Original Converastion https://notebooklm.google.com/notebook/73c277dd-c96f-4c61-9e79-66fa5377fa18

The Linux USB framework is a complex and layered architecture designed to manage the wide array of USB devices and their interactions with a host system
.
Understanding the USB Framework: Components and Their Functions
The Linux USB framework operates through several key components that interact via well-defined APIs:
•
usbcore: This is the central part of the Linux USB stack that acts as an intermediary between user-facing drivers and the underlying hardware controllers
.
◦
API Categorisation: usbcore APIs are broadly divided into two categories: those for general-purpose drivers and those for core drivers
.
◦
Data Transfers: It supports four fundamental types of data transfers: control, bulk, interrupt, and isochronous
. Control and bulk transfers use bandwidth as available, while interrupt and isochronous transfers are scheduled for guaranteed bandwidth
.
◦
I/O Models: usbcore provides both synchronous and asynchronous I/O models
. The asynchronous model primarily uses USB Request Blocks (URBs), where drivers submit requests and a completion callback handles the next step. Synchronous wrapper functions like usb_control_msg() and usb_bulk_msg() are available for simpler, single-buffer transfers, but cannot be used in interrupt context
.
◦
Buffer Management: Drivers need to provide DMA-suitable buffers for I/O
. usbcore offers usb_alloc_coherent() and usb_free_coherent() for allocating DMA-consistent memory, which can prevent issues like DMA bounce buffers and cache coherency problems on certain platforms
.
◦
URB Management: usbcore provides functions to create (usb_alloc_urb, usb_init_urb), initialize (usb_fill_control_urb, usb_fill_bulk_urb, usb_fill_int_urb), submit (usb_submit_urb), and cancel (usb_unlink_urb, usb_kill_urb, usb_poison_urb, usb_block_urb) URBs
. It also handles URB reference counting (usb_get_urb, usb_free_urb) and anchoring (usb_anchor_urb, usb_unanchor_urb) to manage multiple outstanding requests
.
◦
Device/Interface State Management: It includes APIs for checking pipe and endpoint validity (usb_pipe_type_check, usb_urb_ep_type_check)
, setting interface alternate settings (usb_set_interface), and managing device state (usb_set_device_state)
.
◦
Hotplugging: usbcore supports hotplugging, allowing the system to automatically detect and load/bind drivers when devices are connected

. This is primarily facilitated by the id_table within usb_driver structures
.
◦
Power Management (PM): usbcore integrates power management, supporting both system suspend (e.g., hibernate) and dynamic suspend (runtime suspend/autosuspend) for individual devices
. It manages PM states and remote wakeup capabilities
.
◦
User-space Interaction: usbcore exposes interfaces to user-space through character device nodes (traditionally /dev/bus/usb/BBB/DDD) and the /sys/kernel/debug/usb/devices file, allowing user-mode applications or drivers to interact with USB devices directly via ioctl() requests or gather debug information
.
•
Core Drivers: These drivers are an intrinsic part of the usbcore and typically interact directly with USB hardware
.
◦
Hub Driver: Manages trees of USB devices, handling device enumeration and power to downstream ports
.
◦
Host Controller Drivers (HCDs): Control individual USB buses (e.g., UHCI, OHCI, EHCI, XHCI)
. HCDs are the only host-side drivers that directly access hardware registers and handle IRQs. The Synopsys DesignWare Core SuperSpeed USB 3.0 Controller (DWC3) is an example of such a controller. HCDs provide common functionalities through a shared API, although historical differences and fault handling can vary
.
•
General-Purpose Drivers: These are host-side drivers that bind to specific interfaces on USB devices, not entire devices
.
◦
Interface Binding: A USB device can have multiple interfaces, each encapsulating a single high-level function (e.g., an audio stream, a HID control)
. General-purpose drivers typically manage one or more of these interfaces
.
◦
usb_driver Structure: They register with usbcore using the struct usb_driver structure, providing a name, probe() and disconnect() methods, and an id_table
.
◦
Hotplugging: The id_table enables hotplugging, allowing usbcore to match and bind drivers to devices automatically when they are connected
.
◦
Power Management: General-purpose drivers indicate PM support via the .supports_autosuspend flag and use functions like usb_autopm_get_interface() and usb_autopm_put_interface() to manage the device's busy/idle state and allow autosuspension
. They also define suspend, resume, and reset_resume methods for system PM events
.
◦

I/O Operations: They use URBs for asynchronous data transfers to and from device endpoints
.
•
Gadget Drivers: These are device-side drivers that run inside USB peripheral hardware that embeds Linux
.
◦
Role: In USB protocol interactions, gadget drivers act as the slave or function driver, while the host's device driver is the master
.
◦
Layered Architecture: The USB gadget stack typically involves three layers: the USB Controller Driver (lowest level, talks to hardware like DWC3
), the Gadget Driver (implements hardware-neutral USB functions, handles setup requests, configuration, and data transfers), and an Upper Level (connects to other Linux subsystems or user-space applications)
.
◦
Core Objects: They use struct usb_gadget_driver to declare themselves and interact with struct usb_gadget (representing the USB device) and struct usb_ep (representing endpoints)
.
◦
Life Cycle: Gadget drivers register with usb_gadget_register_driver_owner()
, then bind() to a usb_gadget, activating the data line pull-up so the host can detect the device. They handle enumeration steps like returning descriptors and setting configurations. disconnect() is called when the device is disconnected
.
•
USB Type-C Connector Class: This is a kernel class designed to expose USB Type-C port capabilities and control to user space in a unified manner
.
◦
Components: It represents Type-C ports, connected partners, and cable plugs as devices under /sys/class/typec/
.
◦
Alternate Modes: It supports Alternate Modes, which require communication using Vendor Defined Messages (VDM)
. Each alternate mode has a unique SVID (Standard or Vendor ID) and mode number. Alternate Mode drivers bind to partner alternate mode devices, while port drivers handle port alternate mode devices. They use typec_altmode_vdm() for SVID-specific communication. If pin reconfiguration is needed, typec_altmode_notify() is used to inform the bus
.
◦
Power Delivery: The class allows user space control over roles and alternate modes
, and port drivers can report power role changes, data role changes, and VCONN source changes via specific APIs
.
USB OTG Specification
USB On-The-Go (OTG) is a specification that allows a single USB port on a device to function as either a host or a peripheral (referred to as Dual-Role operation)

. Traditionally, USB systems have a clear master-slave asymmetry, with a host always being the master
. OTG introduces flexibility, enabling devices like smartphones to connect to a PC as a peripheral or to a USB stick as a host.
What extra things does USB OTG support? OTG introduces two new protocols that enhance its dual-role capabilities and power efficiency:
•
Host Negotiation Protocol (HNP): This protocol allows the device and host to swap roles during USB suspend processing
. For example, a phone acting as a host could hand over the host role to a printer, becoming a peripheral, to conserve power
.
•
Session Request Protocol (SRP): This is a more battery-friendly version of a device wakeup protocol, allowing a suspended device to request that the host resume the USB bus
. It's analogous to "Wake On LAN" but for USB
.
How does a driver developer handle that? Driver developers need to implement specific behaviors and use usbcore APIs to support OTG:
•
Gadget Driver Awareness: Gadget drivers that are OTG-capable must check the is_otg flag of their usb_gadget structure. If is_otg is true, the gadget driver must include an OTG descriptor in each of its configurations during enumeration
.
•
HNP Reporting: During the SET_CONFIGURATION request, OTG device feature flags like b_hnp_enable, a_hnp_support, and a_alt_hnp_support are updated, and these capabilities might need to be reported through a user interface
.
•
HNP Invocation: Gadget drivers may have the option to invoke HNP during some suspend callbacks
.
•
SRP Semantics: SRP changes the semantics of usb_gadget_wakeup slightly, allowing a user-initiated wakeup
.
•
Host-Side Interaction: On the host side, USB device drivers need to be taught to trigger HNP at appropriate moments, using usb_suspend_device()
. This also helps conserve battery power even for non-OTG configurations
.
•
Targeted Peripheral List (TPL): Host-side OTG implementations must support a Targeted Peripheral List, which acts as a whitelist to reject unsupported peripherals. This whitelist is product-specific
.
•
OTG Controller Driver: Beneath the generic usb_bus and usb_gadget interfaces, a dedicated OTG Controller Driver manages the OTG transceiver and state machine. This driver activates and deactivates USB controllers based on the current role (host or peripheral)
. For instance, the JZ4740 USB Device Controller (UDC) is noted as not OTG compatible, with its multipoint member set to 0

.
Suggestions, Methods, and Important Things to Develop a USB Device Driver from Scratch (Host-Side)
Developing a USB device driver from scratch requires a thorough understanding of the USB protocol and the Linux USB API. Here are key suggestions, methods, and important considerations:
1.
Understand the USB Protocol Specification:
◦
Familiarity with USB 2.0/3.x specifications: This is crucial for understanding device descriptors, configurations, interfaces, endpoints, and transfer types
. The include/uapi/linux/usb/ch9.h file contains standard USB data types
.
◦
Device Model: Recognize that USB device drivers typically bind to interfaces, not entire devices. An interface represents a single function of a device
.
2.
Utilise the usb_driver Structure:
◦
Declaration: All Linux USB drivers must register themselves using the struct usb_driver
.
◦
Mandatory Fields:
▪
.name: A unique string identifying your driver
.
▪
.probe(): Called when a device matching your id_table is detected. This is where you initialize the device
.
▪
.disconnect(): Called when the device is removed or the module is unloaded. This is where you clean up resources
.
▪
.id_table: A struct usb_device_id array that describes the devices your driver supports. This is essential for hotplugging
. Use macros like USB_DEVICE, USB_INTERFACE_INFO, USB_DEVICE_INFO to populate it
.
◦
Optional Fields:
▪
.unlocked_ioctl: For user-space communication via usbfs
.
▪
Power Management callbacks (.suspend, .resume, .reset_resume): For managing device power states
.
▪
Device Reset callbacks (.pre_reset, .post_reset, .shutdown): For handling device resets
.
▪

.supports_autosuspend: Flag to enable autosuspension for interfaces bound to your driver
.
3.
Registration and Deregistration:
◦
usb_register_driver(): Call this in your module's init function to register your usb_driver
.
◦
usb_deregister(): Call this in your module's exit function to unregister your driver
.
◦
MODULE_DEVICE_TABLE(): Export your id_table using this macro to enable automatic driver loading by hotplug utilities
.
4.
Implementing probe() and disconnect():
◦
probe() considerations:
▪
Return 0 on success, or a negative error code (e.g., -ENODEV, -ENOMEM) if you cannot or will not manage the device
.
▪
Use usb_set_intfdata() to associate a private data structure with the usb_interface for your driver's state
.
▪
You can perform I/O to the interface and endpoint 0 during probe()
.
▪
If necessary, use usb_set_interface() to select a different alternate setting for the interface
.
◦
disconnect() considerations:
▪
This callback signals that the interface is no longer accessible
.
▪
Crucially, all outstanding URBs must be cancelled or completed before disconnect() returns
. Even if usbcore kills URBs on physical disconnection, your driver must be robust against failing I/O requests before the disconnect() is called
.
▪
Free any private data and allocated buffers
.
5.
Handling Data Transfers with URBs:
◦
Asynchronous is primary: The most robust and common way to perform I/O is using URBs
.
◦
Allocation and Initialization:

▪
Allocate URBs with usb_alloc_urb()
. Pass 0 for non-isochronous transfers
.
▪
Initialize URBs using helper functions like usb_fill_control_urb(), usb_fill_bulk_urb(), or usb_fill_int_urb()
.
▪
Ensure transfer_buffer is DMA-suitable (e.g., allocated with kmalloc() or usb_alloc_coherent())
.
◦
Submission:
▪
Submit URBs using usb_submit_urb()
. This call returns immediately, queueing the request
.
▪
mem_flags (e.g., GFP_KERNEL, GFP_ATOMIC) are important for memory allocation context
.
◦
Completion Handlers:
▪
The complete callback (type usb_complete_t) is invoked when an URB finishes
.
▪
Do NOT sleep in a completion handler
. They often run in atomic context
.
▪
Check urb->status for transfer success or error
. actual_length indicates bytes transferred
.
◦
Cancellation:
▪
usb_unlink_urb(): Asynchronously cancels an URB. The completion handler will be called later
.
▪
usb_kill_urb(): Synchronously cancels an URB and waits for its completion handler to finish. Guarantees the URB is idle
.
▪
Reference Counting for Safety: When cancelling an URB that might be freed by its completion handler, use usb_get_urb() before cancelling and usb_free_urb() after to prevent race conditions
.
◦
Synchronous Wrappers: usb_control_msg(), usb_bulk_msg(), usb_interrupt_msg() offer synchronous behavior for single transfers. While simpler, they cannot be called from interrupt context and are less flexible for complex I/O patterns
.
◦

Scatter/Gather (SG) I/O: For multi-buffer transfers, usb_sg_init(), usb_sg_wait(), and usb_sg_cancel() can be used to handle scatterlists
.
6.
Power Management Implementation:
◦
Driver PM Methods: Implement suspend, resume, and reset_resume in your usb_driver
.
◦
Autosuspension: Set the supports_autosuspend flag in usb_driver
.
◦
Usage Counters: Use usb_autopm_get_interface() to increment an interface's usage counter when it becomes busy (preventing autosuspend) and usb_autopm_put_interface() to decrement it when idle (allowing autosuspend)
. Use _async versions for atomic contexts
.
◦
Remote Wakeup: If your device supports remote wakeup and your driver requires it, set intf->needs_remote_wakeup = 1
.
7.
Device Reset Handling:
◦
Implement pre_reset() and post_reset() callbacks in usb_driver. pre_reset() is for ceasing I/O and saving state, post_reset() for restoring state and resuming I/O
.
◦
If you need to trigger a reset, use usb_reset_device() (requires device lock)
or usb_queue_reset_device() for atomic contexts
.
8.
Reference Counting:
◦
Drivers should use usb_get_intf() when they bind to an interface in probe() to increment its reference count and usb_put_intf() in disconnect() to release it
.
9.
Debugging and Testing:
◦
Disconnect Testing: Always test your driver by physically disconnecting the device while it's active with various host controller drivers (HCDs) to ensure proper fault handling
.
◦
USB Skeleton Driver: Review drivers/usb/usb-skeleton.c in the kernel source as a template and guide
.
◦
Logging: Use dev_err, pr_err for error messages
.
◦

DebugFS and Trace Events: For more in-depth debugging, the kernel provides DebugFS (/sys/kernel/debug/usb/devices)
and Trace Events (e.g., for DWC3 controllers)
to inspect device status, register dumps, and URB/endpoint lifecycles.
◦
USB Sniffer: While not strictly required, a USB sniffer can be helpful for understanding on-the-wire protocol interactions
.
By following these guidelines and thoroughly studying the provided Linux kernel documentation, a developer can build a robust and functional USB device driver.