Enhancements to USB Gadgets

Embedded Linux Conference 2008

Conrad Röber

April 16, 2008

About the Author

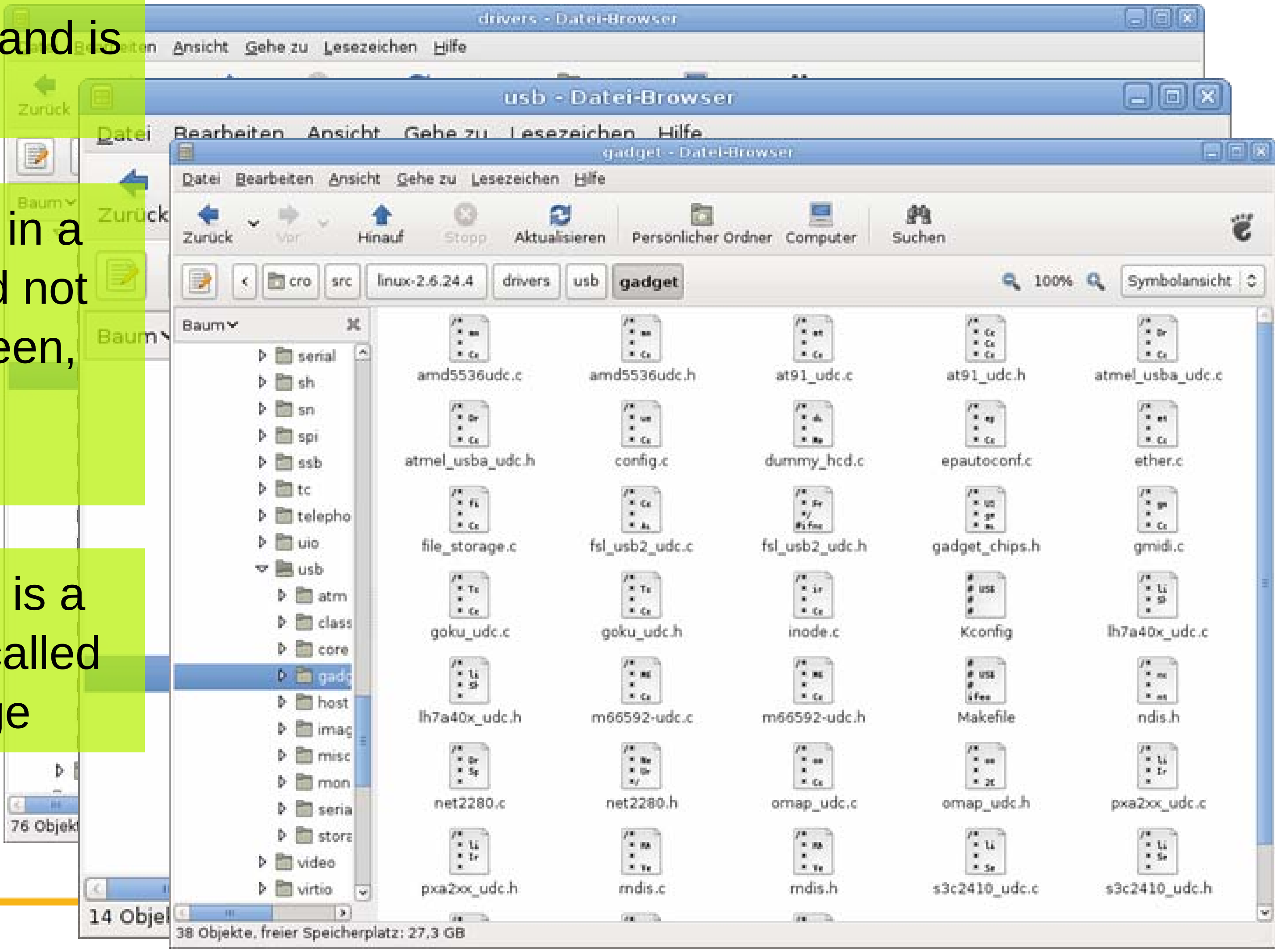
Conrad Röber

- I have a background in Physic but have spent my professional life working with computers, mostly embedded systems. Domains were process control and telecommunications.
- Special interest computing architectures and operating systems.

- I’m working in NXP’s *Business Line Cellular Systems*; the BL is making reference designs – silicon and software – for cellular phones.

The Topography

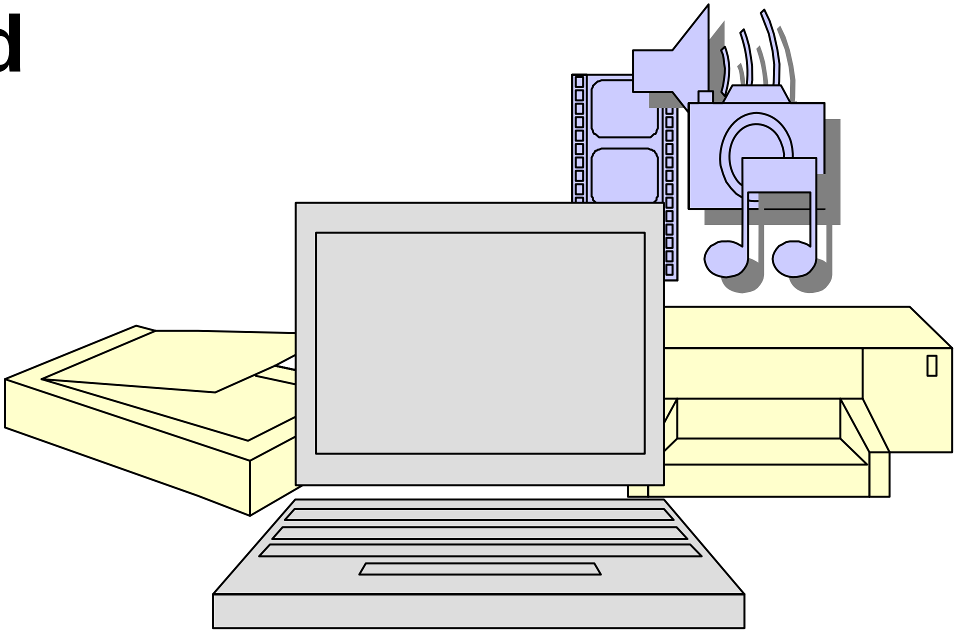
- Linux driver land is large
- Somewhere, in a place that did not fit on the screen, there is USB county
- Finally, there is a small place called Gadget village



amd5536udc.c amd5536udc.h at91_udc.c at91_udc.h atmel_usba_udc.c
atmel_usba_udc.h config.c dummy_hcd.c epautoconf.c ether.c
file_storage.c fsl_usb2_udc.c fsl_usb2_udc.h gadget_chips.h gmidi.c
goku_udc.c goku_udc.h inode.c Kconfig lh7a40x_udc.c
lh7a40x_udc.h m66592-udc.c m66592-udc.h Makefile ndis.h
net2280.c net2280.h omap_udc.c omap_udc.h pxa2xx_udc.c
pxa2xx_udc.h rndis.c rndis.h s3c2410_udc.c s3c2410_udc.h

38 Objekte, freier Speicherplatz: 27,3 GB

Desktop vs. Embedded



- Unlike other buses, USB is asymmetric – it distinguishes *host* and *device*
- Desktop computers act as USB *host*. The Linux host implementation is mature solution – after all, it is used in millions of desktop systems
- The Linux *device* side – named Gadget – is less developed.
- I am going to describe in the following:
  - What is there …
  - What are the requirements and where do they stem from …
  - What is missing …
  - How the gap can be filled

Contents

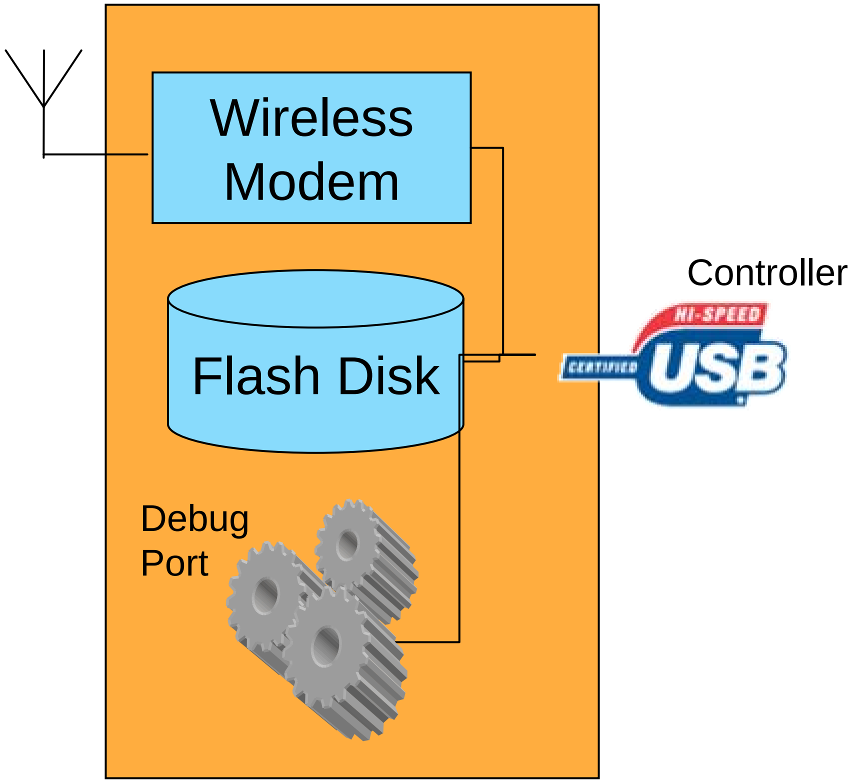
Motivation

Motivation

- There are things that one can sensibly implement with USB (and which *are* implemented in NXP's non-Linux reference design), …
- …but go beyond what the Linux USB Gadget allows.

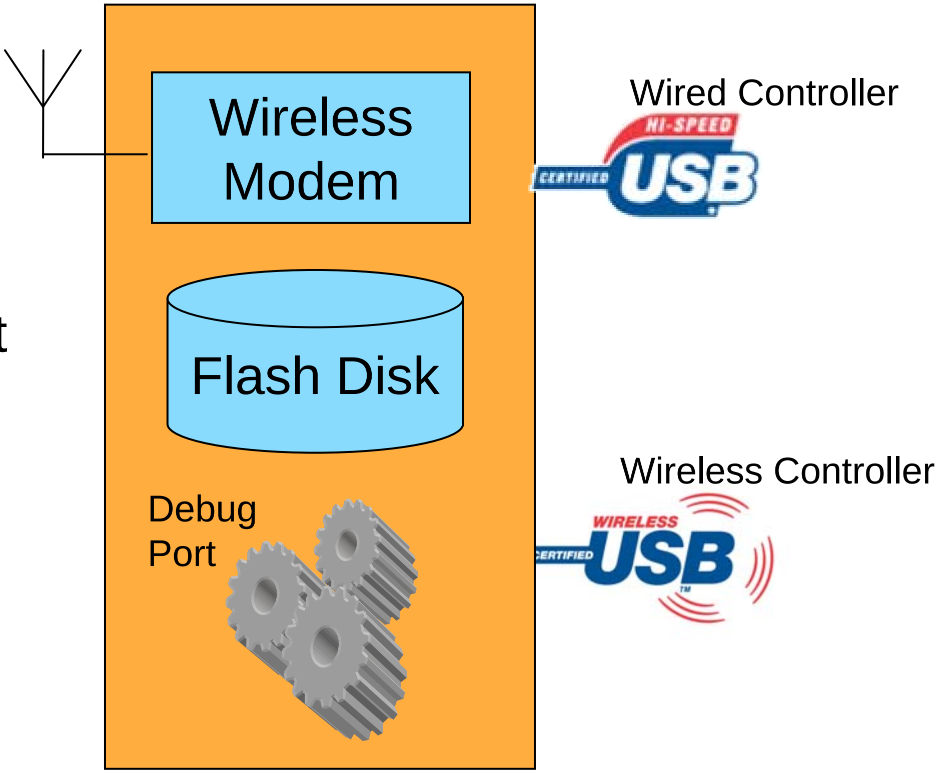
Motivation 1 / 2

- *Composite USB devices* have multiple functions accessible through a single USB port
- Example:
  - Mobile phone, composed of a wireless modem, a USB flash disk, and a debug port
- The USB gadget framework does not support the aggregation of several functions; *only one function at a time is accessible via USB*.



Motivation 2 / 2

- USB devices may have *more than one USB controller*
- Example:
  - Mobile phone, having both wired and wireless controllers
- The USB gadget framework supports *only a single controller* at a time.

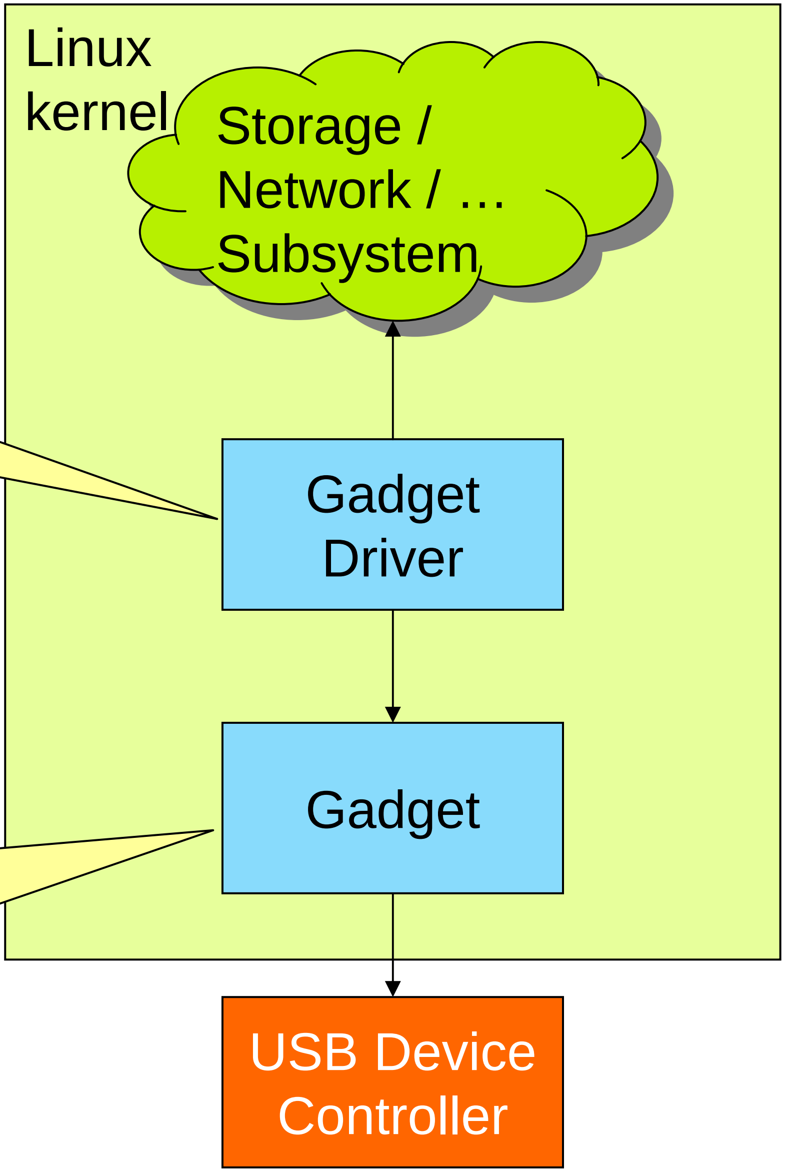


Background

Terminology 1 / 2

- *Gadget driver*: Implements the functions of one or more USB classes.
- Equivalent to the term "class driver".

- *Gadget*: USB device controller as used in the Linux USB Gadget framework
- Sometimes used as "Linux system with role of a USB device"

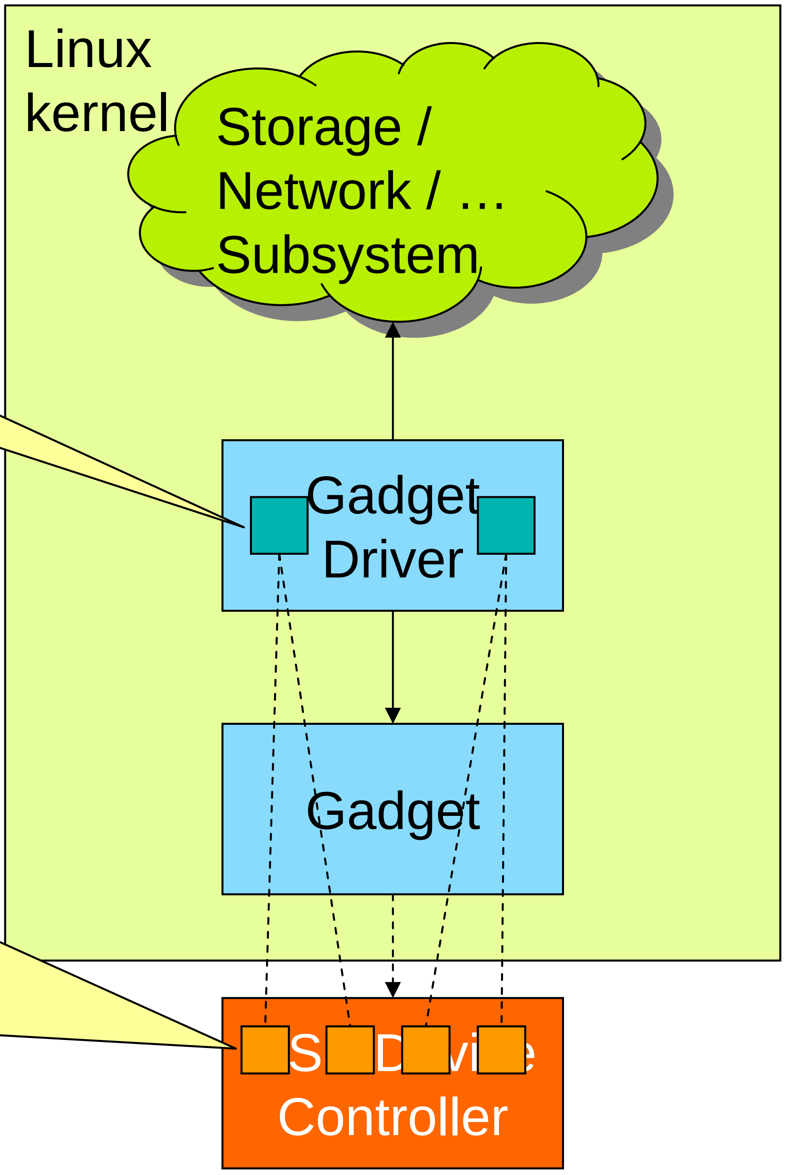


Terminology 2 / 2

- *Interface*: An interface is a logical group of endpoints making a certain functionality accessible over USB.

Example: The Mass Storage Class defines a single interface consisting of one IN and one OUT endpoint and transporting SCSI commands.

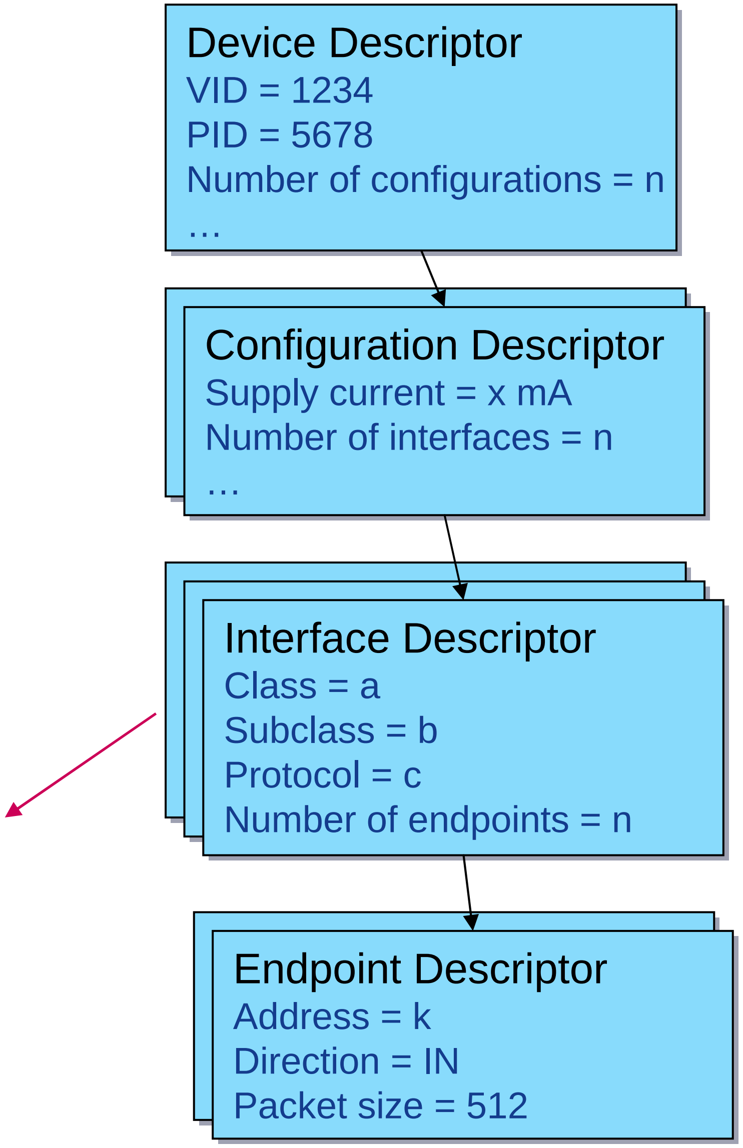
- *Endpoint*: A uniquely addressable portion of a USB device that is the source or sink of information in a communication flow between the host and device.
- Endpoints are realised in silicon as a packet buffer connected to the serial engine.
- They are uniquely identified by their number (0 ..15) and their direction (*IN* or *OUT*)



Descriptors 1/2

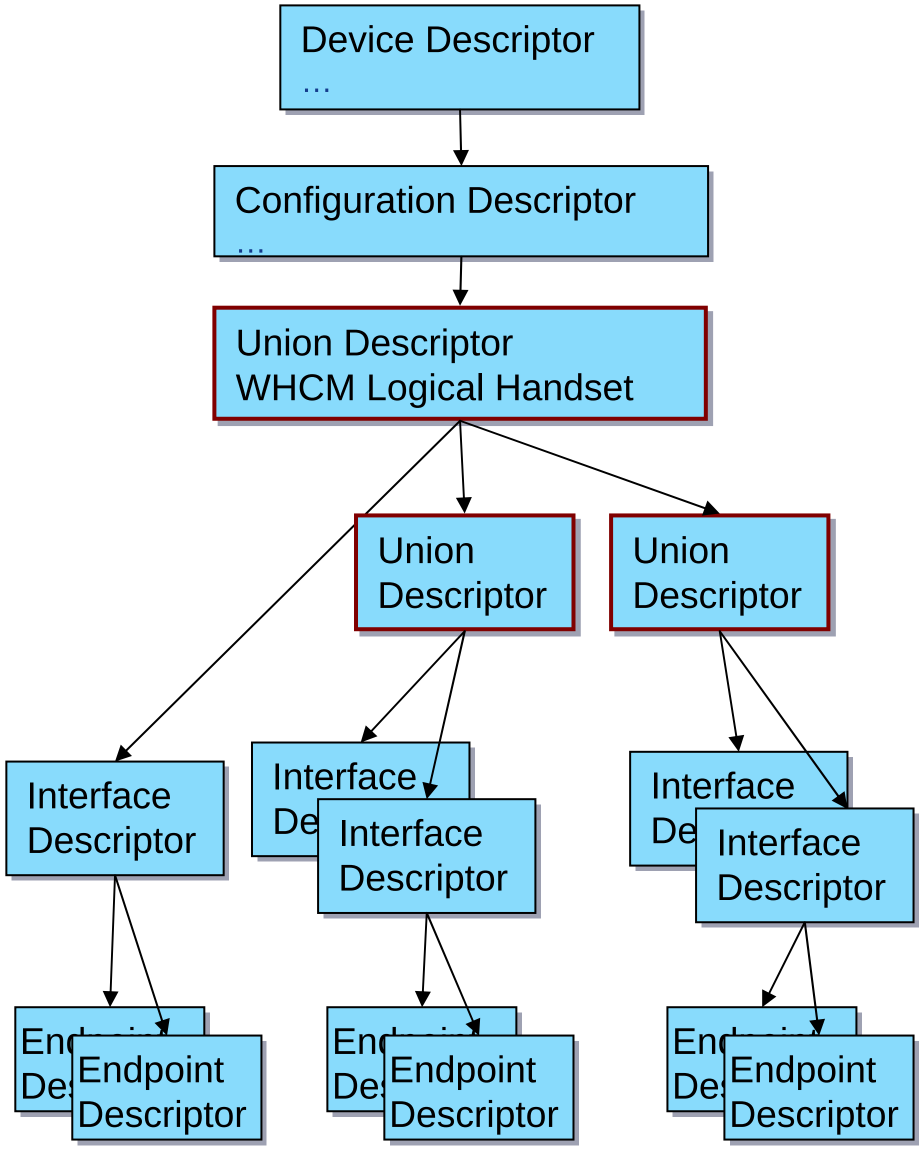
- USB devices are self-describing:
  - From the formal description alone, a host can determine capabilities, functions, protocols, etc.
  - The USB standard defines a four-level descriptor hierarchy

- Example USB flash disk
  - Wrapper to transport SCSI commands over USB
  - *Class* = Mass Storage
  - *Subclass* = MMC-2 (SCSI dialect)
  - *Protocol* = Bulk-Only Transport



Descriptors 2 / 2

- 4-layer descriptor hierarchy *cannot adequately describe complex devices* like cellular handsets
- Addressed by the introduction of *union* and *interface association* descriptors: they act as a bracket to group logically related interfaces
- Because of this ad-hoc extension, the descriptors become difficult to generate (by the device) and difficult to parse (by the host).

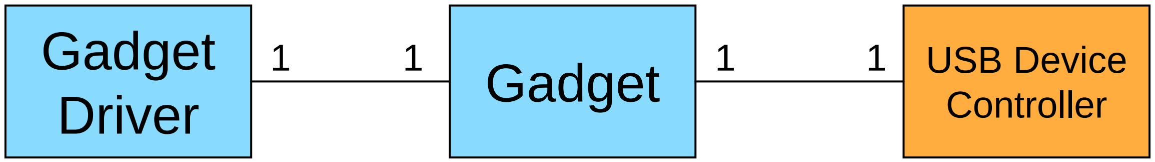


- Example:
  - Telephone Control Model:
    - *Audio* interface
    - Associated AT command *call control* interface

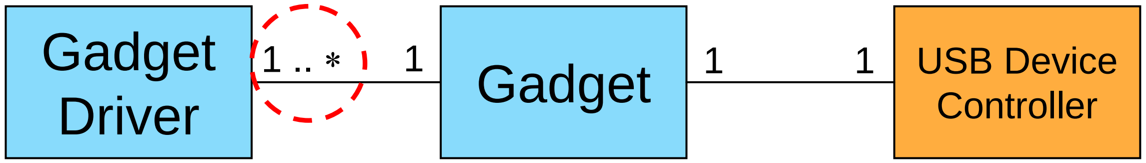
Requirements

Functional Gap 1 / 2

- In a more formal way:
  - Limitation is that *exactly one* Gadget Driver is bound to a Gadget.



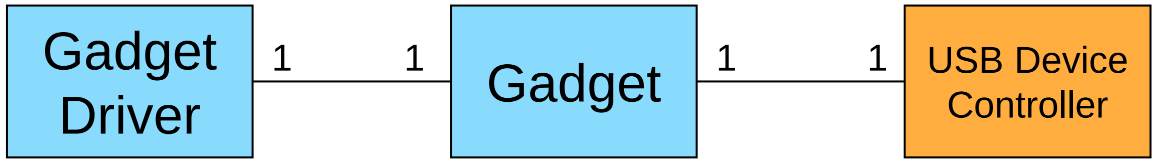
- Desired functionality:
  - *n* different Gadget Drivers can be bound to a given Gadget



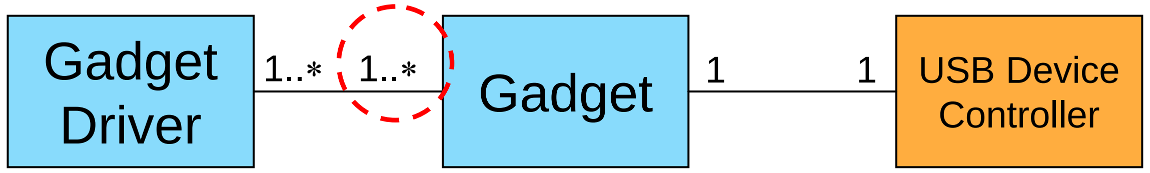
- This addresses the use case:
  - Composite device: Example:
    - *Wireless modem* plus *USB flash disk*
  - Several instances of one Gadget driver. Example:
    - Network connection to (public) Internet *and* to (private) operator IP network for MMS or WAP

Functional Gap 2 / 2

- Second limitation:
  - *Exactly one* Gadget and *one* controller may exist.



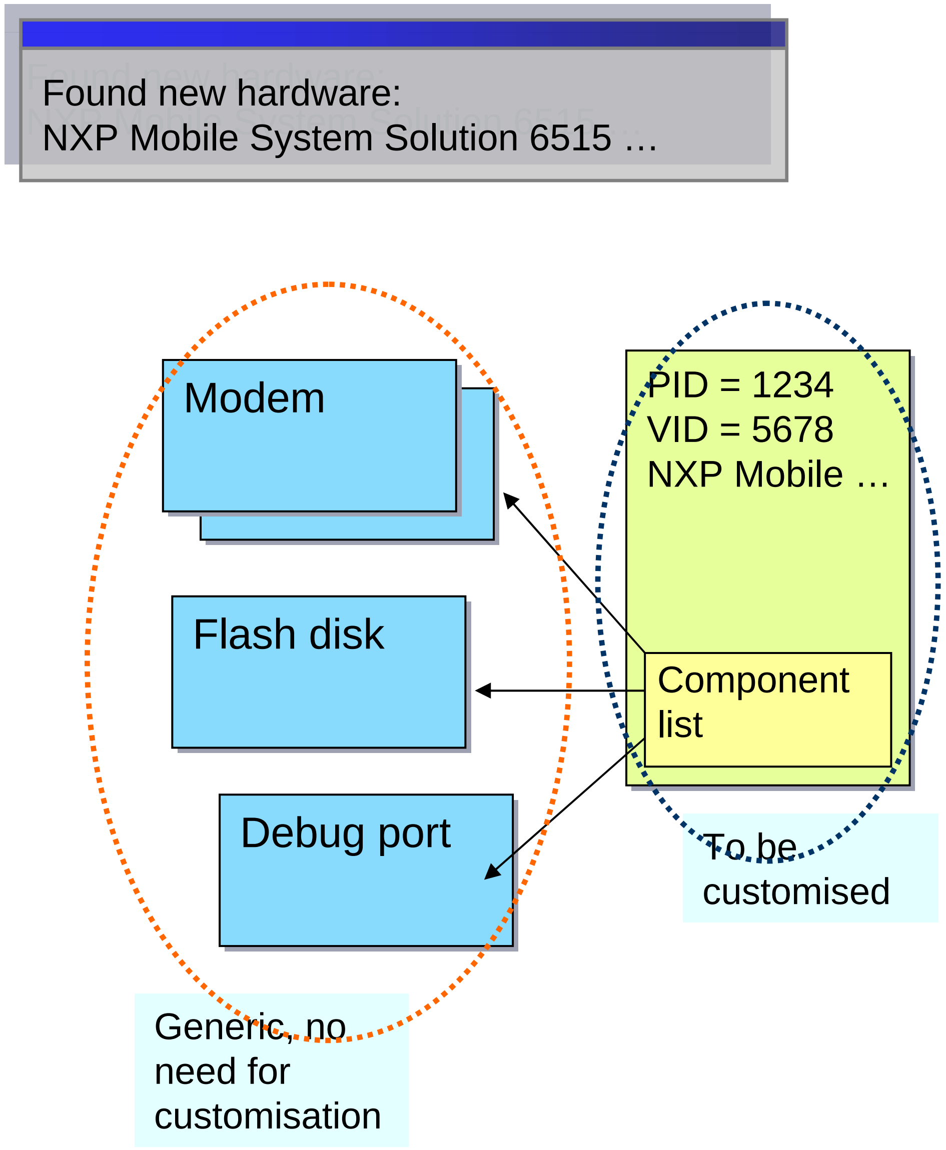
- Desired functionality:
  - *n* different Gadget Drivers can be bound to *m* different Gadgets



- This addresses the use cases:
  - Wireless USB: Device has 1 wireless and 1 wired controller
  - Other cases where more than one controller is needed.

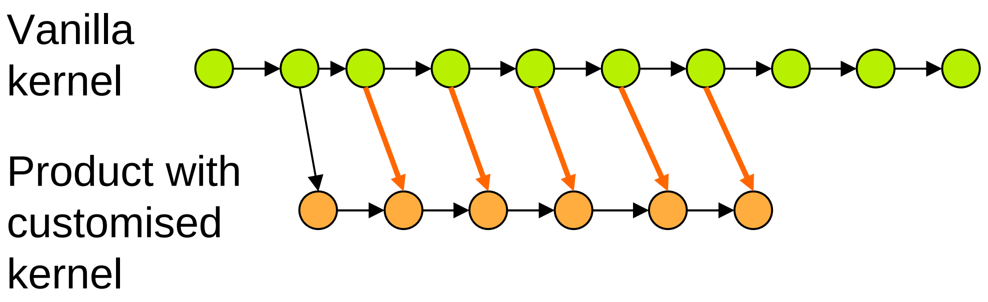
Customisation 1 / 2

- A Gadget has customisable items:
  - Numeric vendor and product IDs
  - Strings like manufacturer and product names
- These items are currently hard-coded in the Gadget drivers: Only appropriate for single-function devices, not for composite devices
- Names should refer to the product as a *collection* of functions, not to individual functions

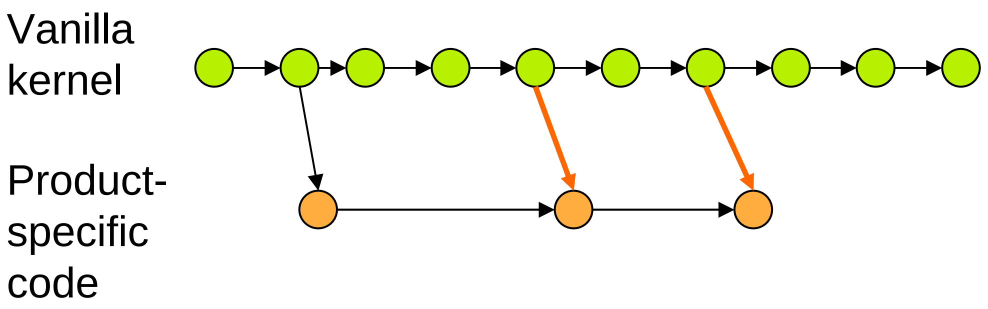


Customisation 2 / 2

- Separation of customisable and generic kernel code is vital for companies making (embedded) Linux products. (Desktop distros have other means of customisation, e.g. the GUI.)
- If properly separated:
  - Companies can follow evolution of the "vanilla" kernel without having to merge their modifications in.
- Only if interfaces change, companies have to update their customised code. Interface changes are less frequent than compatible changes like bug fixes.



Mix of generic and custom code:
Frequent merge of kernel changes even if the product spec does not change.



Separation of generic and custom code:
Less frequent merging; kernel and product can have independent evolution.
• Less effort
• Less error-prone

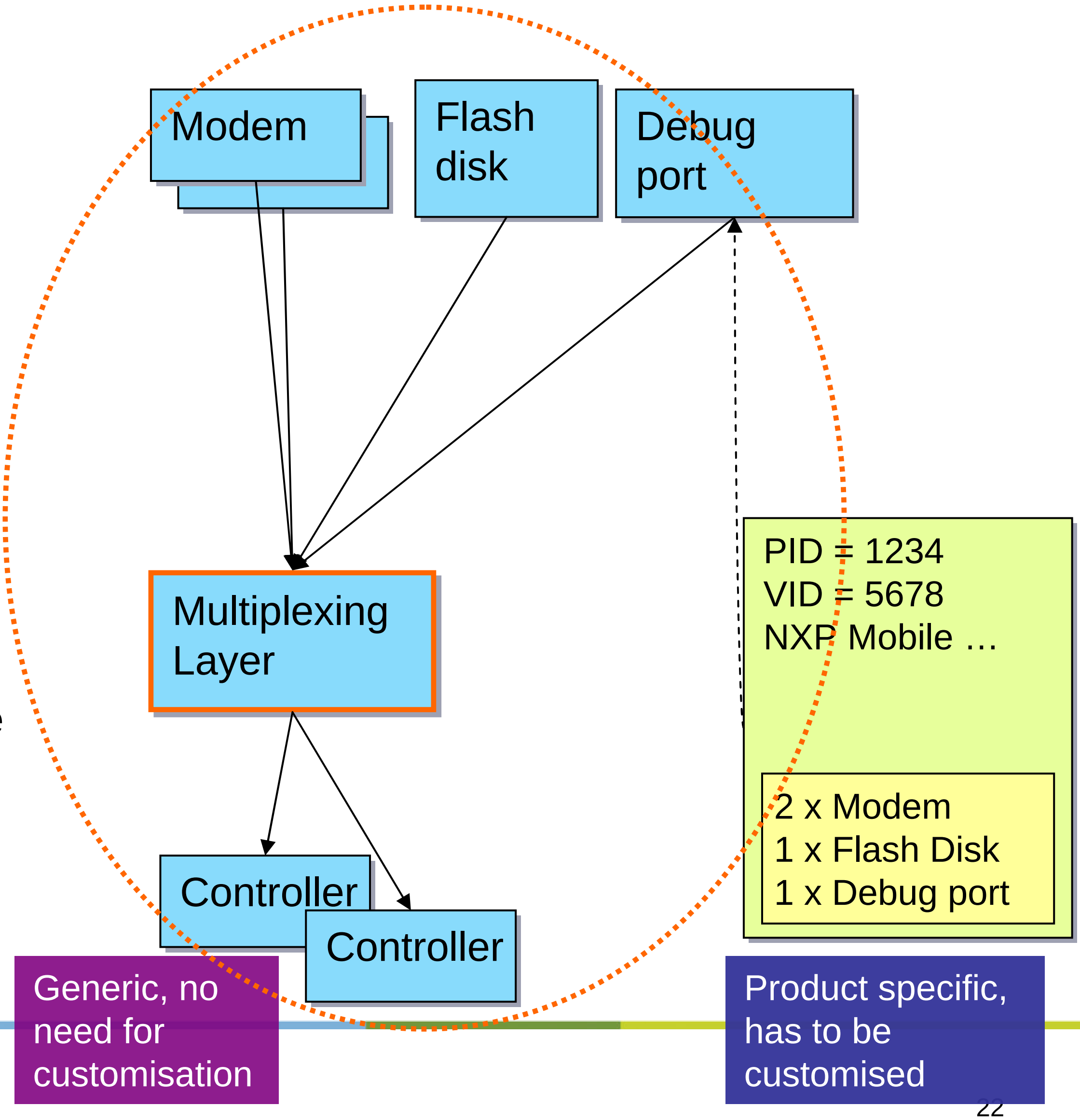
Solution

Just to make sure …

- What I'm presenting in the following slides is just in the concept phase – there is no code yet

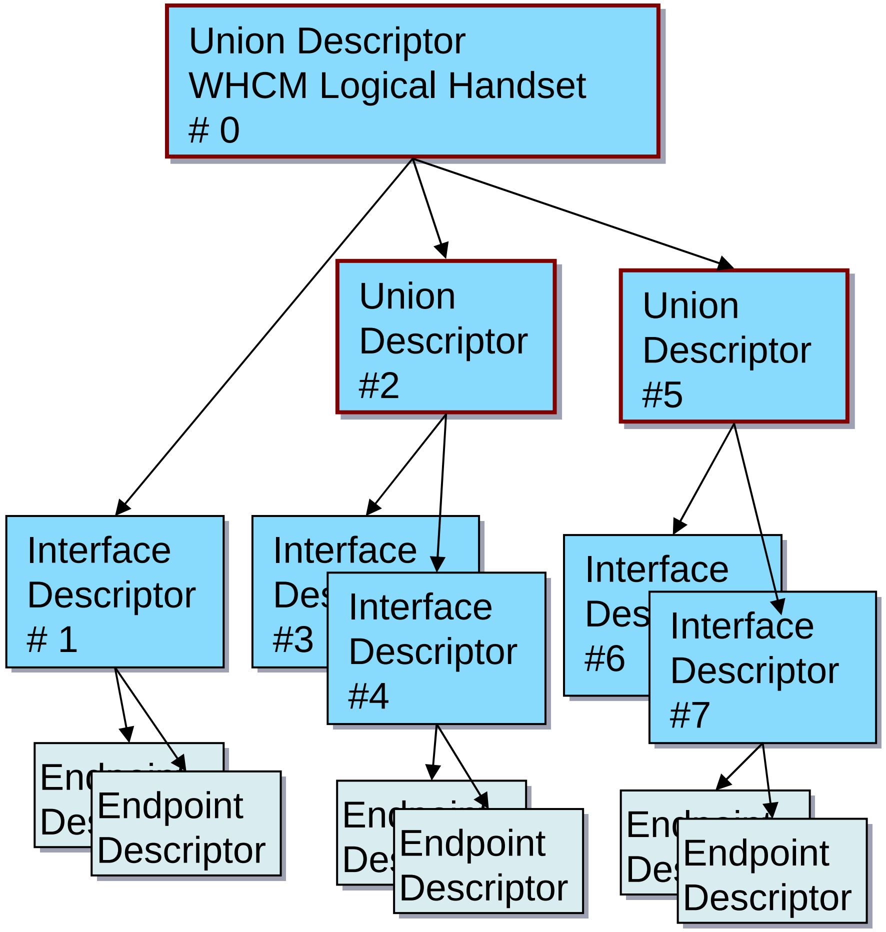
Design Sketch

- Introduce the following new entities:
  - A generic multiplexing layer
  - A product-specific component responsible for …
    - … constructing the composite device from its Gadget driver constituents
    - … branding
- Boundary conditions:
  - Keep interfaces as compatible as possible to minimise transition effort
  - Keep multiplexing layer simple so that single-function Gadgets are still simple



Solution – Union Descriptors

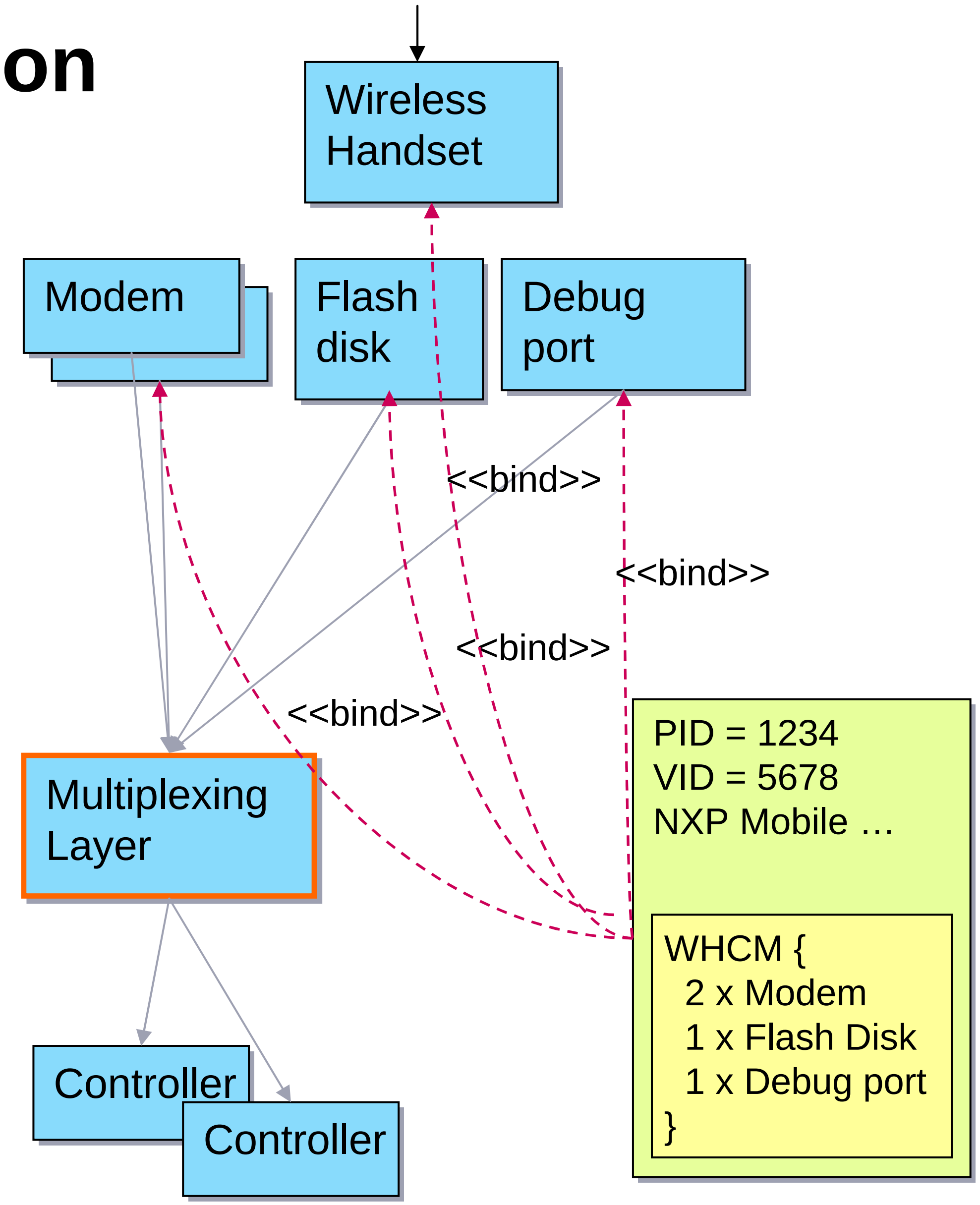
- Those descriptors that form “brackets” around interfaces contain *indices* pointing to the subordinate interfaces
- Descriptors from a tree, with descriptors as nodes and indices as arcs.
- To build the final descriptor tree:
  - Each Gadget driver defines a descriptor “template”, with indices starting at 0. (Can be used unchanged for single-function device.)
  - For each Gadget driver to be added, the Gadget driver’s template is merged in and an offset added to the indices
- Note that the merge function needs to be aware of the descriptor formats.



Descriptor tree (simplified, functional descriptors etc. left out.)

Solution – Configuration

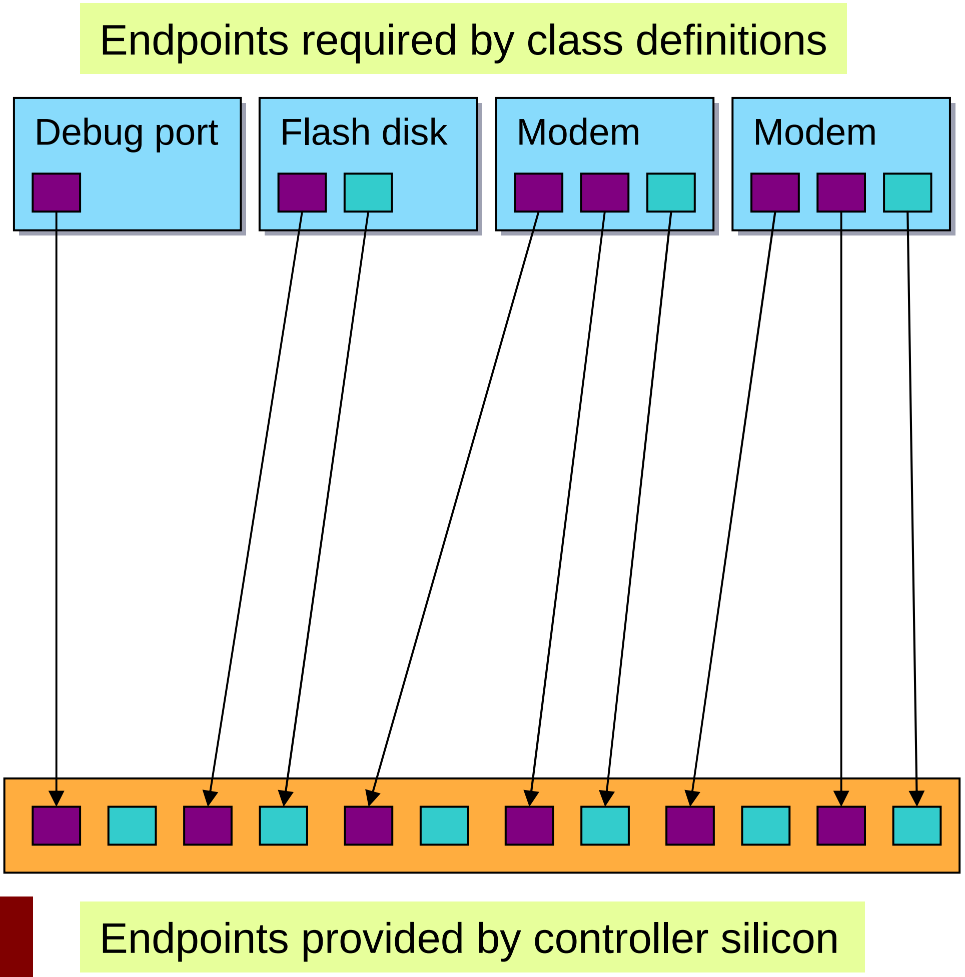
- Building the descriptors is in essence constructing a tree.
- The Gadget drivers should keep only the descriptors that characterise the Gadget driver as such: interface, endpoint, and union descriptors.
- These descriptors of a Gadget form a subtree; this subtree is either at the root directly blow the configuration descriptor (the WHCM union descriptor in the example), or is inserted further down in the tree.
- As graph operations are not a strong point of the C language, a textual representation interpreted at runtime could be used. (Gadgets have names anyway.)



Solution – Endpoint Mapping

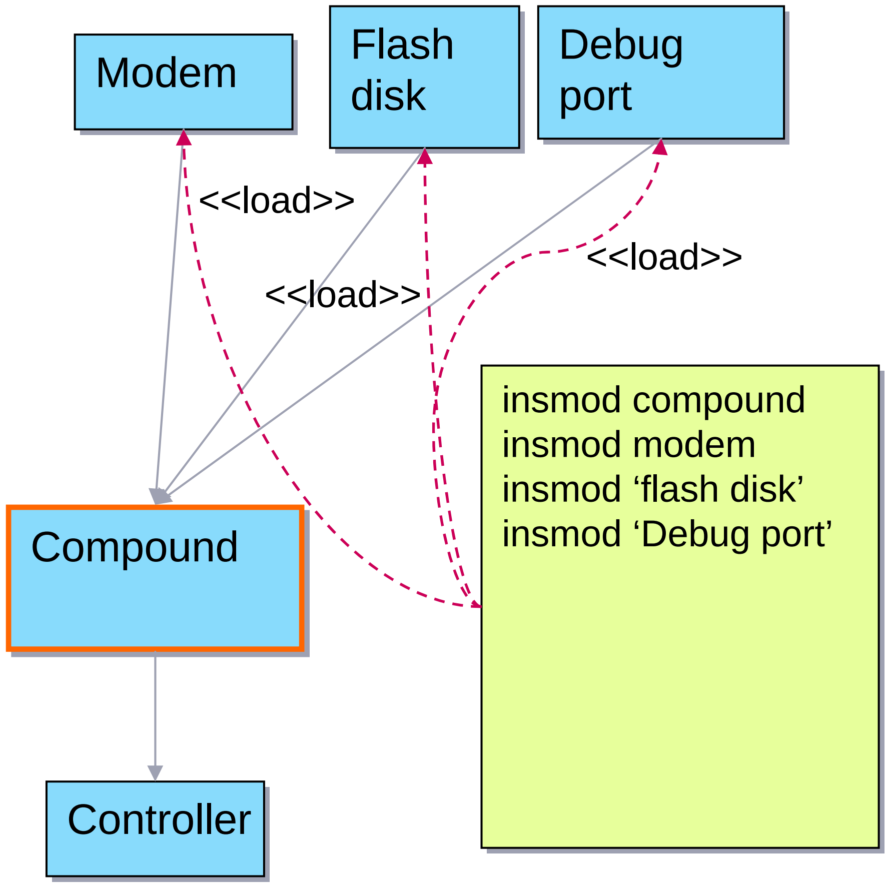
- When the Gadget is initialised, the endpoints *available in the controller* and the endpoints *required by the class definitions* have to be matched.
- Matching process is simple if controller has a large enough set of equivalent endpoints that can be configured as needed
- Sometimes complex:
  - The controller silicon has a fixed amount of RAM that can be flexibly assigned to endpoints (endpoints single/double buffered, large or small buffer)
  - Endpoints are not equivalent

If the endpoint structure of the controller cannot be sensibly formalised then let the programmer supply a mapping function.


Endpoints required by class definitions
Debug port
Flash disk
Modem
Modem
Endpoints provided by controller silicon

Related Work

- David Brownell is working on a Gadget enhancement
- His design is introducing a *compound* gadget
- A multi-function device is formed by loading the compound Gadget driver, followed by the remaining Gadget drivers.
- The Gadget drivers are parameterised by module parameters.
- The requirements are more modest than the ones presented above:
  - Only a single instance of a Gadget driver
  - Only one controller



Conclusion

Conclusions

- USB gadgets are a viable solution for simple single-function devices
- Multi-function devices – in particular the Wireless Mobile Communication devices – need enhancements to the Gadget framework
- The same is true for Wireless USB
- Ongoing work by D. Brownell addresses part of the functional gap
- NXP will work with the community on improvements

Your Feedback

- Questions?
- Are the enhancements relevant?
- To which types of products?
- Do you see further requirements or different solutions?
- Which other device classes do you expect to see?