Training

5th February 2015

USB - from zero to gadget coder

Krzysztof Opasiak,
k.opasiak@samsung.com

Samsung R&D Institute Poland

Agenda

USB overview

Plug and Play

Device overview

ConfigFS composite gadget

libusbg

Credits

- **Alan Ott,**
  USB and a Real World
- **Andrzej Pietrasiewicz,**
  Make your own USB gadget

USB overview

Why USB?

- **Universal**
- **Fast enough**
  - 2.0 - 480 Mb/sec
  - 3.0 - 5.0 Gb/sec
- **Hot pluggable**
- **POPULAR**

Host vs Device

HOST

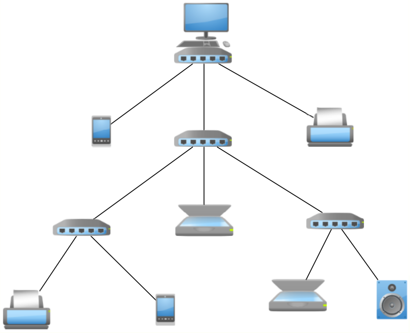
- Can be extended using some devices
- Complex, responsible for management
- Usually bigger machine (desktop, notebook etc.)

DEVICE

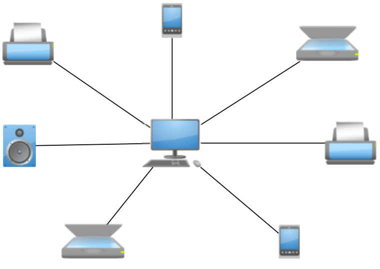
- Provide some functionality
- May extend USB HOST
- Usually small and simple

Logical vs physical topology

Physical

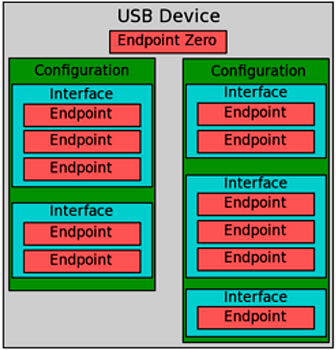


Logical



What is device?

- **Hardware**
- **USB protocol implementation**
- **Some useful protocol implementation**



Endpoint directions

- **Endpoint 0 may transfer data in both directions**
- **All other endpoints may transfer data in one direction:**

IN	**Transfer data from device to host**
OUT	**Transfer data from host to device**

Endpoint types

- **Control**
  - Bi-directional endpoint
  - Used for enumeration
  - Can be used for application

- **Interrupt**
  - Transfers a small amount of low-latency data
  - Reserves bandwidth on the bus
  - Used for time-sensitive data (HID)

Endpoint types

- **Bulk**
  - Used for large data transfers
  - Used for large, time-insensitive data (Network packets, Mass Storage, etc).
  - Does not reserve bandwidth on bus, uses whatever time is left over

- **Isochronous**
  - Transfers a large amount of time-sensitive data
  - Delivery is not guaranteed (no ACKs are sent)
  - Used for Audio and Video streams
  - Late data is as good as no data
  - Better to drop a frame than to delay and force a re-transmission

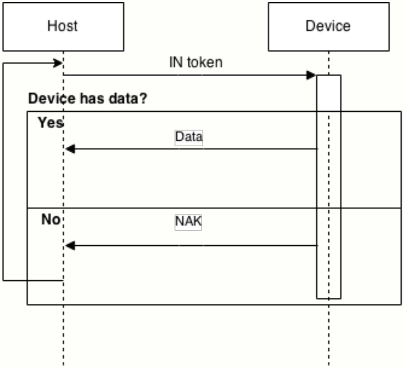
USB bus

- USB is a Host-controlled bus
- Nothing on the bus happens without the host first initiating it.
- Devices cannot initiate any communication.
- The USB is a Polled Bus.
- The Host polls each device, requesting data or sending data.

USB transport

IN

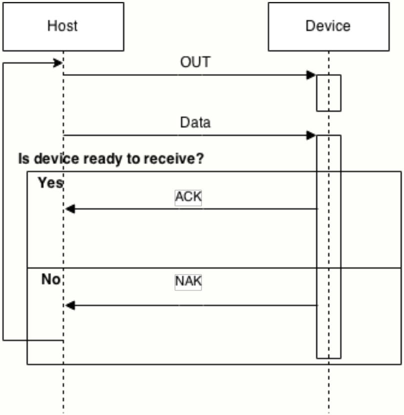
- **Host sends an IN token**
- **If the device has data:**
  - Device sends data
  - Host sends ACK
- **else**
  - Device sends NAK
  - Host will retry until timeout



USB transport

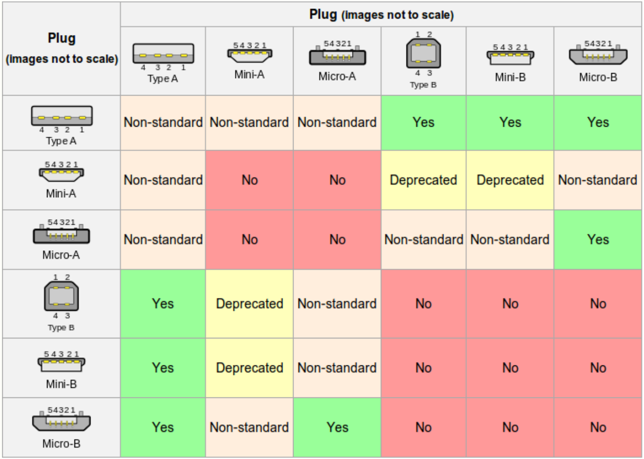
OUT

- **Host sends an OUT token**
- **Host sends the data (one packet)**
- **If device accepts data transfer:**
  - Device sends an ACK
- **else**
  - Device sends an NAK
  - Host will retry until success or timeout



Plug and Play

What should I plug?

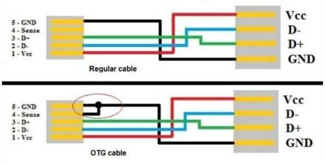


Plug (images not to scale)	Plug (images not to scale)					
	4 3 2 1 Type A	5 4 3 2 1 Mini-A	5 4 3 2 1 Micro-A	1 2 4 3 Type B	5 4 3 2 1 Mini-B	5 4 3 2 1 Micro-B
4 3 2 1 Type A	Non-standard	Non-standard	Non-standard	Yes	Yes	Yes
5 4 3 2 1 Mini-A	Non-standard	No	No	Deprecated	Deprecated	Non-standard
5 4 3 2 1 Micro-A	Non-standard	No	No	Non-standard	Non-standard	Yes
1 2 4 3 Type B	Yes	Deprecated	Non-standard	No	No	No
5 4 3 2 1 Mini-B	Yes	Deprecated	Non-standard	No	No	No
5 4 3 2 1 Micro-B	Yes	Non-standard	Yes	No	No	No

A is usually for host and B for device

OTG magic

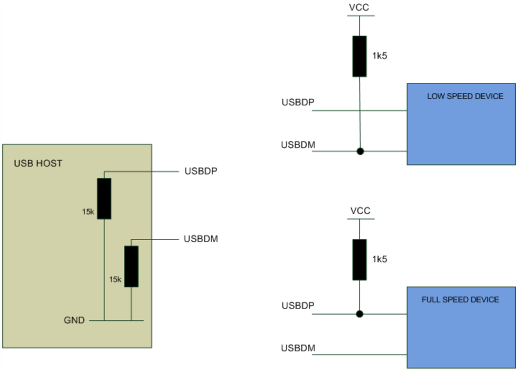
SAMSUNG
17:01
Home
Up
/sdcard
samsungapps
ScreenCapture
ShareViaWifi
SMSBackupRestore
Sounds
usbStorage
WhatsApp


5 - GND
4 - Sense
3 - D+
2 - D-
1 - Vcc
Vcc
D-
D+
GND
Regular cable
5 - GND
4 - Sense
3 - D+
2 - D-
1 - Vcc
Vcc
D-
D+
GND
OTG cable

Enumeration

- Plug in device
- Detect Connection
- Get basic info
- Set address
- Get more details
- Choose a driver
- Choose configuration
- Use it ;)

Detect Connection



What with high-speed? We try to communicate using high speed. If successful the device is HS and FS otherwise .

Get basic info

Offset	Field	Size	Value	Description
0	bLength	1	Number	Size of the Descriptor in Bytes (18 bytes)
1	bDescriptorType	1	Constant	Device Descriptor (0x01)
2	bcdUSB	2	BCD	USB Specification Number which device complies too.
4	**bDeviceClass**	1	Class	Class Code (by USB Org)
5	**bDeviceSubClass**	1	SubClass	Subclass Code (by USB Org)
6	**bDeviceProtocol**	1	Protocol	Protocol Code (by USB Org)
7	bMaxPacketSize	1	Number	Maximum Packet Size for Zero Endpoint. Valid Sizes are 8, 16, 32, 64
8	**idVendor**	2	ID	Vendor ID (by USB Org)
10	**idProduct**	2	ID	Product ID (by Manufacturer)
12	bcdDevice	2	BCD	Device Release Number
14	iManufacturer	1	Index	Index of Manufacturer String Descriptor
15	iProduct	1	Index	Index of Product String Descriptor
16	iSerialNumber	1	Index	Index of Serial Number String Descriptor
17	bNumConfigurations	1	Integer	Number of Possible Configurations

Digression - USB classes

00h	Device	Use class information in the Interface Descriptors
01h	Interface	Audio
02h	Both	Communications and CDC Control
03h	Interface	HID (Human Interface Device)
05h	Interface	Physical
06h	Interface	Image
07h	Interface	Printer
08h	Interface	Mass Storage
09h	Device	Hub
0Ah	Interface	CDC-Data
0Bh	Interface	Smart Card
0Dh	Interface	Content Security
0Eh	Interface	Video
0Fh	Interface	Personal Healthcare
10h	Interface	Audio/Video Devices
11h	Device	Billboard Device Class
DCh	Both	Diagnostic Device
E0h	Interface	Wireless Controller
EFh	Both	Miscellaneous
FEh	Interface	Application Specific
FFh	Both	Vendor Specific

Set address

- On plug-in device use default address 0x00
- Only one device is enumerated at once
- Hosts assigns unique address for new device

Get more details

- **Get Configurations descriptors**
- **Get Interfaces descriptors**
  - bInterfaceClass
  - bInterfaceSubClass
  - bInterfaceProtocol
- **Get Endpoints descriptors**
- **Get Strings**

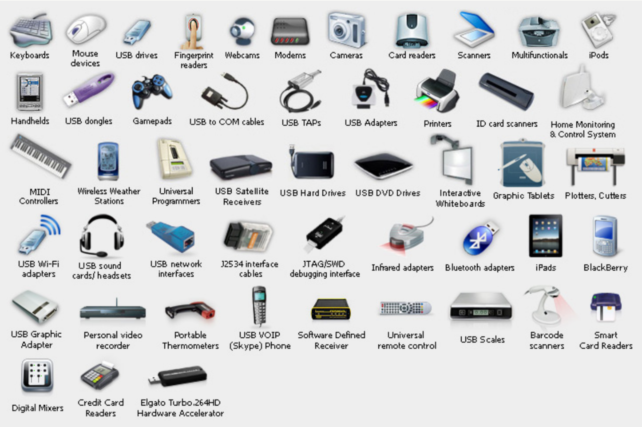
Choose a driver

- Select driver for matching VID, PID
- Select driver for matching Class, Subclass, Protocol
- Select custom driver, use first configuration and try to find drivers for each interface

Choose configuration

- At this moment device is not configured (Config 0)
- We may use one of available configurations
- If we have a driver dedicated for this device he will select suitable for us
- By default we will use first one

Use it;)

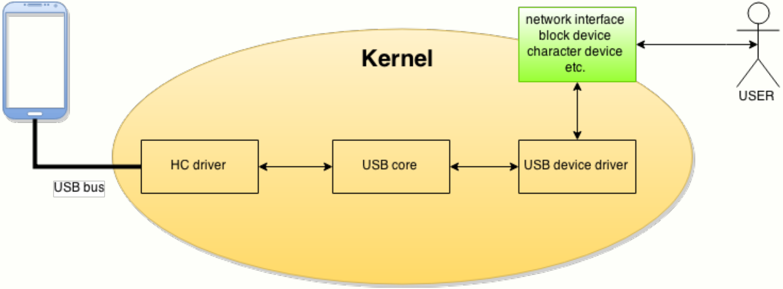

Keyboards
Mouse devices
USB drives
Fingerprint readers
Webcams
Modems
Cameras
Card readers
Scanners
Multifunctionals
iPods
Handhelds
USB dongles
Gamepads
USB to COM cables
USB TAPs
USB Adapters
Printers
ID card scanners
Home Monitoring & Control System
MIDI Controllers
Wireless Weather Stations
Universal Programmers
USB Satellite Receivers
USB Hard Drives
USB DVD Drives
Interactive Whiteboards
Graphic Tablets
Plotters, Cutters
USB Wi-Fi adapters
USB sound cards/ headsets
USB network interfaces
J2534 interface cables
JTAG/SWD debugging interface
Infrared adapters
Bluetooth adapters
iPads
BlackBerry
USB Graphic Adapter
Personal video recorder
Portable Thermometers
USB VOIP (Skype) Phone
Software Defined Receiver
Universal remote control
USB Scales
Barcode scanners
Smart Card Readers
Digital Mixers
Credit Card Readers
Elgato Turbo.264HD Hardware Accelerator

Linux terminology

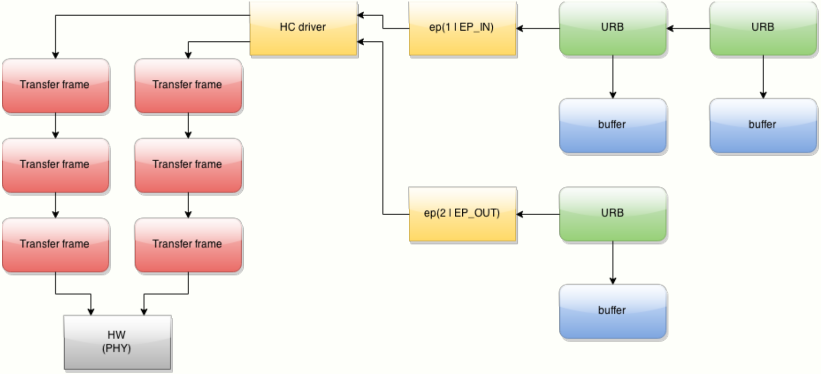
HC driver Driver for USB Host controller

USB device driver driver for particullar USB device (or class/subclass)

- **Communicate with the device**
- **Export its functionality to user space (network interface, block device etc)**



USB request block



libusb

- **Sometimes we don't have a kernel driver**
- **We would like to write driver in userspace**
- **libusb allows for this!**
- **Kernel forwards USB messages to userspace**
- **libusbg provide abstraction layer for kernel API**

Synchronous API

```c
unsigned char buf[64];
int actual_length;
do {
  /* Receive data from the device */
  res = libusb_bulk_transfer(handle, 0x81, buf,
           sizeof(buf), &actual_length, 100000);
  if (res < 0) {
    fprintf(stderr, "bulk transfer (in): %s\n",
           libusb_error_name(res));
    return 1;
  }
} while (res >= 0);
```

Asynchronous API

```
static struct libusb_transfer
  *create_transfer(libusb_device_handle *handle, size_t length) {
      struct libusb_transfer *transfer;
      unsigned char *buf;
      /* Set up the transfer object. */
      buf = malloc(length);
      transfer = libusb_alloc_transfer(0);
      libusb_fill_bulk_transfer(transfer,
                               handle,
                               0x81 /*ep*/,
                               buf,
                               length,
                               read_callback,
                               NULL/*cb data*/,
                               5000/*timeout*/);

     return transfer;
}
```

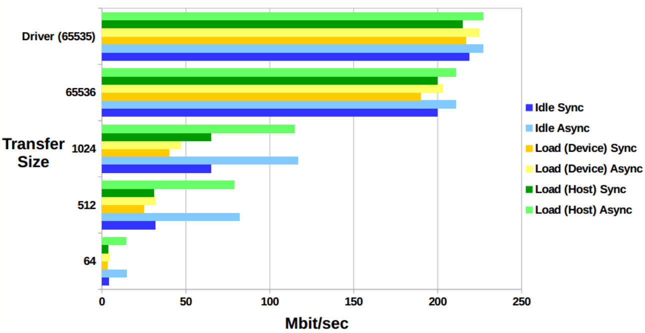
Asynchronous API

```c
static void read_callback(struct libusb_transfer *transfer)
{
    int res;
    if (transfer->status == LIBUSB_TRANSFER_COMPLETED) {
        /* Success! Handle data received */
    } else {
        printf("Error: %d\n", transfer->status);
    }
    /* Re-submit the transfer object. */
    res = libusb_submit_transfer(transfer);
    if (res != 0) {
        printf("submitting. error code: %d\n", res);
    }
}
```

Asynchronous API

```
/* Create Transfers */
for (i = 0; i < 32; i++) {
    struct libusb_transfer *transfer =
                             create_transfer(handle, buflen);
    libusb_submit_transfer(transfer);
}
/* Handle Events */
while (1) {
    res = libusb_handle_events(usb_context);
    if (res < 0) {
        printf("handle_events()error # %d\n", res);
        /* Break out of this loop only on fatal error.*/
        if (res != LIBUSB_ERROR_BUSY &&
            res != LIBUSB_ERROR_TIMEOUT &&
            res != LIBUSB_ERROR_OVERFLOW &&
            res != LIBUSB_ERROR_INTERRUPTED) {
              break;
            }
    }
}
```

When we should use asynchronous?



Conclusion: We should use async in most cases!

usb-utils

- **Human-readable version of most hex values**
- **ID's base**
- **lsusb - your biggest friend!**

  -v *print more details*
  -d VID:PID *print information about selected devices*
  -t *print physical USB tree*

USB host side demo

DEMO

Device overview

Quick checkpoint

- **We know what is:**
  - USB device
  - Configuration
  - Interface
  - Endpoint
- **We have heard about USB descriptors**
- **We have heard about USB transfers**
- **We arenow on DEVICE side (of power)**

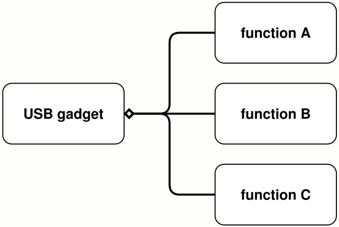
Linux terminology

UDC driver Driver for USB UDC controller

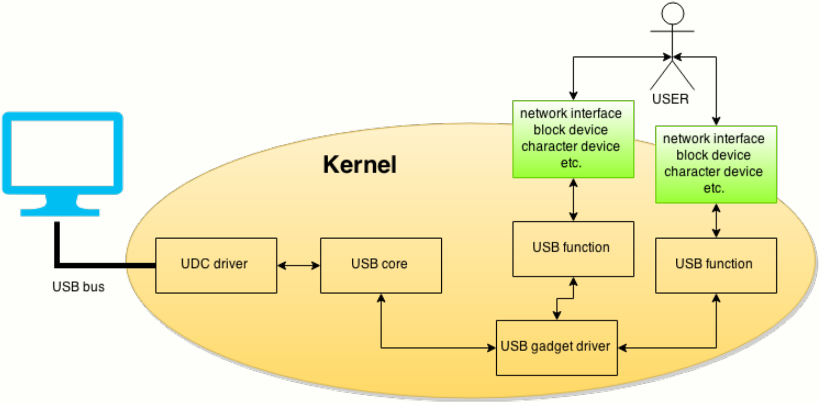
USB function (type) driver which implements some useful protocol (HID, Mass storage)

USB gadget Glue layer for functions

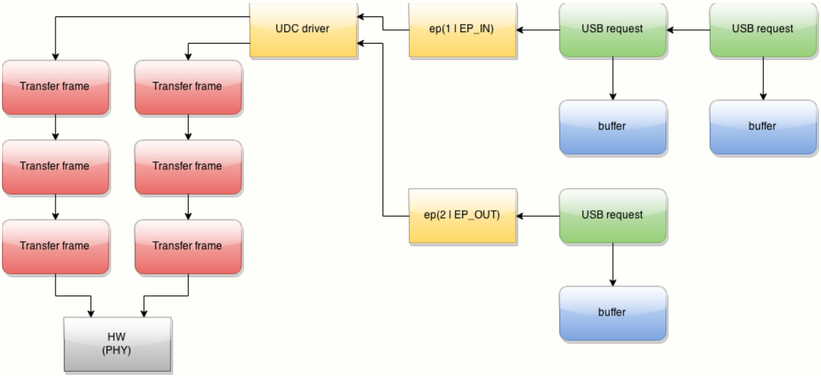
- **Handle enumeration**
- **Respond to most general requests**



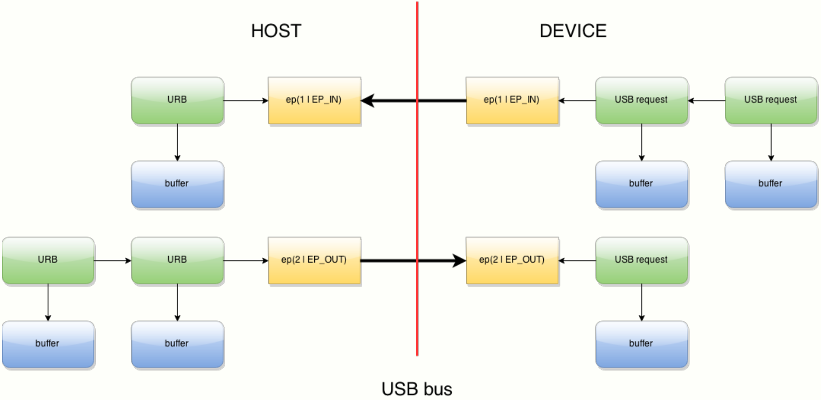
Linux terminology



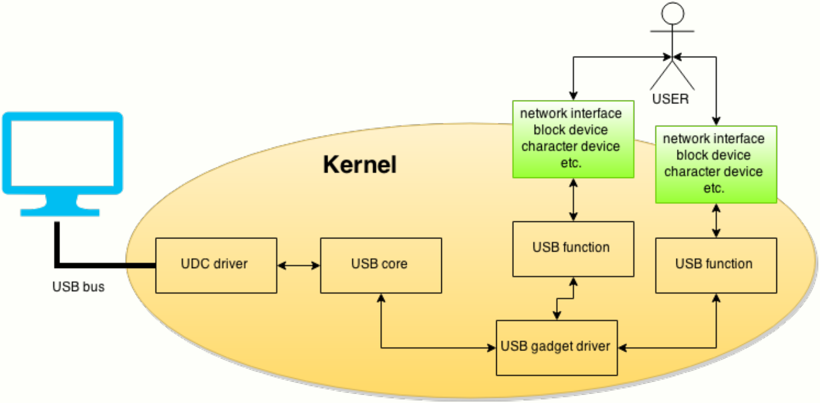
USB request


UDC driver
ep(1 I EP_IN)
USB request
USB request
buffer
buffer
ep(2 I EP_OUT)
USB request
buffer
Transfer frame
Transfer frame
Transfer frame
Transfer frame
Transfer frame
Transfer frame
HW
(PHY)

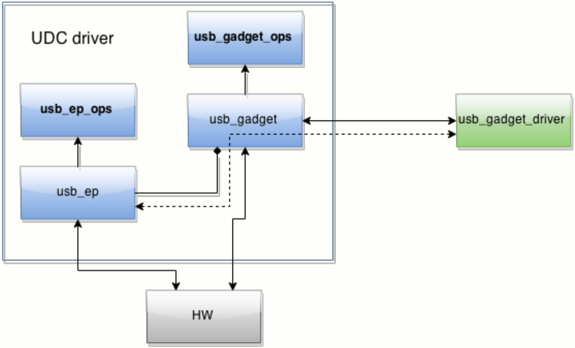
URB vs USB request



Linux terminology - once again

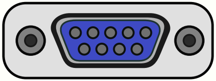


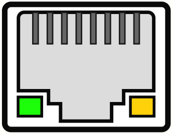
UDC driver


UDC driver
usb_gadget_ops
usb_ep_ops
usb_gadget
usb_gadget_driver
usb_ep
HW

Available functions

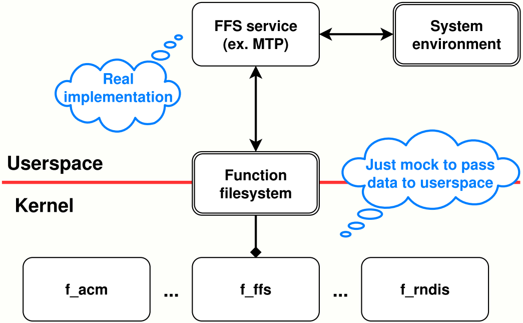
- **Serial**
  - ACM
  - Serial
  - OBEX
- **Ethernet**
  - ECM
  - EEM
  - NCM
  - Subset
  - RNDIS
- **Phonet**
- **Mass Storage**
- **Loopback and SourceSink**
- **UVC**
- **HID**





Our own function

- Access to system environment
- Easier implementation
- Kernel USB function & file system
- Wraps file IO operations into usb_requests



FunctionFS - HOWTO

- **Usage:**
  - open ep0 file
  - Write function descriptors
  - Write function strings
  - Open epXX (if any)
  - Read events from ep0
  - . . . (Protocol specific)
- **Performance - use Async IO**
- **See Alan's Ott presentation:**

"USB and the Real World"

Base composition

- **Fill the identity of gadget**
  - Vendor ID
  - Product ID
  - Device Class details
  - Strings (manufacturer, product and serial)
- **Decide what functions it has**
- **Decide how many configurations**
- **Decide what functions are available in each configuration**

How we can compose a gadget?

We can write a kernel module!

How we can compose a gadget?

We can write a kernel module!

...but don't do this!

Stay on the light side - use ConfigFS!

ConfigFS composite gadget

ConfigFS intro

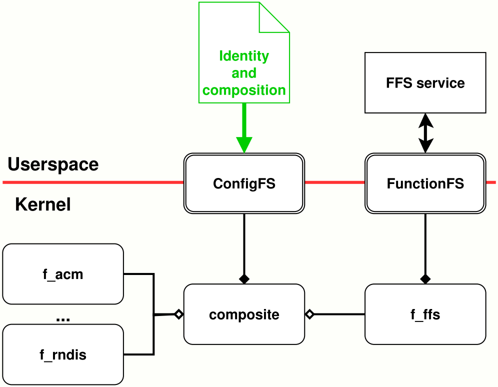
- Created by Joel Becker in 2005
- Generic filesystem for kobject management
- ConfigFS vs SysFS
- API to write your own subsystem

ConfigFS base concept

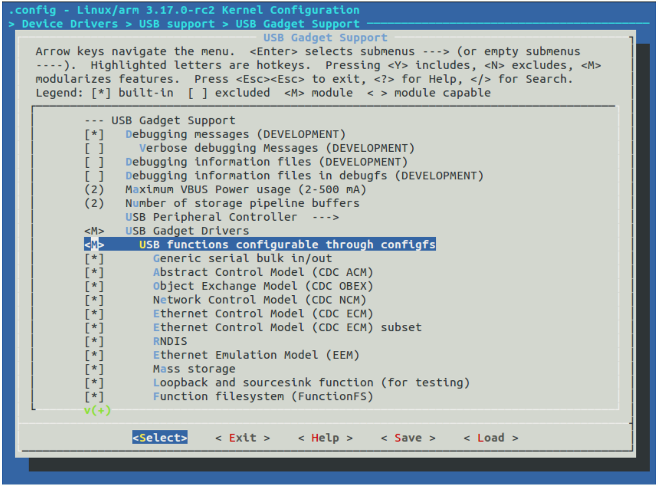
kobject action	Filesystem operation
create	mkdir
destroy	rmdir
set attribute value	write
get attribute value	read
association	ln -s
remove association	rm (on symlink)

Gadget composition: ConfigFS

- **Allow user to compose gadget at runtime**
- **Register as subsystem in ConfigFS**
- **Use file system ops to compose a gadget**
- **Load module on request**



Prerequisites - menuconfig



```
.config - Linux/arm 3.17.0-rc2 Kernel Configuration
> Device Drivers > USB support > USB Gadget Support ──────────────────────────
                         USB Gadget Support
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty submenus
  ----).  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes, <M>
  modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </> for Search.
  Legend: [*] built-in  [ ] excluded  <M> module  < > module capable

        --- USB Gadget Support
        [*]   Debugging messages (DEVELOPMENT)
        [ ]     Verbose debugging Messages (DEVELOPMENT)
        [ ]   Debugging information files (DEVELOPMENT)
        [ ]   Debugging information files in debugfs (DEVELOPMENT)
        (2)   Maximum VBUS Power usage (2-500 mA)
        (2)   Number of storage pipeline buffers
              USB Peripheral Controller  --->
        <M>   USB Gadget Drivers
        <M>     USB functions configurable through configfs
        [*]       Generic serial bulk in/out
        [*]       Abstract Control Model (CDC ACM)
        [*]       Object Exchange Model (CDC OBEX)
        [*]       Network Control Model (CDC NCM)
        [*]       Ethernet Control Model (CDC ECM)
        [*]       Ethernet Control Model (CDC ECM) subset
        [*]       RNDIS
        [*]       Ethernet Emulation Model (EEM)
        [*]       Mass storage
        [*]       Loopback and sourcesink function (for testing)
        [*]       Function filesystem (FunctionFS)
        v(+)

             <Select>    < Exit >    < Help >    < Save >    < Load >
```

Give a chance to request_module() - use depmod

Prologue

Prologue

```
$ modprobe libcomposite
$ mount none -t configfs /sys/kernel/config
$ cd /sys/kernel/config/usb_gadget
```

Simple example

Create gadget, fill identity

```
$ mkdir g1
$ cd g1
$ echo "0x1d6b" > idVendor
$ echo "0x0104" > idProduct
$ mkdir strings/0x409
$ echo "My serial" > strings/0x409/serialnumber
$ echo "My Vendor" > strings/0x409/manufacturer
$ echo "My Product" > strings/0x409/product
```

Simple example

One config, one function

```
$ mkdir functions/rndis.usb0
$ mkdir configs/c.1
$ mkdir configs/c.1/strings/0x409
$ echo "Config 1" > \
        configs/c.1/strings/0x409/configuration
$ ln -s functions/rndis.usb0 configs/c.1/
```

Simple example

Check UDC name

```
$ ls /sys/class/udc
12480000.hsotg
```

Bind gadget to udc

```
$ echo "12480000.hsotg" > UDC
```

Simple example

On host side

```
$ lsusb -v
Bus 003 Device 026: ID 1d6b:0104
  bcdUSB             0.00
  bDeviceClass          0
  bDeviceSubClass       0
  bDeviceProtocol       0
  bMaxPacketSize0      64
  idVendor         0x1d6b Linux Foundation
  idProduct        0x0104 Multifunction Composite
  bcdDevice          3.17
  iManufacturer         1 My Vendor
  iProduct              2 My Product
  iSerial               3 My serial
  bNumConfigurations    1
```

Bare shell demo

DEMO

ConfigFS and FunctionFS

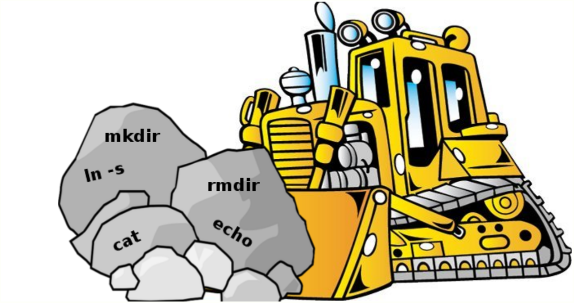
ConfigFS modifications

```
$ echo "" > UDC
$ mkdir functions/ffs.my_func_name
$ ln -s functions/ffs.my_func_name configs/c.1/
$ mount my_func_name -t functionfs /tmp/mount_point
$ run_function_daemon
$ wait_for_daemon_initialization
$ echo "12480000.hsotg" > UDC
```

libusbg

Why?

- **Allow to create gadget from code**
- **Provide abstraction layer for ConfigFS**
- **Reduce number of magic numbers**
- **Limit number of potential mistakes**
- **Allow for fast and easy gadget creation**
- **Make gadget creation declarative**



C API Overview

- **Opaque structures for all entities:**
    - usbg_state
    - usbg_gadget
    - usbg_config
    - usbg_function
    - usbg_binding
    - usbg_udc
- **usbg_gadget_attrs - gadget attributes, similar layout to libusb_device_descriptor ;)**
- **No static buffers, reentrant**
- **Snapshot taken on initialization**

Example

Attributes and strings

```
static usbg_gadget_attrs g_attrs = {
  /* Class defined at interface level */
  .idVendor = 0x1d6b,
  .idProduct = 0x104,
};

usbg_gadget_strs g_strs = {
  .str_ser = "My serial",
  .str_mnf = "My Vendor.",
  .str_prd = "My Product Name",
};

usbg_config_strs c_str = {
  "Config 1"
};
```

Example

Gadget creation

```
usbg_init("/sys/kernel/config", &s);

usbg_create_gadget(s, "g1", &g_attrs, &g_strs, &g);
usbg_create_function(g, F_RNDIS, "usb0", NULL, &f_rndis);
usbg_create_config(g, 1, "c", NULL, &c1_strs, &c);

usbg_add_config_function(c1, "rndis_func", f_rndis);

usbg_enable_gadget(g, DEFAULT_UDC);
usbg_cleanup(s);
```

C API demo

DEMO

Gadget schemes

- **Use configuration files for gadget composition**
- **libconfig syntax instead of reinventing the wheel**
- **Set of usbg_import_*() and usbg_export_*() functions**
- **Complete gadget setup with one command**

libconfig syntax

Canonical form

```
configuration = setting-list | empty
setting-list = setting | setting-list setting
setting = name (":" | "=") value (";" | "," | empty)
value = scalar-value | array | list | group
value-list = value | value-list "," value
scalar-value = boolean | integer | integer64 | hex |
               hex64 | float | string
scalar-value-list = scalar-value |
                 scalar-value-list "," scalar-value
array = "[" (scalar-value-list | empty) "]"
list = "(" (value-list | empty) ")"
group = "{" (setting-list | empty) "}"
empty =
```

Example

Canonical form

```
attrs = {
 idVendor = 0x1D6B
 idProduct = 0x104
}

strings = ({
  lang = 0x409;
  manufacturer = "My vendor"
  product = "My product"
  serialnumber = "My Serial"
})
```

Example

Canonical form

```
functions = {
  rndis_func = {
    instance = "usb0"
    type = "rndis"
  }
}

configs = ({
  id = 1
  name = "c"
  strings = ({
    lang = 0x409
    configuration = "Config 1"
  })
  functions = ("rndis_func")
})
```

Example

Shorter form

```
attrs = {idVendor = 0x1D6B; idProduct = 0x104;}

strings = ({
  lang = 0x409;
  manufacturer = "My vendor"
  product = "My product"
  serialnumber = "My Serial"
})

configs = ({
  id = 1
  name = "c"
  strings = ({lang = 0x409; configuration = "Config 1";})
  functions = ({function = {instance = "usb0"; type = "rndis";}})
})
```

Example

Gadget loader

```
usbg_init("/sys/kernel/config", &s);

file = fopen("my_gadget.gs", "r");

usbg_import_gadget(s, file, "g1", &g);

usbg_enable_gadget(g, DEFAULT_UDC);
usbg_cleanup(s);
```

Example

Gadget loader

```
usbg_init("/sys/kernel/config", &s);

file = fopen("my_gadget.gs", "r");

usbg_import_gadget(s, file, "g1", &g);

usbg_enable_gadget(g, DEFAULT_UDC);
usbg_cleanup(s);
```

- **More gadget schemes tweaks:**

  https://github.com/kopasiak/libusbg/tree/master/doc

- **More examples:**

  https://github.com/kopasiak/libusbg/tree/master/examples

Gadget schemes demo

DEMO

Summary

- **In USB we have host side and device side**
- **Host is responsible for bus management**
- **Linux supports both sides**
- **We can compose gadget using ConfigFS**

Questions?

Refereces

- Andrzej Pietrasiewicz, Make your own USB gadget
- Matt Porter, Kernel USB Gadget ConfigFS Interface
- https://github.com/libusbg/libusbg
- https://github.com/kopasiak/libusbg
- https://github.com/kopasiak/gt
- https://github.com/hyperrealm/libconfig
- http://lwn.net/Articles/395712/
- https://wiki.tizen.org/wiki/USB/Linux_USB_Layers/Configfs_Composite_Gadget