1. Introduction

Un driver (pilote) est un composant logiciel qui fait l'interface entre le noyau du système d'exploitation et un périphérique matériel ou une couche d'abstraction. Il traduit les demandes génériques du système (lire/écrire, configurer, envoyer/recevoir paquets, etc.) en opérations spécifiques au matériel, et inversement expose l'état et les événements matériels au système.

Principales responsabilités

- Initialisation et énumération : détecter le matériel, allouer ressources et configurer le périphérique.
- Traduction d'appels : implémenter les callbacks et API attendus par le sous-système du noyau (ex. net_device ops, block ops, file ops) pour que le reste du système utilise le périphérique de façon standardisée.
- Gestion des I/O : orchestrer transferts (DMA, URB, interruptions), mise en file d'attente, copies de buffers et synchronisation.
- Gestion d'erreurs et robustesse : détecter conditions d'erreur, reprendre, nettoyer proprement à la déconnexion ou en cas d'échec.
- Exposition des fonctionnalités avancées : support d'options matérielles (offloads, mise en veille, statistiques, configurations spécifiques).
- Concurrence et sécurité : protéger les ressources partagées (mutex, spinlock), respecter les contextes d'exécution (process/context IRQ/softirq).

Types et niveaux

- Pilotes noyau (in-kernel) : s'exécutent au sein du noyau, offrent faibles latences et accès direct au matériel (ex. pilotes réseaux, disques, USB).
- Pilotes en espace utilisateur : utilisent un mécanisme de médiation (ex. UIO, libusb) et conviennent pour prototypage ou usages moins critiques.
- Pilotes de classe vs vendor-specific : les pilotes de classe implémentent une interface standard (ex. CDC, HID) interopérable ; les pilotes vendor-specific gèrent fonctionnalités propriétaires.

Propriétés attendues

- Stabilité et sécurité : ne pas corrompre la mémoire noyau, gérer correctement les erreurs et concurrents.
- Performance : minimiser copies, utiliser offloads matériels, gérer pools/URB/queues pour hauts débits.
- Portabilité et intégration : respecter les conventions du sous-système kernel concerné (APIs, structures, callbacks, sysfs/ethtool si applicable).

2. Présentation CH397

Le CH397 est une puce NIC USB hautement intégrée et basse consommation, destinée à étendre une interface Ethernet via USB pour ordinateurs de bureau, portables, tablettes, consoles de jeu, etc. Elle intègre un processeur RISC-V QingKe, un contrôleur USB2.0/2.1 avec transceiver PHY conforme USB2.1, ainsi

qu'un sous-système Ethernet complet (MAC + PHY) conforme IEEE 802.3, supportant 10/100 Mbps.

Architecture matérielle

- **SoC tout-en-un :** processeur embarqué QingKe RISC-V, contrôleur USB, transceiver PHY USB et contrôleur Ethernet MAC+PHY intégrés sur une unique puce (packages QFN24/QFN32).
- **USB 2.0/2.1 Full/High-Speed :** contrôleur et transceiver conformes à la spécification USB2.1, avec LDO intégré et composants périphériques réduits (résistance 50, condensateurs oscillateur).
- **Ethernet 10/100M :** MAC et PHY conformes IEEE 802.3 (10BASE-T / 100BASE-TX) ; support de l'auto-négociation 10/100 Mbps, Auto-MDIX et distances jusqu'à 120 m sur câbles CAT5/CAT6.
- **Mémoire tampon interne :** buffers TX/RX intégrés pour stocker les trames avant transfert sur USB, réduisant la charge CPU de l'hôte.
- **Fonctions matérielles supplémentaires :** LDO interne, support ESD amélioré (6 kV), configuration LED matérielle, wake-up réseau et modes basse consommation.

Interfaces réseau disponibles

- **CDC-ECM, CDC-NCM et RNDIS :** support natif de CDC-ECM et CDC-NCM (USB Ethernet classes) et RNDIS — possibilité d'utilisation sans pilote propriétaire sur de nombreux hôtes, ou avec pilote vendor-specific si souhaité.
- **Endpoints USB standard :** modes vendor-specific (bulk endpoints) et classes réseau standard (interfaces Communications/Data avec alt-setting pour endpoints bulk IN/OUT).
- **Fonctionnalités réseau hardware-assist :**
  - génération et vérification de checksums IPv4/IPv6 (TCP/UDP/HEAD),
  - support IEEE 802.3x flow control et fallback half-duplex,
  - prise en charge VLAN (802.3Q),
  - wake-on-LAN (magic packet / wake packets).

Contraintes et exigences spécifiques

- **Choix de configuration USB :** la puce expose plusieurs configurations (vendor, CDC-ECM, NCM, RNDIS) — le pilote doit détecter la configuration active et sélectionner l'interface/altsetting offrant les endpoints bulk nécessaires (ex. altsetting 1 pour CDC Data).
- **Taille et gestion des buffers :** présence de buffers internes signifie que le périphérique peut livrer des trames agrégées ou de taille > MTU standard ; prévoir côté pilote des buffers hôtes suffisamment grands (ex. >= 1600–2048 octets) et gérer l'alignement skb.
- **Performances et flux :** le CH397 prend en charge features hardware-assist (checksum offload, flow control, VLAN), le pilote peut en tirer parti pour réduire la charge CPU et améliorer le débit ; envisager gestion des URB/queues pour atteindre des débits proches du 100 Mbps.

- **Gestion basse-consommation et wake-up :** la puce propose modes Sleep et wake-on-LAN — le pilote doit exposer et coordonner ces capacités via PM (suspend/resume, autosuspend) et notifications réseau.
- **Robustesse physique et contraintes matérielles :** 6 kV ESD, exigences d'oscillateur externe et simple filtrage passif — le design matériel autour du CH397 est simplifié, mais le firmware peut exposer ou non certaines fonctionnalités (NCM, paramètres LED).
- **Compatibilité multi-OS :** grâce au support CDC-ECM/NCM/RNDIS, le CH397 peut fonctionner sans pilote propriétaire sur de nombreux systèmes ; toutefois, un pilote dédié permet d'exploiter au mieux les optimisations matérielles et les modes vendor-specific.

En synthèse, le CH397 fournit une solution USB→Ethernet complète et optimisée, offrant à la fois compatibilité plug-and-play via classes USB standard et options avancées (offloads, VLAN, wake-up, basse consommation) pour une intégration performante sur hôte via un pilote USB-Ethernet adapté.

4. Conception

4.1 Architecture logicielle (modules et interfaces)

- **Composants principaux**
  - Module USB driver linux : probe, disconnect, suspend, resume.
  - Couche réseau (net_device): instance net_device créée via alloc_etherdev, opérateurs netdev (open, stop, start_xmit, validate_addr, set_mac).
  - Gestion des transferts USB: URB pour RX (dev->rx_urb) et URB temporaires pour TX.
  - Buffering utilisateurs: dev->bulk_in_buffer pour réception, skb pour transmission/réception vers/depuis la pile réseau.
- **Interactions**
  - usb core <-> driver : probe/ disconnect, usb_submit_urb / usb_kill_urb.
  - driver <-> network stack : netif_rx, netdev_ops callbacks, register_netdev/unregister_netdev.
  - Synchronisation locale : mutex io_mutex pour protéger opérations I/O critiques (prise/relâche autour d'opérations longues).

4.2 API exposée au noyau (structures, callbacks)

- Structures principales :
  - struct ch397_device : état du périphérique, pointeurs usb/netdev, endpoints, buffer et URB.
  - struct net_device : structure standard exposée au stack réseau, initialisée via alloc_etherdev.
- Callbacks et points d'entrée :

  - usb driver callbacks : ch397_probe, ch397_disconnect, ch397_suspend, ch397_resume.
  - net_device ops : ch397_net_open (ndo_open), ch397_net_stop (ndo_stop), ch397_start_xmit (ndo_start_xmit), eth_validate_addr (ndo_validate_addr), eth_mac_addr (ndo_set_mac_address).
  - URB callbacks : ch397_read_callback (RX), ch397_write_callback (TX).
- Comportement attendu :
  - Lors de probe : allouer netdev, initialiser ch397_device, trouver endpoints, récupérer MAC, register_netdev.
  - Lors de open : lancer réception (soumettre URB RX), mettre la carrier on et démarrer la queue.
  - Lors de start_xmit : encapsuler skb dans URB bulk-out, soumettre, gérer compteurs et erreurs.
  - Lors de disconnect : arrêter RX, unregister_netdev, libérer ressources.

4.3 Intégration avec etherdevice.h

- Utilisation :
  - alloc_etherdev(sizeof(struct ch397_device)) pour allocation et initialisation d'un net_device avec espace privé.
  - eth_random_addr/mac helpers : eth_random_addr utilisé comme fallback si lecture MAC échoue ; eth_hw_addr_set pour positionner l'adresse MAC.
  - eth_validate_addr/eth_mac_addr fournis comme ndo_validate_addr/ndo_set_mac_address, ce qui délègue la validation et le réglage standard d'adresse MAC.
- Conséquences :
  - Le driver s'appuie sur les helpers pour respecter les conventions Ethernet sans réimplémenter la validation/gestion MAC.
  - L'alignement des sk_buffs est fait via skb_reserve(skb, 2) pour correspondre aux exigences de la pile Ethernet/IP.

4.4 Rôle de kernel.h et interactions avec le noyau

- Inclusion de <linux/kernel.h> pour :
  - Fonctions d'impression et de logging et macros comme KERN_INFO (utilisées dans le code).
  - Types et API kernel de base (HZ, GFP_KERNEL/GFP_ATOMIC, etc.).
- Interactions principales :
  - Allocation mémoire: kmalloc/kfree, netdev/skb allocation.
  - Gestion des URB: usb_alloc_urb/usb_free_urb, usb_submit_urb, usb_kill_urb.
  - Enregistrement/espace d'identification: MODULE_DEVICE_TABLE, module_usb_driver.
- Impact : le driver utilise les primitives noyau pour mémoire, synchronisation

(mutex), temporisation (watchdog_timeo), et communication inter-sous-systèmes (network/usb).

4.5 Gestion des buffers et des descriptors

- Buffers RX :
  - Buffer statique alloué kmalloc(dev->bulk_in_size) (2048 octets) pour réception.
  - RX URB configuré pour pointer sur dev->bulk_in_buffer ; urb->actual_length indique la taille reçue.
  - Après réception, données copiées dans un skb via netdev_alloc_skb + skb_put_data, puis remises au stack réseau.
- Buffers TX :
  - Pas de buffer de copie : l'URB de transmission référence directement skb->data (risque si le stack modifie ou libère le skb) — dans ce code, le skb est passé comme contexte de l'URB et libéré dans write_callback.
  - URB allouées par transmission et libérées dans ch397_write_callback.
- Descriptors / URB lifecycle :
  - RX : un URB long-vie (dev->rx_urb) est alloué lors de ch397_start_rx, soumis, et resoumis par le callback. En stop, usb_kill_urb + usb_free_urb.
  - TX : URB allouées par start_xmit, soumises, et libérées dans le callback de fin.
- Statistiques et erreurs :
  - netdev->stats.{rx_packets, rx_bytes, tx_packets, tx_bytes, rx_errors, tx_errors, rx_dropped, tx_dropped} mis à jour aux emplacements appropriés.
- Remarques sur robustesse :
  - Vérifier et gérer le cas où usb_submit_urb échoue (déjà partiellement géré).
  - Potentiel reusage / pool d'URB pour haute performance non implémenté ; actuellement allocation à la volée.
  - Attention à l'utilisation de skb->data directement : garantir que le skb n'est pas modifié tant que l'URB est en vol (ici le skb est conservé comme contexte et libéré dans le callback, ce qui est acceptable).

5. Implémentation

- Le code fournit une base fonctionnelle pour un driver USB Ethernet CDC-like : découverte endpoints, RX continu via URB ré-submis, TX via URB à la demande, intégration basique avec la stack réseau.

5.1 Stratégie d'initialisation et d'args parsing

- Initialisation principale dans ch397_probe :

  - Vérifier configuration/interface USB et ignorer l'interface CDC Comm quand nécessaire.
  - Allouer net_device via alloc_etherdev et récupérer ch397_device avec netdev_priv.
  - Initialiser mutex, obtenir référence usb_get_dev, définir altsetting si nécessaire (usb_set_interface pour CDC Data).
  - Parcourir endpoints de l'altsetting courant pour découvrir bulk-in et bulk-out ; allouer bulk_in_buffer (kmalloc).
  - Récupérer l'adresse MAC via ch397_get_mac_address (usb_string) et l'appliquer via eth_hw_addr_set.
  - Initialiser netdev->netdev_ops, watchdog_timeo, et appeler register_netdev.
  - En cas d'erreur, faire cleanup (usb_put_dev, kfree buffer, free_netdev).
- Pas d'args parsing via module_param dans ce code — stratégie : valeurs par défaut et auto-détection depuis l'USB device string/config.

5.2 Routine d'attach/detach

- Attach (probe) : allouer structures, configurer endpoints, set MAC, register_netdev, netif_carrier_off (attente de l'open).
- Detach (disconnect) :
  - Récupérer dev via usb_get_intfdata, retirer association usb_set_intfdata(NULL).
  - unregister_netdev (qui arrêtera la queue si nécessaire).
  - Arrêter RX via ch397_stop_rx (usb_kill_urb + usb_free_urb).
  - Libérer bulk_in_buffer, usb_put_dev, free_netdev.
  - Logging de déconnexion.
- Robustesse :
  - Garantir que disconnect peut être appelé même si probe a échoué partiellement (vérifier NULLs avant free).
  - Utiliser usb_kill_urb pour attendre la fin des callbacks avant de free les ressources partagées.

5.3 Transmissions (TX) : file d'attente et gestion

- Flux d'envoi :
  - ch397_start_xmit est appelé avec un skb.
  - Allouer une URB (usb_alloc_urb) ; si échec, libérer skb et incrémenter tx_dropped.
  - Préparer URB avec usb_fill_bulk_urb en pointant sur skb->data et en passant skb comme contexte.
  - Soumettre URB (usb_submit_urb). En cas d'échec : libérer urb et skb, incrémenter tx_errors.
  - Si succès : skb conservé jusqu'au ch397_write_callback qui libère le skb et l'URB.

- Gestion de la queue :
  - Implémentation actuelle ne stoppe pas la queue quand trop d'URB en vol (commentaire présent). Pour production : ajouter compteur d'URB en vol et netif_stop_queue/netif_wake_queue.
- Sécurité mémoire :
  - Le skb est référencé via urb->context et libéré dans le callback ; s'assurer que rien ne libère ou modifie skb avant callback.
- Erreurs et métriques :
  - Incrémenter netdev->stats.tx_errors sur échec de submit, tx_packets/tx_bytes sur succès (dans callback).

5.4 Réception (RX) : traitement des paquets

- Setup :
  - ch397_start_rx alloue dev->rx_urb, le configure via usb_fill_bulk_urb sur bulk_in_buffer et soumet.
- Traitement dans ch397_read_callback :
  - Vérifier urb->status ; si success et actual_length > 0 :
    - hexdump pour debug.
    - Allouer skb via netdev_alloc_skb, réserver 2 octets pour alignement, copier données avec skb_put_data.
    - Déterminer protocole via eth_type_trans et injecter via netif_rx.
    - Mettre à jour stats rx_packets/rx_bytes.
  - Sur erreurs terminales (-ENOENT, -ECONNRESET, -ESHUTDOWN) : retourner (pas de resubmit).
  - Sur autres erreurs : incrémenter rx_errors.
  - Resoumettre l'URB via usb_submit_urb(urb, GFP_ATOMIC) pour continuer la réception.
- Robustesse :
  - Gestion d'allocation skb échouée : incrémenter rx_dropped.
  - Toujours resoumettre URB tant que l'interface reste active ; gérer échecs de resubmit (logger, incrémenter erreur).
  - S'assurer que le buffer dev->bulk_in_buffer reste alloué pendant la durée de vie de l'URB (libéré seulement à disconnect).

5.5 Gestion des erreurs et reprise

- Erreurs USB :
  - Dans read/write callbacks, vérifier urb->status et traiter codes connus.
  - Echec de usb_submit_urb lors d'initialisation : nettoyage et retour d'erreur dans probe/start_rx.
- Reprise :
  - Probe refuse config non supportées (config 3, interface 0 en config 2) évitant états incohérents.
  - On peut améliorer la reprise en ajoutant tentatives limitées de resubmit et backoff si submit échoue.

- Nettoyage : toujours free les URB (usb_free_urb) et buffers si submit échoue.
- Consistance :
  - Utiliser usb_kill_urb dans disconnect pour garantir callbacks terminés avant free.
  - mutex io_mutex disponible pour séquencer opérations critiques sur l'USB (bien que son usage soit minimal actuellement).

5.6 Synchronisation et verrouillage

- Mutex :
  - mutex_init(&dev->io_mutex) lors du probe.
  - mutex utilisé dans suspend pour verrouiller les opérations I/O (exemple minimal présent). Recommandation : entourer submit/kill URB et manipulations dev->rx_urb/dev->bulk_in_buffer par la mutex si nécessaire.
- Contexte d'exécution :
  - Callbacks URB s'exécutent en contexte interruption ou softirq ; utiliser GFP_ATOMIC pour allocations faites dans callbacks si nécessaire.
  - Les fonctions net_device ops (ndo_start_xmit, ndo_open/stop) s'exécutent en context process ; utiliser GFP_KERNEL pour allocations là.
- Protection des ressources :
  - Accès concurrent à rx_urb, bulk_in_buffer et autres champs doivent être protégés lors de modification (start/stop Rx, disconnect).
  - Lors du disconnect, utiliser usb_set_intfdata(interface, NULL) avant de libérer afin d'éviter race avec callbacks.

5.7 Extraits de code clés (fonctions importantes)

- ch397_probe : allocation netdev, découverte endpoints, set_interface altsetting, récupération MAC, register_netdev, gestion d'erreurs.
- ch397_start_rx / ch397_stop_rx : allocation/soumission et arrêt/libération de RX URB.
- ch397_read_callback : traitement packet -> skb -> netif_rx, resubmit URB.
- ch397_start_xmit / ch397_write_callback : envoi via URB, statistiques, libération skb/urb.
- ch397_disconnect : cleanup complet — unregister_netdev, usb_kill_urb, free buffers/structures.