altering the defaults for any format that does not link to a different one.

12.3.5 Header linking

The UVC specification requires that Format and Frame descriptors be preceded by Headers detailing things such as the number and cumulative size of the different Format descriptors that follow. This and similar operations are achieved in configfs by linking between the configfs item representing the header and the config items representing those other descriptors, in this manner:

```
mkdir $FUNCTION/streaming/header/h

# This section links the format descriptors and their associated frames
# to the header
cd $FUNCTION/streaming/header/h
ln -s ../../uncompressed/yuyv
ln -s ../../mjpeg/mjpeg

# This section ensures that the header will be transmitted for each
# speed's set of descriptors. If support for a particular speed is not
# needed then it can be skipped here.
cd ../../class/fs
ln -s ../../header/h
cd ../../class/hs
ln -s ../../header/h
cd ../../class/ss
ln -s ../../header/h
cd ../../../control
mkdir header/h
ln -s header/h class/fs
ln -s header/h class/ss
```

12.3.6 Extension Unit Support

A UVC Extension Unit (XU) basically provides a distinct unit to which control set and get requests can be addressed. The meaning of those control requests is entirely implementation dependent, but may be used to control settings outside of the UVC specification (for example enabling or disabling video effects). An XU can be inserted into the UVC unit chain or left free-hanging.

Configuring an extension unit involves creating an entry in the appropriate directory and setting its attributes appropriately, like so:

```
mkdir $FUNCTION/control/extensions/xu.0
pushd $FUNCTION/control/extensions/xu.0

# Set the bUnitID of the Processing Unit as the source for this
# Extension Unit
echo 2 > baSourceID

# Set this XU as the source of the default output terminal. This inserts
# the XU into the UVC chain between the PU and OT such that the final
# chain is IT > PU > XU.0 > OT
cat bUnitID > ../../terminal/output/default/baSourceID

# Flag some controls as being available for use. The bmControl field is
# a bitmap with each bit denoting the availability of a particular
# control. For example to flag the 0th, 2nd and 3rd controls available:
echo 0x0d > bmControls

# Set the GUID; this is a vendor-specific code identifying the XU.
echo -e -n "\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f\x10" >
 ↪ guidExtensionCode

popd
```

The bmControls attribute and the baSourceID attribute are multi-value attributes. This means that you may write multiple newline separated values to them. For example to flag the 1st, 2nd, 9th and 10th controls as being available you would need to write two values to bmControls, like so:

```
cat << EOF > bmControls
0x03
0x03
EOF
```

The multi-value nature of the baSourceID attribute belies the fact that XUs can be multiple-input, though note that this currently has no significant effect.

The bControlSize attribute reflects the size of the bmControls attribute, and similarly bNrInPins reflects the size of the baSourceID attributes. Both attributes are automatically increased / decreased as you set bmControls and baSourceID. It is also possible to manually increase or decrease bControlSize which has the effect of truncating entries to the new size, or padding entries out with 0x00, for example:

```
$ cat bmControls
0x03
0x05

$ cat bControlSize
2

$ echo 1 > bControlSize
$ cat bmControls
0x03

$ echo 2 > bControlSize
$ cat bmControls
0x03
0x00
```

bNrInPins and baSourceID function in the same way.

12.3.7 Configuring Supported Controls for Camera Terminal and Processing Unit

The Camera Terminal and Processing Units in the UVC chain also have bmControls attributes which function similarly to the same field in an Extension Unit. Unlike XUs however, the meaning of the bitflag for these units is defined in the UVC specification; you should consult the "Camera Terminal Descriptor" and "Processing Unit Descriptor" sections for an enumeration of the flags.

```
# Set the Processing Unit's bmControls, flagging Brightness, Contrast
# and Hue as available controls:
echo 0x05 > $FUNCTION/control/processing/default/bmControls

# Set the Camera Terminal's bmControls, flagging Focus Absolute and
# Focus Relative as available controls:
echo 0x60 > $FUNCTION/control/terminal/camera/default/bmControls
```

If you do not set these fields then by default the Auto-Exposure Mode control for the Camera Terminal and the Brightness control for the Processing Unit will be flagged as available; if they are not supported you should set the field to 0x00.

Note that the size of the bmControls field for a Camera Terminal or Processing Unit is fixed by the UVC specification, and so the bControlSize attribute is read-only here.

12.3.8 Custom Strings Support

String descriptors that provide a textual description for various parts of a USB device can be defined in the usual place within USB configfs, and may then be linked to from the UVC function root or from Extension Unit directories to assign those strings as descriptors:

```
# Create a string descriptor in us-EN and link to it from the function
# root. The name of the link is significant here, as it declares this
# descriptor to be intended for the Interface Association Descriptor.
# Other significant link names at function root are vs0_desc and vs1_desc
# For the VideoStreaming Interface 0/1 Descriptors.

mkdir -p $GADGET/strings/0x409/iad_desc
echo -n "Interface Associaton Descriptor" > $GADGET/strings/0x409/iad_desc/s
ln -s $GADGET/strings/0x409/iad_desc $FUNCTION/iad_desc

# Because the link to a String Descriptor from an Extension Unit clearly
# associates the two, the name of this link is not significant and may
# be set freely.

mkdir -p $GADGET/strings/0x409/xu.0
echo -n "A Very Useful Extension Unit" > $GADGET/strings/0x409/xu.0/s
ln -s $GADGET/strings/0x409/xu.0 $FUNCTION/control/extensions/xu.0
```

12.3.9 The interrupt endpoint

The VideoControl interface has an optional interrupt endpoint which is by default disabled. This is intended to support delayed response control set requests for UVC (which should respond through the interrupt endpoint rather than tying up endpoint 0). At present support for sending data through this endpoint is missing and so it is left disabled to avoid confusion. If you wish to enable it you can do so through the configfs attribute:

```
echo 1 > $FUNCTION/control/enable_interrupt_ep
```

12.3.10 Bandwidth configuration

There are three attributes which control the bandwidth of the USB connection. These live in the function root and can be set within limits:

```
# streaming_interval sets bInterval. Values range from 1..255
echo 1 > $FUNCTION/streaming_interval

# streaming_maxpacket sets wMaxPacketSize. Valid values are 1024/2048/3072
echo 3072 > $FUNCTION/streaming_maxpacket

# streaming_maxburst sets bMaxBurst. Valid values are 1..15
echo 1 > $FUNCTION/streaming_maxburst
```

The values passed here will be clamped to valid values according to the UVC specification (which depend on the speed of the USB connection). To understand how the settings influence

bandwidth you should consult the UVC specifications, but a rule of thumb is that increasing the streaming_maxpacket setting will improve bandwidth (and thus the maximum possible framerate), whilst the same is true for streaming_maxburst provided the USB connection is running at SuperSpeed. Increasing streaming_interval will reduce bandwidth and framerate.

12.4 The userspace application

By itself, the UVC Gadget driver cannot do anything particularly interesting. It must be paired with a userspace program that responds to UVC control requests and fills buffers to be queued to the V4L2 device that the driver creates. How those things are achieved is implementation dependent and beyond the scope of this document, but a reference application can be found at https://gitlab.freedesktop.org/camera/uvc-gadget

CHAPTER

THIRTEEN

GADGET TESTING

This file summarizes information on basic testing of USB functions provided by gadgets.

13.1 1. ACM function

The function is provided by usb_f_acm.ko module.

13.1.1 Function-specific configfs interface

The function name to use when creating the function directory is “acm”. The ACM function provides just one attribute in its function directory:

port_num

The attribute is read-only.

There can be at most 4 ACM/generic serial/OBEX ports in the system.

13.1.2 Testing the ACM function

On the host:

```
cat > /dev/ttyACM<X>
```

On the device:

```
cat /dev/ttyGS<Y>
```

then the other way round

On the device:

```
cat > /dev/ttyGS<Y>
```

On the host:

```
cat /dev/ttyACM<X>
```

13.2 2. ECM function

The function is provided by usb_f_ecm.ko module.

13.2.1 Function-specific configfs interface

The function name to use when creating the function directory is “ecm”. The ECM function provides these attributes in its function directory:

ifname	network device interface name associated with this function instance
qmult	queue length multiplier for high and super speed
host_addr	MAC address of host’s end of this Ethernet over USB link
dev_addr	MAC address of device’s end of this Ethernet over USB link

and after creating the functions/ecm.<instance name> they contain default values: qmult is 5, dev_addr and host_addr are randomly selected. The ifname can be written to if the function is not bound. A write must be an interface pattern such as “usb%d”, which will cause the net core to choose the next free usbX interface. By default, it is set to “usb%d”.

13.2.2 Testing the ECM function

Configure IP addresses of the device and the host. Then:

On the device:

```
ping <host's IP>
```

On the host:

```
ping <device's IP>
```

13.3 3. ECM subset function

The function is provided by usb_f_ecm_subset.ko module.

13.3.1 Function-specific configfs interface

The function name to use when creating the function directory is “geth”. The ECM subset function provides these attributes in its function directory:

ifname	network device interface name associated with this function instance
qmult	queue length multiplier for high and super speed
host_addr	MAC address of host’s end of this Ethernet over USB link
dev_addr	MAC address of device’s end of this Ethernet over USB link

and after creating the functions/ecm.<instance name> they contain default values: qmult is 5, dev_addr and host_addr are randomly selected. The ifname can be written to if the function is not bound. A write must be an interface pattern such as “usb%d”, which will cause the net core to choose the next free usbX interface. By default, it is set to “usb%d”.

13.3.2 Testing the ECM subset function

Configure IP addresses of the device and the host. Then:

On the device:

```
ping <host's IP>
```

On the host:

```
ping <device's IP>
```

13.4 4. EEM function

The function is provided by usb_f_eem.ko module.

13.4.1 Function-specific configfs interface

The function name to use when creating the function directory is “eem”. The EEM function provides these attributes in its function directory:

ifname	network device interface name associated with this function instance
qmult	queue length multiplier for high and super speed
host_addr	MAC address of host’s end of this Ethernet over USB link
dev_addr	MAC address of device’s end of this Ethernet over USB link

and after creating the functions/eem.<instance name> they contain default values: qmult is 5, dev_addr and host_addr are randomly selected. The ifname can be written to if the function is not bound. A write must be an interface pattern such as “usb%d”, which will cause the net core to choose the next free usbX interface. By default, it is set to “usb%d”.

13.4.2 Testing the EEM function

Configure IP addresses of the device and the host. Then:

On the device:

```
ping <host's IP>
```

On the host:

```
ping <device's IP>
```

13.5 5. FFS function

The function is provided by usb_f_fs.ko module.

13.5.1 Function-specific configfs interface

The function name to use when creating the function directory is “ffs”. The function directory is intentionally empty and not modifiable.

After creating the directory there is a new instance (a “device”) of FunctionFS available in the system. Once a “device” is available, the user should follow the standard procedure for using FunctionFS (mount it, run the userspace process which implements the function proper). The gadget should be enabled by writing a suitable string to usb_gadget/<gadget>/UDC.

13.5.2 Testing the FFS function

On the device: start the function’s userspace daemon, enable the gadget

On the host: use the USB function provided by the device

13.6 6. HID function

The function is provided by usb_f_hid.ko module.

13.6.1 Function-specific configfs interface

The function name to use when creating the function directory is “hid”. The HID function provides these attributes in its function directory:

protocol	HID protocol to use
report_desc	data to be used in HID reports, except data passed with /dev/hidg<X>
report_length	HID report length
subclass	HID subclass to use

For a keyboard the protocol and the subclass are 1, the report_length is 8, while the report_desc is:

```
$ hd my_report_desc
00000000  05 01 09 06 a1 01 05 07  19 e0 29 e7 15 00 25 01  |..........)...%.|
00000010  75 01 95 08 81 02 95 01  75 08 81 03 95 05 75 01  |u.......u.....u.|
00000020  05 08 19 01 29 05 91 02  95 01 75 03 91 03 95 06  |....).....u.....|
00000030  75 08 15 00 25 65 05 07  19 00 29 65 81 00 c0     |u...%e....)e...|
0000003f
```

Such a sequence of bytes can be stored to the attribute with echo:

```
$ echo -ne \\x05\\x01\\x09\\x06\\xa1.....
```

13.6.2 Testing the HID function

Device:

- create the gadget
- connect the gadget to a host, preferably not the one used to control the gadget
- run a program which writes to /dev/hidg<N>, e.g. a userspace program found in *Linux USB HID gadget driver*:

```
$ ./hid_gadget_test /dev/hidg0 keyboard
```

Host:

- observe the keystrokes from the gadget

13.7 7. LOOPBACK function

The function is provided by usb_f_ss_lb.ko module.

13.7.1 Function-specific configfs interface

The function name to use when creating the function directory is “Loopback”. The LOOPBACK function provides these attributes in its function directory:

qlen	depth of loopback queue
bulk_buflen	buffer length

13.7.2 Testing the LOOPBACK function

device: run the gadget

host: test-usb (tools/usb/testusb.c)

13.8 8. MASS STORAGE function

The function is provided by usb_f_mass_storage.ko module.

13.8.1 Function-specific configfs interface

The function name to use when creating the function directory is “mass_storage”. The MASS STORAGE function provides these attributes in its directory: files:

stall	Set to permit function to halt bulk endpoints. Disabled on some USB devices known not to work correctly. You should set it to true.
num_buffers	Number of pipeline buffers. Valid numbers are 2..4. Available only if CONFIG_USB_GADGET_DEBUG_FILES is set.

and a default lun.0 directory corresponding to SCSI LUN #0.

A new lun can be added with mkdir:

```
$ mkdir functions/mass_storage.0/partition.5
```

Lun numbering does not have to be continuous, except for lun #0 which is created by default. A maximum of 8 luns can be specified and they all must be named following the <name>.<number> scheme. The numbers can be 0..8. Probably a good convention is to name the luns “lun.<number>”, although it is not mandatory.

In each lun directory there are the following attribute files:

file	The path to the backing file for the LUN. Required if LUN is not marked as removable.
ro	Flag specifying access to the LUN shall be read-only. This is implied if CD-ROM emulation is enabled as well as when it was impossible to open “filename” in R/W mode.
removable	Flag specifying that LUN shall be indicated as being removable.
cdrom	Flag specifying that LUN shall be reported as being a CD-ROM.
nofua	Flag specifying that FUA flag in SCSI WRITE(10,12)
forced_eject	This write-only file is useful only when the function is active. It causes the backing file to be forcibly detached from the LUN, regardless of whether the host has allowed it. Any non-zero number of bytes written will result in ejection.

13.8.2 Testing the MASS STORAGE function

device: connect the gadget, enable it host: dmesg, see the USB drives appear (if system configured to automatically mount)

13.9 9. MIDI function

The function is provided by usb_f_midi.ko module.

13.9.1 Function-specific configfs interface

The function name to use when creating the function directory is “midi”. The MIDI function provides these attributes in its function directory:

buflen	MIDI buffer length
id	ID string for the USB MIDI adapter
in_ports	number of MIDI input ports
index	index value for the USB MIDI adapter
out_ports	number of MIDI output ports
qlen	USB read request queue length

13.9.2 Testing the MIDI function

There are two cases: playing a mid from the gadget to the host and playing a mid from the host to the gadget.

1) Playing a mid from the gadget to the host:

host:

```
$ arecordmidi -l
 Port    Client name                      Port name
 14:0    Midi Through                     Midi Through Port-0
 24:0    MIDI Gadget                      MIDI Gadget MIDI 1
$ arecordmidi -p 24:0 from_gadget.mid
```

gadget:

```
$ aplaymidi -l
 Port    Client name                      Port name
 20:0    f_midi                           f_midi

$ aplaymidi -p 20:0 to_host.mid
```

2) Playing a mid from the host to the gadget

gadget:

```
$ arecordmidi -l
 Port    Client name                      Port name
 20:0    f_midi                           f_midi

$ arecordmidi -p 20:0 from_host.mid
```

host:

```
$ aplaymidi -l
 Port    Client name                      Port name
 14:0    Midi Through                     Midi Through Port-0
 24:0    MIDI Gadget                      MIDI Gadget MIDI 1

$ aplaymidi -p24:0 to_gadget.mid
```

The from_gadget.mid should sound identical to the to_host.mid.

The from_host.id should sound identical to the to_gadget.mid.

MIDI files can be played to speakers/headphones with e.g. timidity installed:

```
$ aplaymidi -l
 Port    Client name                      Port name
 14:0    Midi Through                     Midi Through Port-0
 24:0    MIDI Gadget                      MIDI Gadget MIDI 1
128:0    TiMidity                         TiMidity port 0
128:1    TiMidity                         TiMidity port 1
128:2    TiMidity                         TiMidity port 2
128:3    TiMidity                         TiMidity port 3

$ aplaymidi -p 128:0 file.mid
```

MIDI ports can be logically connected using the aconnect utility, e.g.:

```
$ aconnect 24:0 128:0 # try it on the host
```

After the gadget's MIDI port is connected to timidity's MIDI port, whatever is played at the gadget side with aplaymidi -l is audible in host's speakers/headphones.

13.10 10. NCM function

The function is provided by usb_f_ncm.ko module.

13.10.1 Function-specific configfs interface

The function name to use when creating the function directory is "ncm". The NCM function provides these attributes in its function directory:

ifname	network device interface name associated with this function instance
qmult	queue length multiplier for high and super speed
host_addr	MAC address of host's end of this Ethernet over USB link
dev_addr	MAC address of device's end of this Ethernet over USB link

and after creating the functions/ncm.<instance name> they contain default values: qmult is 5, dev_addr and host_addr are randomly selected. The ifname can be written to if the function is not bound. A write must be an interface pattern such as "usb%d", which will cause the net core to choose the next free usbX interface. By default, it is set to "usb%d".

13.10.2 Testing the NCM function

Configure IP addresses of the device and the host. Then:

On the device:

```
ping <host's IP>
```

On the host:

```
ping <device's IP>
```

13.11 11. OBEX function

The function is provided by usb_f_obex.ko module.

13.11.1 Function-specific configfs interface

The function name to use when creating the function directory is “obex”. The OBEX function provides just one attribute in its function directory:

port_num

The attribute is read-only.

There can be at most 4 ACM/generic serial/OBEX ports in the system.

13.11.2 Testing the OBEX function

On device:

```
seriald -f /dev/ttyGS<Y> -s 1024
```

On host:

```
serialc -v <vendorID> -p <productID> -i<interface#> -a1 -s1024 \
        -t<out endpoint addr> -r<in endpoint addr>
```

where seriald and serialc are Felipe’s utilities found here:

https://github.com/felipebalbi/usb-tools.git master

13.12 12. PHONET function

The function is provided by usb_f_phonet.ko module.

13.12.1 Function-specific configfs interface

The function name to use when creating the function directory is “phonet”. The PHONET function provides just one attribute in its function directory:

ifname	network device interface name associated with this function instance

13.12.2 Testing the PHONET function

It is not possible to test the SOCK_STREAM protocol without a specific piece of hardware, so only SOCK_DGRAM has been tested. For the latter to work, in the past I had to apply the patch mentioned here:

http://www.spinics.net/lists/linux-usb/msg85689.html

These tools are required:

git://git.gitorious.org/meego-cellular/phonet-utils.git

On the host:

```
$ ./phonet -a 0x10 -i usbpn0
$ ./pnroute add 0x6c usbpn0
$./pnroute add 0x10 usbpn0
$ ifconfig usbpn0 up
```

On the device:

```
$ ./phonet -a 0x6c -i upnlink0
$ ./pnroute add 0x10 upnlink0
$ ifconfig upnlink0 up
```

Then a test program can be used:

```
http://www.spinics.net/lists/linux-usb/msg85690.html
```

On the device:

```
$ ./pnxmit -a 0x6c -r
```

On the host:

```
$ ./pnxmit -a 0x10 -s 0x6c
```

As a result some data should be sent from host to device. Then the other way round:

On the host:

```
$ ./pnxmit -a 0x10 -r
```

On the device:

```
$ ./pnxmit -a 0x6c -s 0x10
```

13.13 13. RNDIS function

The function is provided by usb_f_rndis.ko module.

13.13.1 Function-specific configfs interface

The function name to use when creating the function directory is “rndis”. The RNDIS function provides these attributes in its function directory:

ifname	network device interface name associated with this function instance
qmult	queue length multiplier for high and super speed
host_addr	MAC address of host’s end of this Ethernet over USB link
dev_addr	MAC address of device’s end of this Ethernet over USB link

and after creating the functions/rndis.<instance name> they contain default values: qmult is 5, dev_addr and host_addr are randomly selected. The ifname can be written to if the function is not bound. A write must be an interface pattern such as “usb%d”, which will cause the net core to choose the next free usbX interface. By default, it is set to “usb%d”.

13.13.2 Testing the RNDIS function

Configure IP addresses of the device and the host. Then:

On the device:

```
ping <host's IP>
```

On the host:

```
ping <device's IP>
```

13.14 14. SERIAL function

The function is provided by usb_f_gser.ko module.

13.14.1 Function-specific configfs interface

The function name to use when creating the function directory is “gser”. The SERIAL function provides just one attribute in its function directory:

port_num

The attribute is read-only.

There can be at most 4 ACM/generic serial/OBEX ports in the system.

13.14.2 Testing the SERIAL function

On host:

```
insmod usbserial
echo VID PID >/sys/bus/usb-serial/drivers/generic/new_id
```

On host:

```
cat > /dev/ttyUSB<X>
```

On target:

```
cat /dev/ttyGS<Y>
```

then the other way round

On target:

```
cat > /dev/ttyGS<Y>
```

On host:

```
cat /dev/ttyUSB<X>
```

13.15 15. SOURCESINK function

The function is provided by usb_f_ss_lb.ko module.

13.15.1 Function-specific configfs interface

The function name to use when creating the function directory is “SourceSink”. The SOURCESINK function provides these attributes in its function directory:

pattern	0 (all zeros), 1 (mod63), 2 (none)
isoc_interval	1..16
isoc_maxpacket	0 - 1023 (fs), 0 - 1024 (hs/ss)
isoc_mult	0..2 (hs/ss only)
isoc_maxburst	0..15 (ss only)
bulk_buflen	buffer length
bulk_qlen	depth of queue for bulk
iso_qlen	depth of queue for iso

13.15.2 Testing the SOURCESINK function

device: run the gadget

host: test-usb (tools/usb/testusb.c)

13.16 16. UAC1 function (legacy implementation)

The function is provided by usb_f_uac1_legacy.ko module.

13.16.1 Function-specific configfs interface

The function name to use when creating the function directory is “uac1_legacy”. The uac1 function provides these attributes in its function directory:

audio_buf_size	audio buffer size
fn_cap	capture pcm device file name
fn_cntl	control device file name
fn_play	playback pcm device file name
req_buf_size	ISO OUT endpoint request buffer size
req_count	ISO OUT endpoint request count

The attributes have sane default values.

13.16.2 Testing the UAC1 function

device: run the gadget

host:

```
aplay -l # should list our USB Audio Gadget
```

13.17 17. UAC2 function

The function is provided by usb_f_uac2.ko module.

13.17.1 Function-specific configfs interface

The function name to use when creating the function directory is “uac2”. The uac2 function provides these attributes in its function directory:

c_chmask	capture channel mask
c_srate	list of capture sampling rates (comma-separated)
c_ssize	capture sample size (bytes)
c_sync	capture synchronization type (async/adaptive)
c_mute_present	capture mute control enable
c_volume_present	capture volume control enable
c_volume_min	capture volume control min value (in 1/256 dB)
c_volume_max	capture volume control max value (in 1/256 dB)
c_volume_res	capture volume control resolution (in 1/256 dB)
c_hs_bint	capture bInterval for HS/SS (1-4: fixed, 0: auto)
fb_max	maximum extra bandwidth in async mode
p_chmask	playback channel mask
p_srate	list of playback sampling rates (comma-separated)
p_ssize	playback sample size (bytes)
p_mute_present	playback mute control enable
p_volume_present	playback volume control enable
p_volume_min	playback volume control min value (in 1/256 dB)
p_volume_max	playback volume control max value (in 1/256 dB)
p_volume_res	playback volume control resolution (in 1/256 dB)
p_hs_bint	playback bInterval for HS/SS (1-4: fixed, 0: auto)
req_number	the number of pre-allocated request for both capture and playback
function_name	name of the interface

The attributes have sane default values.

13.17.2 Testing the UAC2 function

device: run the gadget host: aplay -l # should list our USB Audio Gadget

This function does not require real hardware support, it just sends a stream of audio data to/from the host. In order to actually hear something at the device side, a command similar to this must be used at the device side:

```
$ arecord -f dat -t wav -D hw:2,0 | aplay -D hw:0,0 &
```

e.g.:

```
$ arecord -f dat -t wav -D hw:CARD=UAC2Gadget,DEV=0 | \
  aplay -D default:CARD=OdroidU3
```

13.18 18. UVC function

The function is provided by usb_f_uvc.ko module.

13.18.1 Function-specific configfs interface

The function name to use when creating the function directory is “uvc”. The uvc function provides these attributes in its function directory:

streaming_interval	interval for polling endpoint for data transfers
streaming_maxburst	bMaxBurst for super speed companion descriptor
streaming_maxpacket	maximum packet size this endpoint is capable of sending or receiving when this configuration is selected
function_name	name of the interface

There are also “control” and “streaming” subdirectories, each of which contain a number of their subdirectories. There are some sane defaults provided, but the user must provide the following:

control header	create in control/header, link from control/class/fs and/or control/class/ss
streaming header	create in streaming/header, link from streaming/class/fs and/or streaming/class/hs and/or streaming/class/ss
format description	create in streaming/mjpeg and/or streaming/uncompressed
frame description	create in streaming/mjpeg/<format> and/or in streaming/uncompressed/<format>

Each frame description contains frame interval specification, and each such specification consists of a number of lines with an interval value in each line. The rules stated above are best illustrated with an example:

```
# mkdir functions/uvc.usb0/control/header/h
# cd functions/uvc.usb0/control/
# ln -s header/h class/fs
# ln -s header/h class/ss
# mkdir -p functions/uvc.usb0/streaming/uncompressed/u/360p
# cat <<EOF > functions/uvc.usb0/streaming/uncompressed/u/360p/dwFrameInterval
666666
1000000
5000000
EOF
# cd $GADGET_CONFIGFS_ROOT
```

```
# mkdir functions/uvc.usb0/streaming/header/h
# cd functions/uvc.usb0/streaming/header/h
# ln -s ../../uncompressed/u
# cd ../../class/fs
# ln -s ../../header/h
# cd ../../class/hs
# ln -s ../../header/h
# cd ../../class/ss
# ln -s ../../header/h
```

13.18.2 Testing the UVC function

device: run the gadget, modprobe vivid:

```
# uvc-gadget -u /dev/video<uvc video node #> -v /dev/video<vivid video node #>
```

where uvc-gadget is this program:

http://git.ideasonboard.org/uvc-gadget.git

with these patches:

http://www.spinics.net/lists/linux-usb/msg99220.html

host:

```
luvcview -f yuv
```

13.19 19. PRINTER function

The function is provided by usb_f_printer.ko module.

13.19.1 Function-specific configfs interface

The function name to use when creating the function directory is “printer”. The printer function provides these attributes in its function directory:

pnp_string	Data to be passed to the host in pnp string
q_len	Number of requests per endpoint

13.19.2 Testing the PRINTER function

The most basic testing:

device: run the gadget:

```
# ls -l /devices/virtual/usb_printer_gadget/
```

should show g_printer<number>.

If udev is active, then /dev/g_printer<number> should appear automatically.

host:

If udev is active, then e.g. /dev/usb/lp0 should appear.

host->device transmission:

device:

```
# cat /dev/g_printer<number>
```

host:

```
# cat > /dev/usb/lp0
```

device->host transmission:

```
# cat > /dev/g_printer<number>
```

host:

```
# cat /dev/usb/lp0
```

More advanced testing can be done with the prn_example described in *Linux USB Printer Gadget Driver*.

13.20 20. UAC1 function (virtual ALSA card, using u_audio API)

The function is provided by usb_f_uac1.ko module. It will create a virtual ALSA card and the audio streams are simply sinked to and sourced from it.

13.20.1 Function-specific configfs interface

The function name to use when creating the function directory is “uac1”. The uac1 function provides these attributes in its function directory:

c_chmask	capture channel mask
c_srate	list of capture sampling rates (comma-separated)
c_ssize	capture sample size (bytes)
c_mute_present	capture mute control enable
c_volume_present	capture volume control enable
c_volume_min	capture volume control min value (in 1/256 dB)
c_volume_max	capture volume control max value (in 1/256 dB)
c_volume_res	capture volume control resolution (in 1/256 dB)
p_chmask	playback channel mask
p_srate	list of playback sampling rates (comma-separated)
p_ssize	playback sample size (bytes)
p_mute_present	playback mute control enable
p_volume_present	playback volume control enable
p_volume_min	playback volume control min value (in 1/256 dB)
p_volume_max	playback volume control max value (in 1/256 dB)
p_volume_res	playback volume control resolution (in 1/256 dB)
req_number	the number of pre-allocated requests for both capture and playback
function_name	name of the interface

The attributes have sane default values.

13.20.2 Testing the UAC1 function

device: run the gadget host: aplay -l # should list our USB Audio Gadget

This function does not require real hardware support, it just sends a stream of audio data to/from the host. In order to actually hear something at the device side, a command similar to this must be used at the device side:

```
$ arecord -f dat -t wav -D hw:2,0 | aplay -D hw:0,0 &
```

e.g.:

```
$ arecord -f dat -t wav -D hw:CARD=UAC1Gadget,DEV=0 | \
  aplay -D default:CARD=OdroidU3
```

13.21 21. MIDI2 function

The function is provided by usb_f_midi2.ko module. It will create a virtual ALSA card containing a UMP rawmidi device where the UMP packet is looped back. In addition, a legacy rawmidi device is created. The UMP rawmidi is bound with ALSA sequencer clients, too.

13.21.1 Function-specific configfs interface

The function name to use when creating the function directory is “midi2”. The midi2 function provides these attributes in its function directory as the card top-level information:

process_ump	Bool flag to process UMP Stream messages (0 or 1)
static_block	Bool flag for static blocks (0 or 1)
iface_name	Optional interface name string

The directory contains a subdirectory “ep.0”, and this provides the attributes for a UMP Endpoint (which is a pair of USB MIDI Endpoints):

protocol_caps	MIDI protocol capabilities; 1: MIDI 1.0, 2: MIDI 2.0, or 3: both protocols
protocol	Default MIDI protocol (either 1 or 2)
ep_name	UMP Endpoint name string
product_id	Product ID string
manufacturer	Manufacture ID number (24 bit)
family	Device family ID number (16 bit)
model	Device model ID number (16 bit)
sw_revision	Software revision (32 bit)

Each Endpoint subdirectory contains a subdirectory “block.0”, which represents the Function Block for Block 0 information. Its attributes are:

name	Function Block name string
direction	Direction of this FB 1: input, 2: output, or 3: bidirectional
first_group	The first UMP Group number (0-15)
num_groups	The number of groups in this FB (1-16)
midi1_first_group	The first UMP Group number for MIDI 1.0 (0-15)
midi1_num_groups	The number of groups for MIDI 1.0 (0-16)
ui_hint	UI-hint of this FB 0: unknown, 1: receiver, 2: sender, 3: both
midi_ci_verison	Supported MIDI-CI version number (8 bit)
is_midi1	Legacy MIDI 1.0 device (0-2) 0: MIDI 2.0 device, 1: MIDI 1.0 without restriction, or 2: MIDI 1.0 with low speed
sysex8_streams	Max number of SysEx8 streams (8 bit)
active	Bool flag for FB activity (0 or 1)

If multiple Function Blocks are required, you can add more Function Blocks by creating subdirectories “block.<num>” with the corresponding Function Block number (1, 2, ...). The FB subdirectories can be dynamically removed, too. Note that the Function Block numbers must be continuous.

Similarly, if you multiple UMP Endpoints are required, you can add more Endpoints by creating subdirectories “ep.<num>”. The number must be continuous.

For emulating the old MIDI 2.0 device without UMP v1.1 support, pass 0 to *process_ump* flag. Then the whole UMP v1.1 requests are ignored.

13.21.2 Testing the MIDI2 function

On the device: run the gadget, and running:

```
$ cat /proc/asound/cards
```

will show a new sound card containing a MIDI2 device.

OTOH, on the host:

```
$ cat /proc/asound/cards
```

will show a new sound card containing either MIDI1 or MIDI2 device, depending on the USB audio driver configuration.

On both, when ALSA sequencer is enabled on the host, you can find the UMP MIDI client such as “MIDI 2.0 Gadget”.

As the driver simply loops back the data, there is no need for a real device just for testing.

For testing a MIDI input from the gadget to the host (e.g. emulating a MIDI keyboard), you can send a MIDI stream like the following.

On the gadget:

```
$ aconnect -o
....
client 20: 'MIDI 2.0 Gadget' [type=kernel,card=1]
    0 'MIDI 2.0        '
    1 'Group 1 (MIDI 2.0 Gadget I/O)'
$ aplaymidi -p 20:1 to_host.mid
```

On the host:

```
$ aconnect -i
....
client 24: 'MIDI 2.0 Gadget' [type=kernel,card=2]
    0 'MIDI 2.0        '
    1 'Group 1 (MIDI 2.0 Gadget I/O)'
$ arecordmidi -p 24:1 from_gadget.mid
```

If you have a UMP-capable application, you can use the UMP port to send/receive the raw UMP packets, too. For example, aseqdump program with UMP support can receive from UMP port. On the host:

```
$ aseqdump -u 2 -p 24:1
Waiting for data. Press Ctrl+C to end.
Source  Group    Event                  Ch  Data
 24:1   Group  0, Program change          0, program 0, Bank select 0:0
 24:1   Group  0, Channel pressure        0, value 0x80000000
```

For testing a MIDI output to the gadget to the host (e.g. emulating a MIDI synth), it’ll be just other way round.

On the gadget:

```
$ arecordmidi -p 20:1 from_host.mid
```

On the host:

```
$ aplaymidi -p 24:1 to_gadget.mid
```

The access to MIDI 1.0 on altset 0 on the host is supported, and it's translated from/to UMP packets on the gadget. It's bound to only Function Block 0.

The current operation mode can be observed in ALSA control element "Operation Mode" for SND_CTL_IFACE_RAWMIDI. For example:

```
$ amixer -c1 contents
numid=1,iface=RAWMIDI,name='Operation Mode'
  ; type=INTEGER,access=r--v----,values=1,min=0,max=2,step=0
  : values=2
```

where 0 = unused, 1 = MIDI 1.0 (altset 0), 2 = MIDI 2.0 (altset 1). The example above shows it's running in 2, i.e. MIDI 2.0.

CHAPTER

FOURTEEN

INFINITY USB UNLIMITED README

Hi all,

This module provide a serial interface to use your IUU unit in phoenix mode. Loading this module will bring a ttyUSB[0-x] interface. This driver must be used by your favorite application to pilot the IUU

This driver is still in beta stage, so bugs can occur and your system may freeze. As far I now, I never had any problem with it, but I'm not a real guru, so don't blame me if your system is unstable

You can plug more than one IUU. Every unit will have his own device file(/dev/ttyUSB0,/dev/ttyUSB1,...)

14.1 How to tune the reader speed?

A few parameters can be used at load time To use parameters, just unload the module if it is already loaded and use modprobe iuu_phoenix param=value. In case of prebuilt module, use the command insmod iuu_phoenix param=value.

Example:

```
modprobe iuu_phoenix clockmode=3
```

The parameters are:

clockmode:
1=3Mhz579,2=3Mhz680,3=6Mhz (int)

boost:
overclock boost percent 100 to 500 (int)

cdmode:
Card detect mode 0=none, 1=CD, 2=!CD, 3=DSR, 4=!DSR, 5=CTS, 6=!CTS, 7=RING, 8=!RING (int)

xmas:
xmas color enabled or not (bool)

debug:
Debug enabled or not (bool)

- clockmode will provide 3 different base settings commonly adopted by different software:

  1. 3Mhz579

2. 3Mhz680
3. 6Mhz

- boost provide a way to overclock the reader (my favorite :-)) For example to have best performance than a simple clockmode=3, try this:

```
modprobe boost=195
```

  This will put the reader in a base of 3Mhz579 but boosted a 195 % ! the real clock will be now : 6979050 Hz (6Mhz979) and will increase the speed to a score 10 to 20% better than the simple clockmode=3 !!!

- cdmode permit to setup the signal used to inform the userland (ioctl answer) if the card is present or not. Eight signals are possible.
- xmas is completely useless except for your eyes. This is one of my friend who was so sad to have a nice device like the iuu without seeing all color range available. So I have added this option to permit him to see a lot of color (each activity change the color and the frequency randomly)
- debug will produce a lot of debugging messages...

14.2 Last notes

Don’t worry about the serial settings, the serial emulation is an abstraction, so use any speed or parity setting will work. (This will not change anything).Later I will perhaps use this settings to deduce de boost but is that feature really necessary ? The autodetect feature used is the serial CD. If that doesn’t work for your software, disable detection mechanism in it.

Have fun !

Alain Degreffe

eczema(at)ecze.com

CHAPTER

FIFTEEN

MASS STORAGE GADGET (MSG)

15.1 Overview

Mass Storage Gadget (or MSG) acts as a USB Mass Storage device, appearing to the host as a disk or a CD-ROM drive. It supports multiple logical units (LUNs). Backing storage for each LUN is provided by a regular file or a block device, access can be limited to read-only, and gadget can indicate that it is removable and/or CD-ROM (the latter implies read-only access).

Its requirements are modest; only a bulk-in and a bulk-out endpoint are needed. The memory requirement amounts to two 16K buffers. Support is included for full-speed, high-speed and SuperSpeed operation.

Note that the driver is slightly non-portable in that it assumes a single memory/DMA buffer will be usable for bulk-in and bulk-out endpoints. With most device controllers this is not an issue, but there may be some with hardware restrictions that prevent a buffer from being used by more than one endpoint.

This document describes how to use the gadget from user space, its relation to mass storage function (or MSF) and different gadgets using it, and how it differs from File Storage Gadget (or FSG) (which is no longer included in Linux). It will talk only briefly about how to use MSF within composite gadgets.

15.2 Module parameters

The mass storage gadget accepts the following mass storage specific module parameters:

- file=filename[,filename...]

  This parameter lists paths to files or block devices used for backing storage for each logical unit. There may be at most FSG_MAX_LUNS (8) LUNs set. If more files are specified, they will be silently ignored. See also “luns” parameter.

  BEWARE that if a file is used as a backing storage, it may not be modified by any other process. This is because the host assumes the data does not change without its knowledge. It may be read, but (if the logical unit is writable) due to buffering on the host side, the contents are not well defined.

  The size of the logical unit will be rounded down to a full logical block. The logical block size is 2048 bytes for LUNs simulating CD-ROM, block size of the device if the backing file is a block device, or 512 bytes otherwise.

- removable=b[,b...]

  This parameter specifies whether each logical unit should be removable. “b” here is either “y”, “Y” or “1” for true or “n”, “N” or “0” for false.

  If this option is set for a logical unit, gadget will accept an “eject” SCSI request (Start/Stop Unit). When it is sent, the backing file will be closed to simulate ejection and the logical unit will not be mountable by the host until a new backing file is specified by userspace on the device (see “sysfs entries” section).

  If a logical unit is not removable (the default), a backing file must be specified for it with the “file” parameter as the module is loaded. The same applies if the module is built in, no exceptions.

  The default value of the flag is false, *HOWEVER* it used to be true. This has been changed to better match File Storage Gadget and because it seems like a saner default after all. Thus to maintain compatibility with older kernels, it’s best to specify the default values. Also, if one relied on old default, explicit “n” needs to be specified now.

  Note that “removable” means the logical unit’s media can be ejected or removed (as is true for a CD-ROM drive or a card reader). It does *not* mean that the entire gadget can be unplugged from the host; the proper term for that is “hot-unpluggable”.

- cdrom=b[,b...]

  This parameter specifies whether each logical unit should simulate CD-ROM. The default is false.

- ro=b[,b...]

  This parameter specifies whether each logical unit should be reported as read only. This will prevent host from modifying the backing files.

  Note that if this flag for given logical unit is false but the backing file could not be opened in read/write mode, the gadget will fall back to read only mode anyway.

  The default value for non-CD-ROM logical units is false; for logical units simulating CD-ROM it is forced to true.

- nofua=b[,b...]

  This parameter specifies whether FUA flag should be ignored in SCSI Write10 and Write12 commands sent to given logical units.

  MS Windows mounts removable storage in “Removal optimised mode” by default. All the writes to the media are synchronous, which is achieved by setting the FUA (Force Unit Access) bit in SCSI Write(10,12) commands. This forces each write to wait until the data has actually been written out and prevents I/O requests aggregation in block layer dramatically decreasing performance.

  Note that this may mean that if the device is powered from USB and the user unplugs the device without unmounting it first (which at least some Windows users do), the data may be lost.

  The default value is false.

- luns=N

This parameter specifies number of logical units the gadget will have. It is limited by FSG_MAX_LUNS (8) and higher value will be capped.

If this parameter is provided, and the number of files specified in “file” argument is greater then the value of “luns”, all excess files will be ignored.

If this parameter is not present, the number of logical units will be deduced from the number of files specified in the “file” parameter. If the file parameter is missing as well, one is assumed.

- stall=b

  Specifies whether the gadget is allowed to halt bulk endpoints. The default is determined according to the type of USB device controller, but usually true.

In addition to the above, the gadget also accepts the following parameters defined by the composite framework (they are common to all composite gadgets so just a quick listing):

- idVendor -- USB Vendor ID (16 bit integer)
- idProduct -- USB Product ID (16 bit integer)
- bcdDevice -- USB Device version (BCD) (16 bit integer)
- iManufacturer -- USB Manufacturer string (string)
- iProduct -- USB Product string (string)
- iSerialNumber -- SerialNumber string (string)

15.3 sysfs entries

For each logical unit, the gadget creates a directory in the sysfs hierarchy. Inside of it the following three files are created:

- file

  When read it returns the path to the backing file for the given logical unit. If there is no backing file (possible only if the logical unit is removable), the content is empty.

  When written into, it changes the backing file for given logical unit. This change can be performed even if given logical unit is not specified as removable (but that may look strange to the host). It may fail, however, if host disallowed medium removal with the Prevent-Allow Medium Removal SCSI command.

- ro

  Reflects the state of ro flag for the given logical unit. It can be read any time, and written to when there is no backing file open for given logical unit.

- nofua

  Reflects the state of nofua flag for given logical unit. It can be read and written.

- forced_eject

When written into, it causes the backing file to be forcibly detached from the LUN, regardless of whether the host has allowed it. The content doesn't matter, any non-zero number of bytes written will result in ejection.

Can not be read.

Other then those, as usual, the values of module parameters can be read from /sys/module/g_mass_storage/parameters/* files.

15.4 Other gadgets using mass storage function

The Mass Storage Gadget uses the Mass Storage Function to handle mass storage protocol. As a composite function, MSF may be used by other gadgets as well (eg. g_multi and acm_ms).

All of the information in previous sections are valid for other gadgets using MSF, except that support for mass storage related module parameters may be missing, or the parameters may have a prefix. To figure out whether any of this is true one needs to consult the gadget's documentation or its source code.

For examples of how to include mass storage function in gadgets, one may take a look at mass_storage.c, acm_ms.c and multi.c (sorted by complexity).

15.5 Relation to file storage gadget

The Mass Storage Function and thus the Mass Storage Gadget has been based on the File Storage Gadget. The difference between the two is that MSG is a composite gadget (ie. uses the composite framework) while file storage gadget was a traditional gadget. From userspace point of view this distinction does not really matter, but from kernel hacker's point of view, this means that (i) MSG does not duplicate code needed for handling basic USB protocol commands and (ii) MSF can be used in any other composite gadget.

Because of that, File Storage Gadget has been removed in Linux 3.8. All users need to transition to the Mass Storage Gadget. The two gadgets behave mostly the same from the outside except:

1. In FSG the "removable" and "cdrom" module parameters set the flag for all logical units whereas in MSG they accept a list of y/n values for each logical unit. If one uses only a single logical unit this does not matter, but if there are more, the y/n value needs to be repeated for each logical unit.
2. FSG's "serial", "vendor", "product" and "release" module parameters are handled in MSG by the composite layer's parameters named respectively: "iSerialnumber", "idVendor", "idProduct" and "bcdDevice".
3. MSG does not support FSG's test mode, thus "transport", "protocol" and "buflen" FSG's module parameters are not supported. MSG always uses SCSI protocol with bulk only transport mode and 16 KiB buffers.

CHAPTER

SIXTEEN

USB 7-SEGMENT NUMERIC DISPLAY

Manufactured by Delcom Engineering

16.1 Device Information

USB VENDOR_ID 0x0fc5 USB PRODUCT_ID 0x1227 Both the 6 character and 8 character displays have PRODUCT_ID, and according to Delcom Engineering no queryable information can be obtained from the device to tell them apart.

16.2 Device Modes

By default, the driver assumes the display is only 6 characters The mode for 6 characters is:

MSB 0x06; LSB 0x3f

For the 8 character display:

MSB 0x08; LSB 0xff

The device can accept “text” either in raw, hex, or ascii textmode. raw controls each segment manually, hex expects a value between 0-15 per character, ascii expects a value between ‘0’-‘9’ and ‘A’-‘F’. The default is ascii.

16.3 Device Operation

1. Turn on the device: echo 1 > /sys/bus/usb/.../powered
2. Set the device’s mode: echo $mode_msb > /sys/bus/usb/.../mode_msb echo $mode_lsb > /sys/bus/usb/.../mode_lsb
3. Set the textmode: echo $textmode > /sys/bus/usb/.../textmode
4. set the text (for example): echo “123ABC” > /sys/bus/usb/.../text (ascii) echo “A1B2” > /sys/bus/usb/.../text (ascii) echo -ne “x01x02x03” > /sys/bus/usb/.../text (hex)
5. Set the decimal places. The device has either 6 or 8 decimal points. to set the nth decimal place calculate 10 ** n and echo it in to /sys/bus/usb/.../decimals To set multiple decimals points sum up each power. For example, to set the 0th and 3rd decimal place echo 1001 > /sys/bus/usb/.../decimals

CHAPTER

SEVENTEEN

MTOUCHUSB DRIVER

17.1 Changes

- 0.3 - Created based off of scanner & INSTALL from the original touchscreen driver on freecode (http://freecode.com/projects/3mtouchscreendriver)
- Amended for linux-2.4.18, then 2.4.19
- 0.5 - Complete rewrite using Linux Input in 2.6.3 Unfortunately no calibration support at this time
- 1.4 - Multiple changes to support the EXII 5000UC and house cleaning Changed reset from standard USB dev reset to vendor reset Changed data sent to host from compensated to raw coordinates Eliminated vendor/product module params Performed multiple successful tests with an EXII-5010UC

17.2 Supported Hardware

```
All controllers have the Vendor: 0x0596 & Product: 0x0001


Controller Description          Part Number
-----------------------------------------------------

USB Capacitive - Pearl Case     14-205  (Discontinued)
USB Capacitive - Black Case     14-124  (Discontinued)
USB Capacitive - No Case        14-206  (Discontinued)

USB Capacitive - Pearl Case     EXII-5010UC
USB Capacitive - Black Case     EXII-5030UC
USB Capacitive - No Case        EXII-5050UC
```

17.3 Driver Notes

Installation is simple, you only need to add Linux Input, Linux USB, and the driver to the kernel. The driver can also be optionally built as a module.

This driver appears to be one of possible 2 Linux USB Input Touchscreen drivers. Although 3M produces a binary only driver available for download, I persist in updating this driver since I would like to use the touchscreen for embedded apps using QTEmbedded, DirectFB, etc. So I feel the logical choice is to use Linux Input.

Currently there is no way to calibrate the device via this driver. Even if the device could be calibrated, the driver pulls to raw coordinate data from the controller. This means calibration must be performed within the userspace.

The controller screen resolution is now 0 to 16384 for both X and Y reporting the raw touch data. This is the same for the old and new capacitive USB controllers.

Perhaps at some point an abstract function will be placed into evdev so generic functions like calibrations, resets, and vendor information can be requested from the userspace (And the drivers would handle the vendor specific tasks).

17.4 TODO

Implement a control urb again to handle requests to and from the device such as calibration, etc once/if it becomes available.

17.5 Disclaimer

I am not a MicroTouch/3M employee, nor have I ever been. 3M does not support this driver! If you want touch drivers only supported within X, please go to:

http://www.3m.com/3MTouchSystems/

17.6 Thanks

A huge thank you to 3M Touch Systems for the EXII-5010UC controllers for testing!

CHAPTER

EIGHTEEN

OHCI

23-Aug-2002

The “ohci-hcd” driver is a USB Host Controller Driver (HCD) that is derived from the “usb-ohci” driver from the 2.4 kernel series. The “usb-ohci” code was written primarily by Roman Weissgaerber <weissg@vienna.at> but with contributions from many others (read its copyright/licencing header).

It supports the “Open Host Controller Interface” (OHCI), which standardizes hardware register protocols used to talk to USB 1.1 host controllers. As compared to the earlier “Universal Host Controller Interface” (UHCI) from Intel, it pushes more intelligence into the hardware. USB 1.1 controllers from vendors other than Intel and VIA generally use OHCI.

Changes since the 2.4 kernel include

- improved robustness; bugfixes; and less overhead
- supports the updated and simplified usbcore APIs
- interrupt transfers can be larger, and can be queued
- less code, by using the upper level “hcd” framework
- supports some non-PCI implementations of OHCI
- ... more

The “ohci-hcd” driver handles all USB 1.1 transfer types. Transfers of all types can be queued. That was also true in “usb-ohci”, except for interrupt transfers. Previously, using periods of one frame would risk data loss due to overhead in IRQ processing. When interrupt transfers are queued, those risks can be minimized by making sure the hardware always has transfers to work on while the OS is getting around to the relevant IRQ processing.

- David Brownell <dbrownell@users.sourceforge.net>

CHAPTER

NINETEEN

USB RAW GADGET

USB Raw Gadget is a gadget driver that gives userspace low-level control over the gadget's communication process.

Like any other gadget driver, Raw Gadget implements USB devices via the USB gadget API. Unlike most gadget drivers, Raw Gadget does not implement any concrete USB functions itself but requires userspace to do that.

Raw Gadget is currently a strictly debugging feature and should not be used in production. Use GadgetFS instead.

Enabled with CONFIG_USB_RAW_GADGET.

19.1 Comparison to GadgetFS

Raw Gadget is similar to GadgetFS but provides more direct access to the USB gadget layer for userspace. The key differences are:

1. Raw Gadget passes every USB request to userspace to get a response, while GadgetFS responds to some USB requests internally based on the provided descriptors. Note that the UDC driver might respond to some requests on its own and never forward them to the gadget layer.
2. Raw Gadget allows providing arbitrary data as responses to USB requests, while GadgetFS performs sanity checks on the provided USB descriptors. This makes Raw Gadget suitable for fuzzing by providing malformed data as responses to USB requests.
3. Raw Gadget provides a way to select a UDC device/driver to bind to, while GadgetFS currently binds to the first available UDC. This allows having multiple Raw Gadget instances bound to different UDCs.
4. Raw Gadget explicitly exposes information about endpoints addresses and capabilities. This allows the user to write UDC-agnostic gadgets.
5. Raw Gadget has an ioctl-based interface instead of a filesystem-based one.

19.2 Userspace interface

The user can interact with Raw Gadget by opening `/dev/raw-gadget` and issuing ioctl calls; see the comments in include/uapi/linux/usb/raw_gadget.h for details. Multiple Raw Gadget instances (bound to different UDCs) can be used at the same time.

A typical usage scenario of Raw Gadget:

1. Create a Raw Gadget instance by opening `/dev/raw-gadget`.
2. Initialize the instance via `USB_RAW_IOCTL_INIT`.
3. Launch the instance with `USB_RAW_IOCTL_RUN`.
4. In a loop issue `USB_RAW_IOCTL_EVENT_FETCH` to receive events from Raw Gadget and react to those depending on what kind of USB gadget must be implemented.

Note that some UDC drivers have fixed addresses assigned to endpoints, and therefore arbitrary endpoint addresses cannot be used in the descriptors. Nevertheless, Raw Gadget provides a UDC-agnostic way to write USB gadgets. Once `USB_RAW_EVENT_CONNECT` is received via `USB_RAW_IOCTL_EVENT_FETCH`, `USB_RAW_IOCTL_EPS_INFO` can be used to find out information about the endpoints that the UDC driver has. Based on that, userspace must choose UDC endpoints for the gadget and assign addresses in the endpoint descriptors correspondingly.

Raw Gadget usage examples and a test suite:

https://github.com/xairy/raw-gadget

19.3 Internal details

Every Raw Gadget endpoint read/write ioctl submits a USB request and waits until its completion. This is done deliberately to assist with coverage-guided fuzzing by having a single syscall fully process a single USB request. This feature must be kept in the implementation.

19.4 Potential future improvements

- Report more events (suspend, resume, etc.) through `USB_RAW_IOCTL_EVENT_FETCH`.
- Support `O_NONBLOCK` I/O. This would be another mode of operation, where Raw Gadget would not wait until the completion of each USB request.
- Support USB 3 features (accept SS endpoint companion descriptor when enabling endpoints; allow providing `stream_id` for bulk transfers).
- Support ISO transfer features (expose `frame_number` for completed requests).

CHAPTER

TWENTY

USB/IP PROTOCOL

20.1 Architecture

The USB/IP protocol follows a server/client architecture. The server exports the USB devices and the clients import them. The device driver for the exported USB device runs on the client machine.

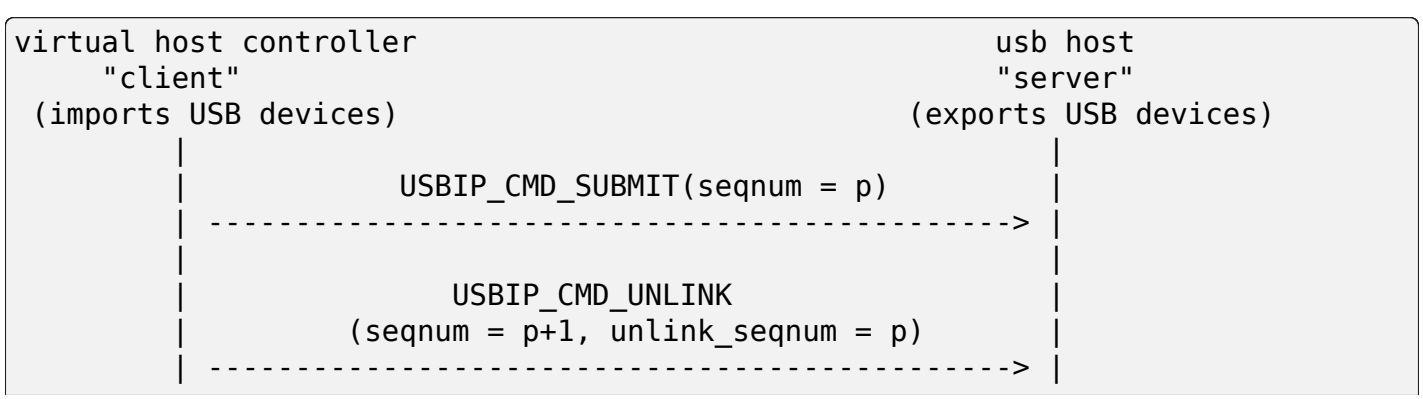
The client may ask for the list of the exported USB devices. To get the list the client opens a TCP/IP connection to the server, and sends an OP_REQ_DEVLIST packet on top of the TCP/IP connection (so the actual OP_REQ_DEVLIST may be sent in one or more pieces at the low level transport layer). The server sends back the OP_REP_DEVLIST packet which lists the exported USB devices. Finally the TCP/IP connection is closed.

```
virtual host controller                                  usb host
     "client"                                            "server"
 (imports USB devices)                                (exports USB devices)
         |                                                    |
         |                   OP_REQ_DEVLIST                   |
         | ------------------------------------------------>  |
         |                                                    |
         |                   OP_REP_DEVLIST                   |
         | <------------------------------------------------  |
         |                                                    |
```

Once the client knows the list of exported USB devices it may decide to use one of them. First the client opens a TCP/IP connection to the server and sends an OP_REQ_IMPORT packet. The server replies with OP_REP_IMPORT. If the import was successful the TCP/IP connection remains open and will be used to transfer the URB traffic between the client and the server. The client may send two types of packets: the USBIP_CMD_SUBMIT to submit an URB, and USBIP_CMD_UNLINK to unlink a previously submitted URB. The answers of the server may be USBIP_RET_SUBMIT and USBIP_RET_UNLINK respectively.

```
virtual host controller                                  usb host
     "client"                                            "server"
 (imports USB devices)                                (exports USB devices)
         |                                                    |
         |                   OP_REQ_IMPORT                    |
         | ------------------------------------------------>  |
         |                                                    |
         |                   OP_REP_IMPORT                    |
```

For UNLINK, note that after a successful USBIP_RET_UNLINK, the unlinked URB submission would not have a corresponding USBIP_RET_SUBMIT (this is explained in function stub_recv_cmd_unlink of drivers/usb/usbip/stub_rx.c).



```
        |                                                 |
        |               USBIP_RET_UNLINK                  |
        |        (seqnum = p+1, status = -ECONNRESET)     |
        | <---------------------------------------------- |
        |                                                 |
        |         Note: No USBIP_RET_SUBMIT(seqnum = p)   |
        | <--X---X---X---X---X---X---X---X---X---X---X--- |
        |                         .                       |
        |                         :                       |
        |                                                 |
        |            USBIP_CMD_SUBMIT(seqnum = q)         |
        | ----------------------------------------------> |
        |                                                 |
        |            USBIP_RET_SUBMIT(seqnum = q)         |
        | <---------------------------------------------- |
        |                                                 |
        |               USBIP_CMD_UNLINK                  |
        |         (seqnum = q+1, unlink_seqnum = q)       |
        | ----------------------------------------------> |
        |                                                 |
        |               USBIP_RET_UNLINK                  |
        |           (seqnum = q+1, status = 0)            |
        | <---------------------------------------------- |
        |                                                 |
```

The fields are in network (big endian) byte order meaning that the most significant byte (MSB) is stored at the lowest address.

20.2 Protocol Version

The documented USBIP version is v1.1.1. The binary representation of this version in message headers is 0x0111.

This is defined in tools/usb/usbip/configure.ac

20.3 Message Format

OP_REQ_DEVLIST:
Retrieve the list of exported USB devices.

Offset	Length	Value	Description
0	2		USBIP version
2	2	0x8005	Command code: Retrieve the list of exported USB devices.
4	4	0x00000000	Status: unused, shall be set to 0

OP_REP_DEVLIST:
Reply with the list of exported USB devices.

Offset	Length	Value	Description
0	2		USBIP version
2	2	0x0005	Reply code: The list of exported USB devices.
4	4	0x00000000	Status: 0 for OK
8	4	n	Number of exported devices: 0 means no exported devices.
0x0C			From now on the exported n devices are described, if any. If no devices are exported the message ends with the previous “number of exported devices” field.
	256		path: Path of the device on the host exporting the USB device, string closed with zero byte, e.g. “/sys/devices/pci0000:00/0000:00:1d.1/usb3/3-2” The unused bytes shall be filled with zero bytes.
0x10C	32		busid: Bus ID of the exported device, string closed with zero byte, e.g. “3-2”. The unused bytes shall be filled with zero bytes.
0x12C	4		busnum
0x130	4		devnum
0x134	4		speed
0x138	2		idVendor
0x13A	2		idProduct
0x13C	2		bcdDevice
0x13E	1		bDeviceClass
0x13F	1		bDeviceSubClass
0x140	1		bDeviceProtocol
0x141	1		bConfigurationValue
0x142	1		bNumConfigurations
0x143	1		bNumInterfaces
0x144		m_0	From now on each interface is described, all together bNumInterfaces times, with the following 4 fields:
	1		bInterfaceClass
0x145	1		bInterfaceSubClass
0x146	1		bInterfaceProtocol
0x147	1		padding byte for alignment, shall be set to zero
0xC + i*0x138 + m_(i-1)*4			The second exported USB device starts at i=1 with the path field.

OP_REQ_IMPORT:

Request to import (attach) a remote USB device.

Offset	Length	Value	Description
0	2		USBIP version
2	2	0x8003	Command code: import a remote USB device.
4	4	0x00000000	Status: unused, shall be set to 0
8	32		busid: the busid of the exported device on the remote host. The possible values are taken from the message field OP_REP_DEVLIST.busid. A string closed with zero, the unused bytes shall be filled with zeros.

OP_REP_IMPORT:

Reply to import (attach) a remote USB device.

Offset	Length	Value	Description
0	2		USBIP version
2	2	0x0003	Reply code: Reply to import.
4	4	0x00000000	Status: • 0 for OK • 1 for error
8			From now on comes the details of the imported device, if the previous status field was OK (0), otherwise the reply ends with the status field.
	256		path: Path of the device on the host exporting the USB device, string closed with zero byte, e.g. “/sys/devices/pci0000:00/0002” The unused bytes shall be filled with zero bytes.
0x108	32		busid: Bus ID of the exported device, string closed with zero byte, e.g. “3-2”. The unused bytes shall be filled with zero bytes.
0x128	4		busnum
0x12C	4		devnum
0x130	4		speed
0x134	2		idVendor
0x136	2		idProduct
0x138	2		bcdDevice
0x139	1		bDeviceClass
0x13A	1		bDeviceSubClass
0x13B	1		bDeviceProtocol
0x13C	1		bConfigurationValue
0x13D	1		bNumConfigurations
0x13E	1		bNumInterfaces

The following four commands have a common basic header called ‘usbip_header_basic’, and their headers, called ‘usbip_header’ (before transfer_buffer payload), have the same length, therefore paddings are needed.

usbip_header_basic:

Offset	Length	Description
0	4	command
4	4	seqnum: sequential number that identifies requests and corresponding responses; incremented per connection
8	4	devid: specifies a remote USB device uniquely instead of busnum and devnum; for client (request), this value is ((busnum << 16) | devnum); for server (response), this shall be set to 0
0xC	4	direction: • 0: USBIP_DIR_OUT • 1: USBIP_DIR_IN only used by client, for server this shall be 0
0x10	4	ep: endpoint number only used by client, for server this shall be 0; for UNLINK, this shall be 0

USBIP_CMD_SUBMIT:
Submit an URB

Offset	Length	Description
0	20	usbip_header_basic, 'command' shall be 0x00000001
0x14	4	transfer_flags: possible values depend on the USBIP_URB transfer_flags. Refer to include/uapi/linux/usbip.h and Documentation/driver-api/usb/URB.rst. Refer to usbip_pack_cmd_submit() and tweak_transfer_flags() in drivers/usb/usbip/ usbip_common.c.
0x18	4	transfer_buffer_length: use URB transfer_buffer_length
0x1C	4	start_frame: use URB start_frame; initial frame for ISO transfer; shall be set to 0 if not ISO transfer
0x20	4	number_of_packets: number of ISO packets; shall be set to 0xffffffff if not ISO transfer
0x24	4	interval: maximum time for the request on the server-side host controller
0x28	8	setup: data bytes for USB setup, filled with zeros if not used.
0x30	n	transfer_buffer. If direction is USBIP_DIR_OUT then n equals transfer_buffer_length; otherwise n equals 0. For ISO transfers the padding between each ISO packets is not transmitted.
0x30+n	m	iso_packet_descriptor

USBIP_RET_SUBMIT:
Reply for submitting an URB

Offset	Length	Description
0	20	usbip_header_basic, ‘command’ shall be 0x00000003
0x14	4	status: zero for successful URB transaction, otherwise some kind of error happened.
0x18	4	actual_length: number of URB data bytes; use URB actual_length
0x1C	4	start_frame: use URB start_frame; initial frame for ISO transfer; shall be set to 0 if not ISO transfer
0x20	4	number_of_packets: number of ISO packets; shall be set to 0xffffffff if not ISO transfer
0x24	4	error_count
0x28	8	padding, shall be set to 0
0x30	n	transfer_buffer. If direction is USBIP_DIR_IN then n equals actual_length; otherwise n equals 0. For ISO transfers the padding between each ISO packets is not transmitted.
0x30+n	m	iso_packet_descriptor

USBIP_CMD_UNLINK:
Unlink an URB

Offset	Length	Description
0	20	usbip_header_basic, ‘command’ shall be 0x00000002
0x14	4	unlink_seqnum, of the SUBMIT request to unlink
0x18	24	padding, shall be set to 0

USBIP_RET_UNLINK:
Reply for URB unlink

Offset	Length	Description
0	20	usbip_header_basic, ‘command’ shall be 0x00000004
0x14	4	status: This is similar to the status of USBIP_RET_SUBMIT (share the same memory offset). When UNLINK is successful, status is -ECONNRESET; when USBIP_CMD_UNLINK is after USBIP_RET_SUBMIT status is 0
0x18	24	padding, shall be set to 0

20.4 EXAMPLE

The following data is captured from wire with Human Interface Devices (HID) payload

```
CmdIntrIN:  00000001 00000d05 0001000f 00000001 00000001 00000200 00000040␣
↪ffffffff 00000000 00000004 00000000 00000000
CmdIntrOUT: 00000001 00000d06 0001000f 00000000 00000001 00000000 00000040␣
↪ffffffff 00000000 00000004 00000000 00000000

↪ffffffff␣860008a784ce5ae21237630000000000000000000000000000000000000000000000000
RetIntrOut: 00000003 00000d06 00000000 00000000 00000000 00000000 00000040␣
↪ffffffff 00000000 00000000 00000000 00000000
```

```
RetIntrIn:  00000003 00000d05 00000000 00000000 00000000 00000000 00000040␣
↪ffffffff 00000000 00000000 00000000 00000000

↪ffffffff␣860011a784ce5ae2123763612891b10201000004000000000000000000000000000000000000
```

CHAPTER

TWENTYONE

USBMON

21.1 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.

Two APIs are currently implemented: “text” and “binary”. The binary API is available through a character device in /dev namespace and is an ABI. The text API is deprecated since 2.6.35, but available for convenience.

21.2 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.

21.2.1 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.

21.2.2 2. Find which bus connects to the desired device

Run "cat /sys/kernel/debug/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 the two /sys/kernel/debug/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. Alternatively, you can look at the output from "lsusb" and get the bus number from the appropriate line. Example:

Bus 003 Device 002: ID 0557:2004 ATEN UC100KM V2.00

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

21.2.4 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.

21.2.5 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.

21.3 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 reported in the Data Length word. In the case of an Isochronous input (Zi) completion where the received data is sparse in the buffer, the length of the collected data can be greater than the Data Length value (because Data Length counts only the bytes that were received whereas the Data words contain the entire transfer buffer).

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 0x28 (READ_10) in a 31-byte Bulk wrapper to a storage device at address 5:

```
dd65f0e8 4128379752 S Bo:1:005:2 -115 31 = 55534243 ad000000 00800000 80010a28␣
↪20000000 20000040 00000000 000000
dd65f0e8 4128379808 C Bo:1:005:2 0 31 >
```

21.4 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, unprivileged 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.

CHAPTER

TWENTYTWO

USB SERIAL

22.1 Introduction

The USB serial driver currently supports a number of different USB to serial converter products, as well as some devices that use a serial interface from userspace to talk to the device.

See the individual product section below for specific information about the different devices.

22.2 Configuration

Currently the driver can handle up to 256 different serial interfaces at one time.

The major number that the driver uses is 188 so to use the driver, create the following nodes:

```
mknod /dev/ttyUSB0 c 188 0
mknod /dev/ttyUSB1 c 188 1
mknod /dev/ttyUSB2 c 188 2
mknod /dev/ttyUSB3 c 188 3
        .
        .
        .
mknod /dev/ttyUSB254 c 188 254
mknod /dev/ttyUSB255 c 188 255
```

When the device is connected and recognized by the driver, the driver will print to the system log, which node(s) the device has been bound to.

22.3 Specific Devices Supported

22.3.1 ConnectTech WhiteHEAT 4 port converter

ConnectTech has been very forthcoming with information about their device, including providing a unit to test with.

The driver is officially supported by Connect Tech Inc. http://www.connecttech.com

For any questions or problems with this driver, please contact Connect Tech's Support Department at support@connecttech.com

22.3.2 HandSpring Visor, Palm USB, and Clié USB driver

This driver works with all HandSpring USB, Palm USB, and Sony Clié USB devices.

Only when the device tries to connect to the host, will the device show up to the host as a valid USB device. When this happens, the device is properly enumerated, assigned a port, and then communication _should_ be possible. The driver cleans up properly when the device is removed, or the connection is canceled on the device.

NOTE:
This means that in order to talk to the device, the sync button must be pressed BEFORE trying to get any program to communicate to the device. This goes against the current documentation for pilot-xfer and other packages, but is the only way that it will work due to the hardware in the device.

When the device is connected, try talking to it on the second port (this is usually /dev/ttyUSB1 if you do not have any other usb-serial devices in the system.) The system log should tell you which port is the port to use for the HotSync transfer. The "Generic" port can be used for other device communication, such as a PPP link.

For some Sony Clié devices, /dev/ttyUSB0 must be used to talk to the device. This is true for all OS version 3.5 devices, and most devices that have had a flash upgrade to a newer version of the OS. See the kernel system log for information on which is the correct port to use.

If after pressing the sync button, nothing shows up in the system log, try resetting the device, first a hot reset, and then a cold reset if necessary. Some devices need this before they can talk to the USB port properly.

Devices that are not compiled into the kernel can be specified with module parameters. e.g. modprobe visor vendor=0x54c product=0x66

There is a webpage and mailing lists for this portion of the driver at: http://sourceforge.net/projects/usbvisor/

For any questions or problems with this driver, please contact Greg Kroah-Hartman at greg@kroah.com

22.3.3 PocketPC PDA Driver

This driver can be used to connect to Compaq iPAQ, HP Jornada, Casio EM500 and other PDAs running Windows CE 3.0 or PocketPC 2002 using a USB cable/cradle. Most devices supported by ActiveSync are supported out of the box. For others, please use module parameters to specify the product and vendor id. e.g. modprobe ipaq vendor=0x3f0 product=0x1125

The driver presents a serial interface (usually on /dev/ttyUSB0) over which one may run ppp and establish a TCP/IP link to the PDA. Once this is done, you can transfer files, backup, download email etc. The most significant advantage of using USB is speed - I can get 73 to 113 kbytes/sec for download/upload to my iPAQ.

This driver is only one of a set of components required to utilize the USB connection. Please visit http://synce.sourceforge.net which contains the necessary packages and a simple step-by-step howto.

Once connected, you can use Win CE programs like ftpView, Pocket Outlook from the PDA and xcerdisp, synce utilities from the Linux side.

To use Pocket IE, follow the instructions given at http://www.tekguru.co.uk/EM500/usbtonet.htm to achieve the same thing on Win98. Omit the proxy server part; Linux is quite capable of forwarding packets unlike Win98. Another modification is required at least for the iPAQ - disable autosync by going to the Start/Settings/Connections menu and unchecking the “Automatically synchronize ...” box. Go to Start/Programs/Connections, connect the cable and select “usbdial” (or whatever you named your new USB connection). You should finally wind up with a “Connected to usbdial” window with status shown as connected. Now start up PIE and browse away.

If it doesn’t work for some reason, load both the usbserial and ipaq module with the module parameter “debug” set to 1 and examine the system log. You can also try soft-resetting your PDA before attempting a connection.

Other functionality may be possible depending on your PDA. According to Wes Cilldhaire <billybobjoehenrybob@hotmail.com>, with the Toshiba E570, ...if you boot into the bootloader (hold down the power when hitting the reset button, continuing to hold onto the power until the bootloader screen is displayed), then put it in the cradle with the ipaq driver loaded, open a terminal on /dev/ttyUSB0, it gives you a “USB Reflash” terminal, which can be used to flash the ROM, as well as the microP code.. so much for needing Toshiba’s $350 serial cable for flashing!! :D NOTE: This has NOT been tested. Use at your own risk.

For any questions or problems with the driver, please contact Ganesh Varadarajan <ganesh@veritas.com>

22.3.4 Keyspan PDA Serial Adapter

Single port DB-9 serial adapter, pushed as a PDA adapter for iMacs (mostly sold in Macintosh catalogs, comes in a translucent white/green dongle). Fairly simple device. Firmware is homebrew. This driver also works for the Xircom/Entrega single port serial adapter.

Current status:

Things that work:

- basic input/output (tested with 'cu')
- blocking write when serial line can't keep up
- changing baud rates (up to 115200)
- getting/setting modem control pins (TIOCM{GET,SET,BIS,BIC})
- sending break (although duration looks suspect)

Things that don't:

- device strings (as logged by kernel) have trailing binary garbage
- device ID isn't right, might collide with other Keyspan products
- changing baud rates ought to flush tx/rx to avoid mangled half characters

Big Things on the todo list:

- parity, 7 vs 8 bits per char, 1 or 2 stop bits
- HW flow control
- not all of the standard USB descriptors are handled: Get_Status, Set_Feature, O_NONBLOCK, select()

For any questions or problems with this driver, please contact Brian Warner at warner@lothar.com

22.3.5 Keyspan USA-series Serial Adapters

Single, Dual and Quad port adapters - driver uses Keyspan supplied firmware and is being developed with their support.

Current status:

The USA-18X, USA-28X, USA-19, USA-19W and USA-49W are supported and have been pretty thoroughly tested at various baud rates with 8-N-1 character settings. Other character lengths and parity setups are presently untested.

The USA-28 isn't yet supported though doing so should be pretty straightforward. Contact the maintainer if you require this functionality.

More information is available at:

http://www.carnationsoftware.com/carnation/Keyspan.html

For any questions or problems with this driver, please contact Hugh Blemings at hugh@misc.nu

22.3.6 FTDI Single Port Serial Driver

This is a single port DB-25 serial adapter.

Devices supported include:

- TripNav TN-200 USB GPS
- Navis Engineering Bureau CH-4711 USB GPS

For any questions or problems with this driver, please contact Bill Ryder.

22.3.7 ZyXEL omni.net lcd plus ISDN TA

This is an ISDN TA. Please report both successes and troubles to azummo@towertech.it

22.3.8 Cypress M8 CY4601 Family Serial Driver

This driver was in most part developed by Neil “koyama” Whelchel. It has been improved since that previous form to support dynamic serial line settings and improved line handling. The driver is for the most part stable and has been tested on an smp machine. (dual p2)

Chipsets supported under CY4601 family:

CY7C63723, CY7C63742, CY7C63743, CY7C64013

Devices supported:

- DeLorme’s USB Earthmate GPS (SiRF Star II lp arch)
- Cypress HID->COM RS232 adapter

Note:
Cypress Semiconductor claims no affiliation with the hid->com device.

Most devices using chipsets under the CY4601 family should work with the driver. As long as they stay true to the CY4601 usbserial specification.

Technical notes:

The Earthmate starts out at 4800 8N1 by default... the driver will upon start init to this setting. usbserial core provides the rest of the termios settings, along with some custom termios so that the output is in proper format and parsable.

The device can be put into sirf mode by issuing NMEA command:

```
$PSRF100,<protocol>,<baud>,<databits>,<stopbits>,<parity>
 ↪*CHECKSUM
$PSRF100,0,9600,8,1,0*0C

It should then be sufficient to change the port termios␣
 ↪to match this
to begin communicating.
```

As far as I can tell it supports pretty much every sirf command as documented online available with firmware 2.31, with some unknown message ids.

The hid->com adapter can run at a maximum baud of 115200bps. Please note that the device has trouble or is incapable of raising line voltage properly. It will be fine with null modem links, as long as you do not try to link two together without hacking the adapter to set the line high.

The driver is smp safe. Performance with the driver is rather low when using it for transferring files. This is being worked on, but I would be willing to accept patches. An urb queue or packet buffer would likely fit the bill here.

If you have any questions, problems, patches, feature requests, etc. you can contact me here via email:

dignome@gmail.com

(your problems/patches can alternately be submitted to usb-devel)

22.3.9 Digi AccelePort Driver

This driver supports the Digi AccelePort USB 2 and 4 devices, 2 port (plus a parallel port) and 4 port USB serial converters. The driver does NOT yet support the Digi AccelePort USB 8.

This driver works under SMP with the usb-uhci driver. It does not work under SMP with the uhci driver.

The driver is generally working, though we still have a few more ioctls to implement and final testing and debugging to do. The parallel port on the USB 2 is supported as a serial to parallel converter; in other words, it appears as another USB serial port on Linux, even though physically it is really a parallel port. The Digi Acceleport USB 8 is not yet supported.

Please contact Peter Berger (pberger@brimson.com) or Al Borchers (al-borchers@steinerpoint.com) for questions or problems with this driver.

22.3.10 Belkin USB Serial Adapter F5U103

Single port DB-9/PS-2 serial adapter from Belkin with firmware by eTEK Labs. The Peracom single port serial adapter also works with this driver, as well as the GoHubs adapter.

Current status:

The following have been tested and work:

- Baud rate 300-230400
- Data bits 5-8
- Stop bits 1-2
- Parity N,E,O,M,S
- Handshake None, Software (XON/XOFF), Hardware (CTSRTS,CTSDTR)[1]
- Break Set and clear
- Line control Input/Output query and control[2]

TO DO List:

- Add true modem control line query capability. Currently tracks the states reported by the interrupt and the states requested.
- Add error reporting back to application for UART error conditions.
- Add support for flush ioctls.
- Add everything else that is missing :)

For any questions or problems with this driver, please contact William Greathouse at wgreathouse@smva.com

22.3.11 Empeg empeg-car Mark I/II Driver

This is an experimental driver to provide connectivity support for the client synchronization tools for an Empeg empeg-car mp3 player.

Tips:

- Don't forget to create the device nodes for ttyUSB{0,1,2,...}
- modprobe empeg (modprobe is your friend)
- emptool --usb /dev/ttyUSB0 (or whatever you named your device node)

For any questions or problems with this driver, please contact Gary Brubaker at xavyer@ix.netcom.com

[1] Hardware input flow control is only enabled for firmware levels above 2.06. Read source code comments describing Belkin firmware errata. Hardware output flow control is working for all firmware versions.

[2] Queries of inputs (CTS,DSR,CD,RI) show the last reported state. Queries of outputs (DTR,RTS) show the last requested state and may not reflect current state as set by automatic hardware flow control.

22.3.12 MCT USB Single Port Serial Adapter U232

This driver is for the MCT USB-RS232 Converter (25 pin, Model No. U232-P25) from Magic Control Technology Corp. (there is also a 9 pin Model No. U232-P9). More information about this device can be found at the manufacturer's web-site: http://www.mct.com.tw.

The driver is generally working, though it still needs some more testing. It is derived from the Belkin USB Serial Adapter F5U103 driver and its TODO list is valid for this driver as well.

This driver has also been found to work for other products, which have the same Vendor ID but different Product IDs. Sitecom's U232-P25 serial converter uses Product ID 0x230 and Vendor ID 0x711 and works with this driver. Also, D-Link's DU-H3SP USB BAY also works with this driver.

For any questions or problems with this driver, please contact Wolfgang Grandegger at wolfgang@ces.ch

22.3.13 Inside Out Networks Edgeport Driver

This driver supports all devices made by Inside Out Networks, specifically the following models:

- Edgeport/4
- Rapidport/4
- Edgeport/4t
- Edgeport/2
- Edgeport/4i
- Edgeport/2i
- Edgeport/421
- Edgeport/21
- Edgeport/8
- Edgeport/8 Dual
- Edgeport/2D8
- Edgeport/4D8
- Edgeport/8i
- Edgeport/2 DIN
- Edgeport/4 DIN
- Edgeport/16 Dual

For any questions or problems with this driver, please contact Greg Kroah-Hartman at greg@kroah.com

22.3.14 REINER SCT cyberJack pinpad/e-com USB chipcard reader

Interface to ISO 7816 compatible contactbased chipcards, e.g. GSM SIMs.

Current status:

> This is the kernel part of the driver for this USB card reader. There is also a user part for a CT-API driver available. A site for downloading is TBA. For now, you can request it from the maintainer (linux-usb@sii.li).

For any questions or problems with this driver, please contact linux-usb@sii.li

22.3.15 Prolific PL2303 Driver

This driver supports any device that has the PL2303 chip from Prolific in it. This includes a number of single port USB to serial converters, more than 70% of USB GPS devices (in 2010), and some USB UPSes. Devices from Aten (the UC-232) and IO-Data work with this driver, as does the DCU-11 mobile-phone cable.

For any questions or problems with this driver, please contact Greg Kroah-Hartman at greg@kroah.com

22.3.16 KL5KUSB105 chipset / PalmConnect USB single-port adapter

Current status:

> The driver was put together by looking at the usb bus transactions done by Palm's driver under Windows, so a lot of functionality is still missing. Notably, serial ioctls are sometimes faked or not yet implemented. Support for finding out about DSR and CTS line status is however implemented (though not nicely), so your favorite autopilot(1) and pilot-manager -daemon calls will work. Baud rates up to 115200 are supported, but handshaking (software or hardware) is not, which is why it is wise to cut down on the rate used is wise for large transfers until this is settled.
>
> See http://www.uuhaus.de/linux/palmconnect.html for up-to-date information on this driver.

22.3.17 Winchiphead CH341 Driver

This driver is for the Winchiphead CH341 USB-RS232 Converter. This chip also implements an IEEE 1284 parallel port, I2C and SPI, but that is not supported by the driver. The protocol was analyzed from the behaviour of the Windows driver, no datasheet is available at present.

The manufacturer's website: http://www.winchiphead.com/.

For any questions or problems with this driver, please contact frank@kingswood-consulting.co.uk.

22.3.18 Moschip MCS7720, MCS7715 driver

These chips are present in devices sold by various manufacturers, such as Syba and Cables Unlimited. There may be others. The 7720 provides two serial ports, and the 7715 provides one serial and one standard PC parallel port. Support for the 7715's parallel port is enabled by a separate option, which will not appear unless parallel port support is first enabled at the top-level of the Device Drivers config menu. Currently only compatibility mode is supported on the parallel port (no ECP/EPP).

TODO:

- Implement ECP/EPP modes for the parallel port.
- Baud rates higher than 115200 are currently broken.
- Devices with a single serial port based on the Moschip MCS7703 may work with this driver with a simple addition to the usb_device_id table. I don't have one of these devices, so I can't say for sure.

22.3.19 Generic Serial driver

If your device is not one of the above listed devices, compatible with the above models, you can try out the "generic" interface. This interface does not provide any type of control messages sent to the device, and does not support any kind of device flow control. All that is required of your device is that it has at least one bulk in endpoint, or one bulk out endpoint.

To enable the generic driver to recognize your device, provide:

```
echo <vid> <pid> >/sys/bus/usb-serial/drivers/generic/new_id
```

where the <vid> and <pid> is replaced with the hex representation of your device's vendor id and product id. If the driver is compiled as a module you can also provide one id when loading the module:

```
insmod usbserial vendor=0x#### product=0x####
```

This driver has been successfully used to connect to the NetChip USB development board, providing a way to develop USB firmware without having to write a custom driver.

For any questions or problems with this driver, please contact Greg Kroah-Hartman at greg@kroah.com

22.4 Contact

If anyone has any problems using these drivers, with any of the above specified products, please contact the specific driver's author listed above, or join the Linux-USB mailing list (information on joining the mailing list, as well as a link to its searchable archive is at http://www.linux-usb.org/)

Greg Kroah-Hartman greg@kroah.com

CHAPTER

TWENTYTHREE

USB REFERENCES

2008-Mar-7

For USB help other than the readme files that are located in *Documentation/usb/**, see the following:

- Linux-USB project: http://www.linux-usb.org mirrors at http://usb.in.tum.de/linux-usb/ and http://it.linux-usb.org
- Linux USB Guide: http://linux-usb.sourceforge.net
- Linux-USB device overview (working devices and drivers): http://www.qbik.ch/usb/devices/

The Linux-USB mailing list is at linux-usb@vger.kernel.org

CHAPTER

TWENTYFOUR

LINUX CDC ACM INF

```
; Windows USB CDC ACM Setup File

; Based on INF template which was:
;     Copyright (c) 2000 Microsoft Corporation
;     Copyright (c) 2007 Microchip Technology Inc.
; likely to be covered by the MLPL as found at:
;    <http://msdn.microsoft.com/en-us/cc300389.aspx#MLPL>.
; For use only on Windows operating systems.

[Version]
Signature="$Windows NT$"
Class=Ports
ClassGuid={4D36E978-E325-11CE-BFC1-08002BE10318}
Provider=%Linux%
DriverVer=11/15/2007,5.1.2600.0

[Manufacturer]
%Linux%=DeviceList, NTamd64

[DestinationDirs]
DefaultDestDir=12


;------------------------------------------------------------------------------
;  Windows 2000/XP/Vista-32bit Sections
;------------------------------------------------------------------------------

[DriverInstall.nt]
include=mdmcpq.inf
CopyFiles=DriverCopyFiles.nt
AddReg=DriverInstall.nt.AddReg

[DriverCopyFiles.nt]
usbser.sys,,,0x20

[DriverInstall.nt.AddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,USBSER.sys
HKR,,EnumPropPages32,,"MsPorts.dll,SerialPortPropPageProvider"
```

```
[DriverInstall.nt.Services]
AddService=usbser, 0x00000002, DriverService.nt

[DriverService.nt]
DisplayName=%SERVICE%
ServiceType=1
StartType=3
ErrorControl=1
ServiceBinary=%12%\USBSER.sys

;------------------------------------------------------------------------------
;  Vista-64bit Sections
;------------------------------------------------------------------------------

[DriverInstall.NTamd64]
include=mdmcpq.inf
CopyFiles=DriverCopyFiles.NTamd64
AddReg=DriverInstall.NTamd64.AddReg

[DriverCopyFiles.NTamd64]
USBSER.sys,,,0x20

[DriverInstall.NTamd64.AddReg]
HKR,,DevLoader,,*ntkern
HKR,,NTMPDriver,,USBSER.sys
HKR,,EnumPropPages32,,"MsPorts.dll,SerialPortPropPageProvider"

[DriverInstall.NTamd64.Services]
AddService=usbser, 0x00000002, DriverService.NTamd64

[DriverService.NTamd64]
DisplayName=%SERVICE%
ServiceType=1
StartType=3
ErrorControl=1
ServiceBinary=%12%\USBSER.sys


;------------------------------------------------------------------------------
;  Vendor and Product ID Definitions
;------------------------------------------------------------------------------
; When developing your USB device, the VID and PID used in the PC side
; application program and the firmware on the microcontroller must match.
; Modify the below line to use your VID and PID.  Use the format as shown
; below.
; Note: One INF file can be used for multiple devices with different
;       VID and PIDs.  For each supported device, append
;       ",USB\VID_xxxx&PID_yyyy" to the end of the line.
;------------------------------------------------------------------------------
```

```
[SourceDisksFiles]
[SourceDisksNames]
[DeviceList]
%DESCRIPTION%=DriverInstall, USB\VID_0525&PID_A4A7, USB\VID_1D6B&PID_0104&MI_
 ↪02, USB\VID_1D6B&PID_0106&MI_00

[DeviceList.NTamd64]
%DESCRIPTION%=DriverInstall, USB\VID_0525&PID_A4A7, USB\VID_1D6B&PID_0104&MI_
 ↪02, USB\VID_1D6B&PID_0106&MI_00


;------------------------------------------------------------------------------
;  String Definitions
;------------------------------------------------------------------------------
;Modify these strings to customize your device
;------------------------------------------------------------------------------
[Strings]
Linux               = "Linux Developer Community"
DESCRIPTION         = "Gadget Serial"
SERVICE             = "USB RS-232 Emulation Driver"
```

CHAPTER

TWENTYFIVE

LINUX INF

```
; Based on template INF file found at
;    <https://msdn.microsoft.com/en-us/library/ff570620.aspx>
; which was:
;    Copyright (c) Microsoft Corporation
; and released under the MLPL as found at:
;    <http://msdn.microsoft.com/en-us/cc300389.aspx#MLPL>.
; For use only on Windows operating systems.

[Version]
Signature           = "$Windows NT$"
Class               = Net
ClassGUID           = {4d36e972-e325-11ce-bfc1-08002be10318}
Provider            = %Linux%
DriverVer           = 06/21/2006,6.0.6000.16384

[Manufacturer]
%Linux%             = LinuxDevices,NTx86,NTamd64,NTia64

; Decoration for x86 architecture
[LinuxDevices.NTx86]
%LinuxDevice%       = RNDIS.NT.5.1, USB\VID_0525&PID_a4a2, USB\VID_1d6b&PID_
 ↪0104&MI_00

; Decoration for x64 architecture
[LinuxDevices.NTamd64]
%LinuxDevice%       = RNDIS.NT.5.1, USB\VID_0525&PID_a4a2, USB\VID_1d6b&PID_
 ↪0104&MI_00

; Decoration for ia64 architecture
[LinuxDevices.NTia64]
%LinuxDevice%       = RNDIS.NT.5.1, USB\VID_0525&PID_a4a2, USB\VID_1d6b&PID_
 ↪0104&MI_00

;@@@ This is the common setting for setup
[ControlFlags]
ExcludeFromSelect=*

; DDInstall section
; References the in-build Netrndis.inf
```

```
[RNDIS.NT.5.1]
Characteristics     = 0x84   ; NCF_PHYSICAL + NCF_HAS_UI
BusType             = 15
; NEVER REMOVE THE FOLLOWING REFERENCE FOR NETRNDIS.INF
include             = netrndis.inf
needs               = Usb_Rndis.ndi
AddReg              = Rndis_AddReg_Vista

; DDInstal.Services section
[RNDIS.NT.5.1.Services]
include             = netrndis.inf
needs               = Usb_Rndis.ndi.Services

; Optional registry settings. You can modify as needed.
[RNDIS_AddReg_Vista]
HKR, NDI\params\VistaProperty, ParamDesc,  0, %Vista_Property%
HKR, NDI\params\VistaProperty, type,       0, "edit"
HKR, NDI\params\VistaProperty, LimitText,  0, "12"
HKR, NDI\params\VistaProperty, UpperCase,  0, "1"
HKR, NDI\params\VistaProperty, default,    0, " "
HKR, NDI\params\VistaProperty, optional,   0, "1"

; No sys copyfiles - the sys files are already in-build
; (part of the operating system).
; We do not support XP SP1-, 2003 SP1-, ME, 9x.

[Strings]
Linux                 = "Linux Developer Community"
LinuxDevice           = "Linux USB Ethernet/RNDIS Gadget"
Vista_Property        = "Optional Vista Property"
```

CHAPTER

TWENTYSIX

USB DEVFS DROP PERMISSIONS SOURCE

```
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <inttypes.h>
#include <unistd.h>

#include <linux/usbdevice_fs.h>

/* For building without an updated set of headers */
#ifndef USBDEVFS_DROP_PRIVILEGES
#define USBDEVFS_DROP_PRIVILEGES                _IOW('U', 30, __u32)
#define USBDEVFS_CAP_DROP_PRIVILEGES                0x40
#endif

void drop_privileges(int fd, uint32_t mask)
{
        int res;

        res = ioctl(fd, USBDEVFS_DROP_PRIVILEGES, &mask);
        if (res)
                printf("ERROR: USBDEVFS_DROP_PRIVILEGES returned %d\n", res);
        else
                printf("OK: privileges dropped!\n");
}

void reset_device(int fd)
{
        int res;

        res = ioctl(fd, USBDEVFS_RESET);
        if (!res)
                printf("OK: USBDEVFS_RESET succeeded\n");
        else
                printf("ERROR: reset failed! (%d - %s)\n",
                       -res, strerror(-res));
```

```
}

void claim_some_intf(int fd)
{
        int i, res;

        for (i = 0; i < 4; i++) {
                res = ioctl(fd, USBDEVFS_CLAIMINTERFACE, &i);
                if (!res)
                        printf("OK: claimed if %d\n", i);
                else
                        printf("ERROR claiming if %d (%d - %s)\n",
                               i, -res, strerror(-res));
        }
}

int main(int argc, char *argv[])
{
        uint32_t mask, caps;
        int c, fd;

        fd = open(argv[1], O_RDWR);
        if (fd < 0) {
                printf("Failed to open file\n");
                goto err_fd;
        }

        /*
         * check if dropping privileges is supported,
         * bail on systems where the capability is not present
         */
        ioctl(fd, USBDEVFS_GET_CAPABILITIES, &caps);
        if (!(caps & USBDEVFS_CAP_DROP_PRIVILEGES)) {
                printf("DROP_PRIVILEGES not supported\n");
                goto err;
        }

        /*
         * Drop privileges but keep the ability to claim all
         * free interfaces (i.e., those not used by kernel drivers)
         */
        drop_privileges(fd, -1U);

        printf("Available options:\n"
                "[0] Exit now\n"
                "[1] Reset device. Should fail if device is in use\n"
                "[2] Claim 4 interfaces. Should succeed where not in use\n"
                "[3] Narrow interface permission mask\n"
                "Which option shall I run?: ");
```

```
        while (scanf("%d", &c) == 1) {
                switch (c) {
                case 0:
                        goto exit;
                case 1:
                        reset_device(fd);
                        break;
                case 2:
                        claim_some_intf(fd);
                        break;
                case 3:
                        printf("Insert new mask: ");
                        scanf("%x", &mask);
                        drop_privileges(fd, mask);
                        break;
                default:
                        printf("I don't recognize that\n");
                }

                printf("Which test shall I run next?: ");
        }

exit:
        close(fd);
        return 0;

err:
        close(fd);
err_fd:
        return 1;
}
```

CHAPTER

TWENTYSEVEN

CREDITS

```
Credits for the Simple Linux USB Driver:

The following people have contributed to this code (in alphabetical
order by last name).  I'm sure this list should be longer, it's
difficult to maintain, add yourself with a patch if desired.

  Georg Acher <acher@informatik.tu-muenchen.de>
  David Brownell <dbrownell@users.sourceforge.net>
  Alan Cox <alan@lxorguk.ukuu.org.uk>
  Randy Dunlap <randy.dunlap@intel.com>
  Johannes Erdfelt <johannes@erdfelt.com>
  Deti Fliegl <deti@fliegl.de>
  ham <ham@unsuave.com>
  Bradley M Keryan <keryan@andrew.cmu.edu>
  Greg Kroah-Hartman <greg@kroah.com>
  Pavel Machek <pavel@suse.cz>
  Paul Mackerras <paulus@cs.anu.edu.au>
  Petko Manlolov <petkan@dce.bg>
  David E. Nelson <dnelson@jump.net>
  Vojtech Pavlik <vojtech@suse.cz>
  Bill Ryder <bryder@sgi.com>
  Thomas Sailer <sailer@ife.ee.ethz.ch>
  Gregory P. Smith <greg@electricrain.com>
  Linus Torvalds <torvalds@linux-foundation.org>
  Roman Weissgaerber <weissg@vienna.at>
  <Kazuki.Yasumatsu@fujixerox.co.jp>

Special thanks to:

  Inaky Perez Gonzalez <inaky@peloncho.fis.ucm.es> for starting the
  Linux USB driver effort and writing much of the larger uusbd driver.
  Much has been learned from that effort.

  The NetBSD & FreeBSD USB developers.  For being on the Linux USB list
  and offering suggestions and sharing implementation experiences.

Additional thanks to the following companies and people for donations
of hardware, support, time and development (this is from the original
THANKS file in Inaky's driver):
```

The following corporations have helped us in the development of Linux USB / UUSBD:

- 3Com GmbH for donating a ISDN Pro TA and supporting me in technical questions and with test equipment. I'd never expect such a great help.

- USAR Systems provided us with one of their excellent USB Evaluation Kits. It allows us to test the Linux-USB driver for compliance with the latest USB specification. USAR Systems recognized the importance of an up-to-date open Operating System and supports this project with Hardware. Thanks!.

- Thanks to Intel Corporation for their precious help.

- We teamed up with Cherry to make Linux the first OS with built-in USB support. Cherry is one of the biggest keyboard makers in the world.

- CMD Technology, Inc. sponsored us kindly donating a CSA-6700 PCI-to-USB Controller Board to test the OHCI implementation.

- Due to their support to us, Keytronic can be sure that they will sell keyboards to some of the 3 million (at least) Linux users.

- Many thanks to ing büro h doran [http://www.ibhdoran.com]! It was almost impossible to get a PC backplate USB connector for the motherboard here at Europe (mine, home-made, was quite lousy :). Now I know where to acquire nice USB stuff!

- Genius Germany donated a USB mouse to test the mouse boot protocol. They've also donated a F-23 digital joystick and a NetMouse Pro. Thanks!

- AVM GmbH Berlin is supporting the development of the Linux USB driver for the AVM ISDN Controller B1 USB. AVM is a leading manufacturer for active and passive ISDN Controllers and CAPI 2.0-based software. The active design of the AVM B1 is open for all OS platforms, including Linux.

- Thanks to Y-E Data, Inc. for donating their FlashBuster-U USB Floppy Disk Drive, so we could test the bulk transfer code.

- Many thanks to Logitech for contributing a three axis USB mouse.

  Logitech designs, manufactures and markets

Human Interface Devices, having a long history and
experience in making devices such as keyboards, mice,
trackballs, cameras, loudspeakers and control devices for
gaming and professional use.

Being a recognized vendor and seller for all these devices,
they have donated USB mice, a joystick and a scanner, as a
way to acknowledge the importance of Linux and to allow
Logitech customers to enjoy support in their favorite
operating systems and all Linux users to use Logitech and
other USB hardware.

Logitech is official sponsor of the Linux Conference on
Feb. 11th 1999 in Vienna, where we'll will present the
current state of the Linux USB effort.

- CATC has provided means to uncover dark corners of the UHCI
  inner workings with a USB Inspector.

- Thanks to Entrega for providing PCI to USB cards, hubs and
  converter products for development.

- Thanks to ConnectTech for providing a WhiteHEAT usb to
  serial converter, and the documentation for the device to
  allow a driver to be written.

- Thanks to ADMtek for providing Pegasus and Pegasus II
  evaluation boards, specs and valuable advices during
  the driver development.

And thanks go to (hey! in no particular order :)

- Oren Tirosh <orenti@hishome.net>, for standing so patiently
  all my doubts'bout USB and giving lots of cool ideas.

- Jochen Karrer <karrer@wpfd25.physik.uni-wuerzburg.de>, for
  pointing out mortal bugs and giving advice.

- Edmund Humemberger <ed@atnet.at>, for his great work on
  public relationships and general management stuff for the
  Linux-USB effort.

- Alberto Menegazzi <flash@flash.iol.it> is starting the
  documentation for the UUSBD. Go for it!

- Ric Klaren <ia_ric@cs.utwente.nl> for doing nice
  introductory documents (competing with Alberto's :).

- Christian Groessler <cpg@aladdin.de>, for his help on those
  itchy bits ... :)

- Paul MacKerras for polishing OHCI and pushing me harder for the iMac support, giving improvements and enhancements.

- Fernando Herrera <fherrera@eurielec.etsit.upm.es> has taken charge of composing, maintaining and feeding the long-awaited, unique and marvelous UUSBD FAQ! Tadaaaa!!!

- Rasca Gmelch <thron@gmx.de> has revived the raw driver and pointed bugs, as well as started the uusbd-utils package.

- Peter Dettori <dettori@ozy.dec.com> is uncovering bugs like crazy, as well as making cool suggestions, great :)

- All the Free Software and Linux community, the FSF & the GNU project, the MIT X consortium, the TeX people ... everyone! You know who you are!

- Big thanks to Richard Stallman for creating Emacs!

- The people at the linux-usb mailing list, for reading so many messages :) Ok, no more kidding; for all your advises!

- All the people at the USB Implementors Forum for their help and assistance.

- Nathan Myers <ncm@cantrip.org>, for his advice! (hope you liked Cibeles' party).

- Linus Torvalds, for starting, developing and managing Linux.

- Mike Smith, Craig Keithley, Thierry Giron and Janet Schank for convincing me USB Standard hubs are not that standard and that's good to allow for vendor specific quirks on the standard hub driver.