* Khi thiết bị cắm vào thì làm sao driver nhận biết được thiết bị và khởi tạo
* Làm thế nào để khởi tạo file device node
* Sau khi tạo xong device node thì sẽ có hành động gì diễn ra

không cần interrupt vì chỉ có 1 thiết bị kết nối với PC trong 1 thời điểm

~~throttled, unthrottled ???~~

void usb\_set\_intfdata(struct usb\_interface \*intf, void \*data);

void \*usb\_get\_intfdata(struct usb\_interface \*intf);

Functions to set and get access to the private data pointer section within the struct usb\_interface.

struct urb;

Structure that describes a USB data transmission.

struct urb \*usb\_alloc\_urb(int iso\_packets, int mem\_flags);

void usb\_free\_urb(struct urb \*urb);

Functions used to create and destroy a struct usb urb \*.

int usb\_submit\_urb(struct urb \*urb, int mem\_flags);

int usb\_kill\_urb(struct urb \*urb);

int usb\_unlink\_urb(struct urb \*urb);

Functions used to start and stop a USB data transmission.

Linux OS has a way of identifying device files via **major device numbers**, which identify modules serving device files or a group of devices it servers, and **minor device numbers**, which identify a specific device from a group of devices, which the major device number specifies.

The CDC is how communication devices interface to the USB

ACM là lớp con của CDC (communications device class)

1. Look up an ACM structure by minor. If found and not disconected,

Increment

1. Find available minor number, if found, associate it with acm
2. Release the minor number associated with acm
3. ACM control messages ( **functions**)
4. Write buffer management
5. Finish write. Caller must hold **acm->write\_lock**
6. ~~poke write. The caller is responsible for locking~~
7. data interface wrote those outgoing bytes
8. TTY handlers (tty\_**install, tty\_open, port\_activate, port\_destruct,port\_shutdown, tty\_cleanup, tty\_hangup, tty\_close, tty\_write,)**

Get\_serial\_info

Set\_serial\_info

Wait\_serial\_change

~~Acm\_tty\_~~**~~ioctl~~**

**Acm\_tty\_set\_termios**

1. USB probe and disconnect routines

//read/write buffer free

Write\_buffers\_free

Read\_buffer\_free

//write buffers allocate

Wite\_buffers\_alloc

Acm\_probe

….

Acm\_disconnect

1. USB structure

Nokia blabla

1. TTY driver structures

.install

.open

.close

….

1. Init / exit

\_\_init acm\_init

\_\_exit acm\_exit

**Module\_init(acm\_init)**

**Module\_exit(acm\_exit)**

static struct acm \*acm\_get\_by\_minor(unsigned int minor)

{

struct acm \*acm;

mutex\_lock(&acm\_minors\_lock);

acm = idr\_find(&acm\_minors, minor); //return pointer for given minor

if (acm) {

mutex\_lock(&acm->mutex);

if (acm->disconnected) {

mutex\_unlock(&acm->mutex);

acm = NULL;

} else {

tty\_port\_get(&acm->port);

mutex\_unlock(&acm->mutex);

}

}

mutex\_unlock(&acm\_minors\_lock);

return acm;



static int acm\_alloc\_minor(struct acm \*acm)

{

int minor;

mutex\_lock(&acm\_minors\_lock);

minor = idr\_alloc(&acm\_minors, acm, 0, ACM\_TTY\_MINORS, GFP\_KERNEL);

mutex\_unlock(&acm\_minors\_lock);

return minor;

}



static void acm\_release\_minor(struct acm \*acm)

{

mutex\_lock(&acm\_minors\_lock);

idr\_remove(&acm\_minors, acm->minor); //clear the entry for acm->minor

mutex\_unlock(&acm\_minors\_lock);

}



static int acm\_ctrl\_msg(struct acm \*acm, int request, int value,

void \*buf, int len)

{

int retval;

retval = usb\_autopm\_get\_interface(acm->control);

if (retval)

return retval;

retval = usb\_control\_msg(acm->dev, usb\_sndctrlpipe(acm->dev, 0),

request, USB\_RT\_ACM, value,

acm->control->altsetting[0].desc.bInterfaceNumber,

buf, len, 5000);

usb\_autopm\_put\_interface(acm->control);

return retval < 0 ? retval : 0;

}



static int acm\_wb\_alloc(struct acm \*acm)

{

int i, wbn;

struct acm\_wb \*wb;

wbn = 0;

i = 0;

for (;;) {

wb = &acm->wb[wbn];

if (!wb->use) {

wb->use = 1;

return wbn;

}

wbn = (wbn + 1) % ACM\_NW;

if (++i >= ACM\_NW)

return -1;

}

}

static int acm\_wb\_is\_avail(struct acm \*acm)

{

int i, n;

unsigned long flags;

n = ACM\_NW;

spin\_lock\_irqsave(&acm->write\_lock, flags);

for (i = 0; i < ACM\_NW; i++)

n -= acm->wb[i].use;

spin\_unlock\_irqrestore(&acm->write\_lock, flags);

return n;

}



static void acm\_write\_done(struct acm \*acm, struct acm\_wb \*wb)

{

wb->use = 0;

acm->transmitting--;

usb\_autopm\_put\_interface\_async(acm->control);

}

1. ~~Bỏ~~

static int acm\_start\_wb(struct acm \*acm, struct acm\_wb \*wb)

{

int rc;

acm->transmitting++;

wb->urb->transfer\_buffer = wb->buf;

wb->urb->transfer\_dma = wb->dmah;

wb->urb->transfer\_buffer\_length = wb->len;

wb->urb->dev = acm->dev;

rc = usb\_submit\_urb(wb->urb, GFP\_ATOMIC);

if (rc < 0) {

dev\_err(&acm->data->dev,

"%s - usb\_submit\_urb(write bulk) failed: %d\n",

\_\_func\_\_, rc);

acm\_write\_done(acm, wb);

}

return rc;

}



static void acm\_write\_bulk(struct urb \*urb)

{

struct acm\_wb \*wb = urb->context;

struct acm \*acm = wb->instance;

unsigned long flags;

int status = urb->status;

if (status || (urb->actual\_length != urb->transfer\_buffer\_length))

dev\_vdbg(&acm->data->dev, "wrote len %d/%d, status %d\n",

urb->actual\_length,

urb->transfer\_buffer\_length,

status);

spin\_lock\_irqsave(&acm->write\_lock, flags); // disables interrupts before taking the spinlock

acm\_write\_done(acm, wb);

spin\_unlock\_irqrestore(&acm->write\_lock, flags);

set\_bit(EVENT\_TTY\_WAKEUP, &acm->flags);

schedule\_work(&acm->work);

}

static void acm\_softint(struct work\_struct \*work)

{

int i;

struct acm \*acm = container\_of(work, struct acm, work);

if (test\_bit(EVENT\_RX\_STALL, &acm->flags)) {// determine whether a bit is set (EVENT\_RX\_STALL: bit number to test)

if (!(usb\_autopm\_get\_interface(acm->data))) {// increment a USB interface’s PM-usage counter)

for (i = 0; i < acm->rx\_buflimit; i++)

usb\_kill\_urb(acm->read\_urbs[i]);

usb\_clear\_halt(acm->dev, acm->in);

acm\_submit\_read\_urbs(acm, GFP\_KERNEL);

usb\_autopm\_put\_interface(acm->data);

}

clear\_bit(EVENT\_RX\_STALL, &acm->flags);

}

if (test\_bit(EVENT\_TTY\_WAKEUP, &acm->flags)) {

tty\_port\_tty\_wakeup(&acm->port);

clear\_bit(EVENT\_TTY\_WAKEUP, &acm->flags);

}

}



static int acm\_tty\_install(struct tty\_driver \*driver, struct tty\_struct \*tty)

{

struct acm \*acm;

int retval;

acm = acm\_get\_by\_minor(tty->index);

if (!acm)

return -ENODEV; // no such device

retval = tty\_standard\_install(driver, tty);// same as tty\_port\_install

if (retval)

goto error\_init\_termios;

tty->driver\_data = acm;

return 0;

error\_init\_termios:

tty\_port\_put(&acm->port);

return retval;

}

static int acm\_tty\_open(struct tty\_struct \*tty, struct file \*filp)

{

struct acm \*acm = tty->driver\_data;

return tty\_port\_open(&acm->port, tty, filp);

}

static void acm\_port\_dtr\_rts(struct tty\_port \*port, int raise)

{

struct acm \*acm = container\_of(port, struct acm, port);

int val;

int res;

if (raise)

val = ACM\_CTRL\_DTR | ACM\_CTRL\_RTS;

else

val = 0;

/\* FIXME: add missing ctrlout locking throughout driver \*/

acm->ctrlout = val;

res = acm\_set\_control(acm, val);

if (res && (acm->ctrl\_caps & USB\_CDC\_CAP\_LINE))

dev\_err(&acm->control->dev, "failed to set dtr/rts\n");

}

static int acm\_port\_activate(struct tty\_port \*port, struct tty\_struct \*tty)

{

struct acm \*acm = container\_of(port, struct acm, port);

int retval = -ENODEV;

int i;

mutex\_lock(&acm->mutex);

if (acm->disconnected)

goto disconnected;

retval = usb\_autopm\_get\_interface(acm->control);

if (retval)

goto error\_get\_interface;

/\*

\* FIXME: Why do we need this? Allocating 64K of physically contiguous

\* memory is really nasty...

\*/

set\_bit(TTY\_NO\_WRITE\_SPLIT, &tty->flags);

acm->control->needs\_remote\_wakeup = 1;

acm->ctrlurb->dev = acm->dev;

retval = usb\_submit\_urb(acm->ctrlurb, GFP\_KERNEL);

if (retval) {

dev\_err(&acm->control->dev,

"%s - usb\_submit\_urb(ctrl irq) failed\n", \_\_func\_\_);

goto error\_submit\_urb;

}

acm\_tty\_set\_termios(tty, NULL);

~~/\*~~

~~\* Unthrottle device in case the TTY was closed while throttled.~~

~~\*/~~

~~spin\_lock\_irq(&acm->read\_lock);~~

~~acm->throttled = 0;~~

~~acm->throttle\_req = 0;~~

~~spin\_unlock\_irq(&acm->read\_lock);~~

retval = acm\_submit\_read\_urbs(acm, GFP\_KERNEL);

if (retval)

goto error\_submit\_read\_urbs;

usb\_autopm\_put\_interface(acm->control);

mutex\_unlock(&acm->mutex);

return 0;

error\_submit\_read\_urbs:

for (i = 0; i < acm->rx\_buflimit; i++)

usb\_kill\_urb(acm->read\_urbs[i]);

usb\_kill\_urb(acm->ctrlurb);

error\_submit\_urb:

usb\_autopm\_put\_interface(acm->control);

error\_get\_interface:

disconnected:

mutex\_unlock(&acm->mutex);

return usb\_translate\_errors(retval);

}

static void acm\_port\_destruct(struct tty\_port \*port)

{

struct acm \*acm = container\_of(port, struct acm, port);

acm\_release\_minor(acm);

usb\_put\_intf(acm->control);

kfree(acm->country\_codes);

kfree(acm);

}

static void acm\_port\_shutdown(struct tty\_port \*port)

{

struct acm \*acm = container\_of(port, struct acm, port);

struct urb \*urb;

struct acm\_wb \*wb;

/\*

\* Need to grab write\_lock to prevent race with resume, but no need to

\* hold it due to the tty-port initialised flag.

\*/

spin\_lock\_irq(&acm->write\_lock);

spin\_unlock\_irq(&acm->write\_lock);

usb\_autopm\_get\_interface\_no\_resume(acm->control);

acm->control->needs\_remote\_wakeup = 0;

usb\_autopm\_put\_interface(acm->control);

for (;;) {

urb = usb\_get\_from\_anchor(&acm->delayed);

if (!urb)

break;

wb = urb->context;

wb->use = 0;

usb\_autopm\_put\_interface\_async(acm->control);

}

acm\_kill\_urbs(acm);

}

static void acm\_tty\_cleanup(struct tty\_struct \*tty)

{

struct acm \*acm = tty->driver\_data;

tty\_port\_put(&acm->port);

}

static void acm\_tty\_hangup(struct tty\_struct \*tty)

{

struct acm \*acm = tty->driver\_data;

tty\_port\_hangup(&acm->port);

}

static void acm\_tty\_close(struct tty\_struct \*tty, struct file \*filp)

{

struct acm \*acm = tty->driver\_data;

tty\_port\_close(&acm->port, tty, filp);

}

static int acm\_tty\_write(struct tty\_struct \*tty,

const unsigned char \*buf, int count)

{

struct acm \*acm = tty->driver\_data;

int stat;

unsigned long flags;

int wbn;

struct acm\_wb \*wb;

if (!count)

return 0;

dev\_vdbg(&acm->data->dev, "%d bytes from tty layer\n", count);

spin\_lock\_irqsave(&acm->write\_lock, flags);

wbn = acm\_wb\_alloc(acm);

if (wbn < 0) {

spin\_unlock\_irqrestore(&acm->write\_lock, flags);

return 0;

}

wb = &acm->wb[wbn];

if (!acm->dev) {

wb->use = 0;

spin\_unlock\_irqrestore(&acm->write\_lock, flags);

return -ENODEV;

}

count = (count > acm->writesize) ? acm->writesize : count;

dev\_vdbg(&acm->data->dev, "writing %d bytes\n", count);

memcpy(wb->buf, buf, count);

wb->len = count;

stat = usb\_autopm\_get\_interface\_async(acm->control);

if (stat) {

wb->use = 0;

spin\_unlock\_irqrestore(&acm->write\_lock, flags);

return stat;

}

if (acm->susp\_count) {

if (acm->putbuffer) {

/\* now to preserve order \*/

usb\_anchor\_urb(acm->putbuffer->urb, &acm->delayed);

acm->putbuffer = NULL;

}

usb\_anchor\_urb(wb->urb, &acm->delayed);

spin\_unlock\_irqrestore(&acm->write\_lock, flags);

return count;

} else {

if (acm->putbuffer) {

/\* at this point there is no good way to handle errors \*/

acm\_start\_wb(acm, acm->putbuffer);

acm->putbuffer = NULL;

}

}

stat = acm\_start\_wb(acm, wb);

spin\_unlock\_irqrestore(&acm->write\_lock, flags);

if (stat < 0)

return stat;

return count;

}

static int get\_serial\_info(struct acm \*acm, struct serial\_struct \_\_user \*info)

{

struct serial\_struct tmp;

memset(&tmp, 0, sizeof(tmp));

tmp.xmit\_fifo\_size = acm->writesize;

tmp.baud\_base = le32\_to\_cpu(acm->line.dwDTERate);

tmp.close\_delay = acm->port.close\_delay / 10;

tmp.closing\_wait = acm->port.closing\_wait == ASYNC\_CLOSING\_WAIT\_NONE ?

ASYNC\_CLOSING\_WAIT\_NONE :

acm->port.closing\_wait / 10;

if (copy\_to\_user(info, &tmp, sizeof(tmp)))

return -EFAULT;

else

return 0;

}

static int set\_serial\_info(struct acm \*acm,

struct serial\_struct \_\_user \*newinfo)

{

struct serial\_struct new\_serial;

unsigned int closing\_wait, close\_delay;

int retval = 0;

if (copy\_from\_user(&new\_serial, newinfo, sizeof(new\_serial)))

return -EFAULT;

close\_delay = new\_serial.close\_delay \* 10;

closing\_wait = new\_serial.closing\_wait == ASYNC\_CLOSING\_WAIT\_NONE ?

ASYNC\_CLOSING\_WAIT\_NONE : new\_serial.closing\_wait \* 10;

mutex\_lock(&acm->port.mutex);

if (!capable(CAP\_SYS\_ADMIN)) {

if ((close\_delay != acm->port.close\_delay) ||

(closing\_wait != acm->port.closing\_wait))

retval = -EPERM;

else

retval = -EOPNOTSUPP;

} else {

acm->port.close\_delay = close\_delay;

acm->port.closing\_wait = closing\_wait;

}

mutex\_unlock(&acm->port.mutex);

return retval;

}

static int wait\_serial\_change(struct acm \*acm, unsigned long arg)

{

int rv = 0;

DECLARE\_WAITQUEUE(wait, current);

struct async\_icount old, new;

do {

spin\_lock\_irq(&acm->read\_lock);

old = acm->oldcount;

new = acm->iocount;

acm->oldcount = new;

spin\_unlock\_irq(&acm->read\_lock);

if ((arg & TIOCM\_DSR) &&

old.dsr != new.dsr)

break;

if ((arg & TIOCM\_CD) &&

old.dcd != new.dcd)

break;

if ((arg & TIOCM\_RI) &&

old.rng != new.rng)

break;

add\_wait\_queue(&acm->wioctl, &wait);

set\_current\_state(TASK\_INTERRUPTIBLE);

schedule();

remove\_wait\_queue(&acm->wioctl, &wait);

if (acm->disconnected) {

if (arg & TIOCM\_CD)

break;

else

rv = -ENODEV;

} else {

if (signal\_pending(current))

rv = -ERESTARTSYS;

}

} while (!rv);

return rv;

}

static void acm\_tty\_set\_termios(struct tty\_struct \*tty,

struct ktermios \*termios\_old)

{

struct acm \*acm = tty->driver\_data;

struct ktermios \*termios = &tty->termios;

struct usb\_cdc\_line\_coding newline;

int newctrl = acm->ctrlout;

newline.dwDTERate = cpu\_to\_le32(tty\_get\_baud\_rate(tty));

newline.bCharFormat = termios->c\_cflag & CSTOPB ? 2 : 0;

newline.bParityType = termios->c\_cflag & PARENB ?

(termios->c\_cflag & PARODD ? 1 : 2) +

(termios->c\_cflag & CMSPAR ? 2 : 0) : 0;

switch (termios->c\_cflag & CSIZE) {

case CS5:

newline.bDataBits = 5;

break;

case CS6:

newline.bDataBits = 6;

break;

case CS7:

newline.bDataBits = 7;

break;

case CS8:

default:

newline.bDataBits = 8;

break;

}

/\* FIXME: Needs to clear unsupported bits in the termios \*/

acm->clocal = ((termios->c\_cflag & CLOCAL) != 0);

if (C\_BAUD(tty) == B0) {

newline.dwDTERate = acm->line.dwDTERate;

newctrl &= ~ACM\_CTRL\_DTR;

} else if (termios\_old && (termios\_old->c\_cflag & CBAUD) == B0) {

newctrl |= ACM\_CTRL\_DTR;

}

if (newctrl != acm->ctrlout)

acm\_set\_control(acm, acm->ctrlout = newctrl);

if (memcmp(&acm->line, &newline, sizeof newline)) {

memcpy(&acm->line, &newline, sizeof newline);

dev\_dbg(&acm->control->dev, "%s - set line: %d %d %d %d\n",

\_\_func\_\_,

le32\_to\_cpu(newline.dwDTERate),

newline.bCharFormat, newline.bParityType,

newline.bDataBits);

acm\_set\_line(acm, &acm->line);

}

}

static const struct tty\_port\_operations acm\_port\_ops = {

.dtr\_rts = acm\_port\_dtr\_rts,

.shutdown = acm\_port\_shutdown,

.activate = acm\_port\_activate,

.destruct = acm\_port\_destruct,

};

10.

/\* Little helpers: write/read buffers free \*/

static void acm\_write\_buffers\_free(struct acm \*acm)

{

int i;

struct acm\_wb \*wb;

for (wb = &acm->wb[0], i = 0; i < ACM\_NW; i++, wb++)

usb\_free\_coherent(acm->dev, acm->writesize, wb->buf, wb->dmah);

}

static void acm\_read\_buffers\_free(struct acm \*acm)

{

int i;

for (i = 0; i < acm->rx\_buflimit; i++)

usb\_free\_coherent(acm->dev, acm->readsize,

acm->read\_buffers[i].base, acm->read\_buffers[i].dma);

}

/\* Little helper: write buffers allocate \*/

static int acm\_write\_buffers\_alloc(struct acm \*acm)

{

int i;

struct acm\_wb \*wb;

for (wb = &acm->wb[0], i = 0; i < ACM\_NW; i++, wb++) {

wb->buf = usb\_alloc\_coherent(acm->dev, acm->writesize, GFP\_KERNEL,

&wb->dmah);

if (!wb->buf) {

while (i != 0) {

--i;

--wb;

usb\_free\_coherent(acm->dev, acm->writesize,

wb->buf, wb->dmah);

}

return -ENOMEM;

}

}

return 0;

}

static int acm\_probe(struct usb\_interface \*intf,

const struct usb\_device\_id \*id)

{

struct usb\_cdc\_union\_desc \*union\_header = NULL;

struct usb\_cdc\_call\_mgmt\_descriptor \*cmgmd = NULL;

unsigned char \*buffer = intf->altsetting->extra;

int buflen = intf->altsetting->extralen;

struct usb\_interface \*control\_interface;

struct usb\_interface \*data\_interface;

struct usb\_endpoint\_descriptor \*epctrl = NULL;

struct usb\_endpoint\_descriptor \*epread = NULL;

struct usb\_endpoint\_descriptor \*epwrite = NULL;

struct usb\_device \*usb\_dev = interface\_to\_usbdev(intf);

struct usb\_cdc\_parsed\_header h;

struct acm \*acm;

int minor;

int ctrlsize, readsize;

u8 \*buf;

int call\_intf\_num = -1;

int data\_intf\_num = -1;

unsigned long quirks;

int num\_rx\_buf;

int i;

int combined\_interfaces = 0;

struct device \*tty\_dev;

int rv = -ENOMEM;

look\_for\_collapsed\_interface:

for (i = 0; i < 3; i++) {

struct usb\_endpoint\_descriptor \*ep;

ep = &data\_interface->cur\_altsetting->endpoint[i].desc;

if (usb\_endpoint\_is\_int\_in(ep))

epctrl = ep;

else if (usb\_endpoint\_is\_bulk\_out(ep))

epwrite = ep;

else if (usb\_endpoint\_is\_bulk\_in(ep))

epread = ep;

else

return -EINVAL;

}

if (!epctrl || !epread || !epwrite)

return -ENODEV;

else

goto made\_compressed\_probe;

}

skip\_normal\_probe:

/\*workaround for switched interfaces \*/

if (data\_interface->cur\_altsetting->desc.bInterfaceClass

!= CDC\_DATA\_INTERFACE\_TYPE) {

if (control\_interface->cur\_altsetting->desc.bInterfaceClass

== CDC\_DATA\_INTERFACE\_TYPE) {

dev\_dbg(&intf->dev,

"Your device has switched interfaces.\n");

swap(control\_interface, data\_interface);

} else {

return -EINVAL;

}

}

/\* Accept probe requests only for the control interface \*/

if (!combined\_interfaces && intf != control\_interface)

return -ENODEV;

if (!combined\_interfaces && usb\_interface\_claimed(data\_interface)) {

/\* valid in this context \*/

dev\_dbg(&intf->dev, "The data interface isn't available\n");

return -EBUSY;

}

if (data\_interface->cur\_altsetting->desc.bNumEndpoints < 2 ||

control\_interface->cur\_altsetting->desc.bNumEndpoints == 0)

return -EINVAL;

epctrl = &control\_interface->cur\_altsetting->endpoint[0].desc;

epread = &data\_interface->cur\_altsetting->endpoint[0].desc;

epwrite = &data\_interface->cur\_altsetting->endpoint[1].desc;

/\* workaround for switched endpoints \*/

if (!usb\_endpoint\_dir\_in(epread)) {

/\* descriptors are swapped \*/

dev\_dbg(&intf->dev,

"The data interface has switched endpoints\n");

swap(epread, epwrite);

}

made\_compressed\_probe:

dev\_dbg(&intf->dev, "interfaces are valid\n");

acm = kzalloc(sizeof(struct acm), GFP\_KERNEL);

if (acm == NULL)

goto alloc\_fail;

minor = acm\_alloc\_minor(acm);

if (minor < 0)

goto alloc\_fail1;

ctrlsize = usb\_endpoint\_maxp(epctrl);

readsize = usb\_endpoint\_maxp(epread) \*

(quirks == SINGLE\_RX\_URB ? 1 : 2);

acm->combined\_interfaces = combined\_interfaces;

acm->writesize = usb\_endpoint\_maxp(epwrite) \* 20;

acm->control = control\_interface;

acm->data = data\_interface;

acm->minor = minor;

acm->dev = usb\_dev;

if (h.usb\_cdc\_acm\_descriptor)

acm->ctrl\_caps = h.usb\_cdc\_acm\_descriptor->bmCapabilities;

if (quirks & NO\_CAP\_LINE)

acm->ctrl\_caps &= ~USB\_CDC\_CAP\_LINE;

acm->ctrlsize = ctrlsize;

acm->readsize = readsize;

acm->rx\_buflimit = num\_rx\_buf;

INIT\_WORK(&acm->work, acm\_softint);

init\_waitqueue\_head(&acm->wioctl);

spin\_lock\_init(&acm->write\_lock);

spin\_lock\_init(&acm->read\_lock);

mutex\_init(&acm->mutex);

if (usb\_endpoint\_xfer\_int(epread)) {

acm->bInterval = epread->bInterval;

acm->in = usb\_rcvintpipe(usb\_dev, epread-> bEndpointAddress);

} else {

acm->in = usb\_rcvbulkpipe(usb\_dev, epread-> bEndpointAddress);

}

if (usb\_endpoint\_xfer\_int(epwrite))

acm->out = usb\_sndintpipe(usb\_dev, epwrite-> bEndpointAddress);

else

acm->out = usb\_sndbulkpipe(usb\_dev, epwrite-> bEndpointAddress);

tty\_port\_init(&acm->port);

acm->port.ops = &acm\_port\_ops;

init\_usb\_anchor(&acm->delayed);

acm->quirks = quirks;

buf = usb\_alloc\_coherent(usb\_dev, ctrlsize, GFP\_KERNEL, &acm-> ctrl\_dma);

if (!buf)

goto alloc\_fail2;

acm->ctrl\_buffer = buf;

if (acm\_write\_buffers\_alloc(acm) < 0)

goto alloc\_fail4;

acm->ctrlurb = usb\_alloc\_urb(0, GFP\_KERNEL);

if (!acm->ctrlurb)

goto alloc\_fail5;

for (i = 0; i < num\_rx\_buf; i++) {

struct acm\_rb \*rb = &(acm->read\_buffers[i]);

struct urb \*urb;

rb->base = usb\_alloc\_coherent(acm->dev, readsize, GFP\_KERNEL,

&rb->dma);

if (!rb->base)

goto alloc\_fail6;

rb->index = i;

rb->instance = acm;

urb = usb\_alloc\_urb(0, GFP\_KERNEL);

if (!urb)

goto alloc\_fail6;

urb->transfer\_flags |= URB\_NO\_TRANSFER\_DMA\_MAP;

urb->transfer\_dma = rb->dma;

if (usb\_endpoint\_xfer\_int(epread))

usb\_fill\_int\_urb(urb, acm->dev, acm->in, rb->base,

acm->readsize,

acm\_read\_bulk\_callback, rb,

acm->bInterval);

else

usb\_fill\_bulk\_urb(urb, acm->dev, acm->in, rb->base,

acm->readsize,

acm\_read\_bulk\_callback, rb);

acm->read\_urbs[i] = urb;

\_\_set\_bit(i, &acm->read\_urbs\_free);

}

for (i = 0; i < ACM\_NW; i++) {

struct acm\_wb \*snd = &(acm->wb[i]);

snd->urb = usb\_alloc\_urb(0, GFP\_KERNEL);

if (snd->urb == NULL)

goto alloc\_fail7;

if (usb\_endpoint\_xfer\_int(epwrite))

usb\_fill\_int\_urb(snd->urb, usb\_dev, acm->out,

NULL, acm->writesize, acm\_write\_bulk, snd, epwrite->bInterval);

else

usb\_fill\_bulk\_urb(snd->urb, usb\_dev, acm->out,

NULL, acm->writesize, acm\_write\_bulk, snd);

snd->urb->transfer\_flags |= URB\_NO\_TRANSFER\_DMA\_MAP;

if (quirks & SEND\_ZERO\_PACKET)

snd->urb->transfer\_flags |= URB\_ZERO\_PACKET;

snd->instance = acm;

}

usb\_set\_intfdata(intf, acm);

i = device\_create\_file(&intf->dev, &dev\_attr\_bmCapabilities);

if (i < 0)

goto alloc\_fail7;

if (h.usb\_cdc\_country\_functional\_desc) {

/\* export the country data \*/

struct usb\_cdc\_country\_functional\_desc \* cfd =

h.usb\_cdc\_country\_functional\_desc;

acm->country\_codes = kmalloc(cfd->bLength - 4, GFP\_KERNEL);

if (!acm->country\_codes)

goto skip\_countries;

acm->country\_code\_size = cfd->bLength - 4;

memcpy(acm->country\_codes, (u8 \*)&cfd->wCountyCode0,

cfd->bLength - 4);

acm->country\_rel\_date = cfd->iCountryCodeRelDate;

i = device\_create\_file(&intf->dev, &dev\_attr\_wCountryCodes);

if (i < 0) {

kfree(acm->country\_codes);

acm->country\_codes = NULL;

acm->country\_code\_size = 0;

goto skip\_countries;

}

i = device\_create\_file(&intf->dev,

&dev\_attr\_iCountryCodeRelDate);

if (i < 0) {

device\_remove\_file(&intf->dev, &dev\_attr\_wCountryCodes);

kfree(acm->country\_codes);

acm->country\_codes = NULL;

acm->country\_code\_size = 0;

goto skip\_countries;

}

}

skip\_countries:

usb\_fill\_int\_urb(acm->ctrlurb, usb\_dev,

usb\_rcvintpipe(usb\_dev, epctrl->bEndpointAddress),

acm->ctrl\_buffer, ctrlsize, acm\_ctrl\_irq, acm,

/\* works around buggy devices \*/

epctrl->bInterval ? epctrl->bInterval : 16);

acm->ctrlurb->transfer\_flags |= URB\_NO\_TRANSFER\_DMA\_MAP;

acm->ctrlurb->transfer\_dma = acm->ctrl\_dma;

dev\_info(&intf->dev, "ttyACM%d: USB ACM device\n", minor);

acm->line.dwDTERate = cpu\_to\_le32(9600);

acm->line.bDataBits = 8;

acm\_set\_line(acm, &acm->line);

usb\_driver\_claim\_interface(&acm\_driver, data\_interface, acm);

usb\_set\_intfdata(data\_interface, acm);

usb\_get\_intf(control\_interface);

tty\_dev = tty\_port\_register\_device(&acm->port, acm\_tty\_driver, minor,

&control\_interface->dev);

if (IS\_ERR(tty\_dev)) {

rv = PTR\_ERR(tty\_dev);

goto alloc\_fail8;

}

if (quirks & CLEAR\_HALT\_CONDITIONS) {

usb\_clear\_halt(usb\_dev, acm->in);

usb\_clear\_halt(usb\_dev, acm->out);

}

return 0;

alloc\_fail8:

if (acm->country\_codes) {

device\_remove\_file(&acm->control->dev,

&dev\_attr\_wCountryCodes);

device\_remove\_file(&acm->control->dev,

&dev\_attr\_iCountryCodeRelDate);

kfree(acm->country\_codes);

}

device\_remove\_file(&acm->control->dev, &dev\_attr\_bmCapabilities);

alloc\_fail7:

usb\_set\_intfdata(intf, NULL);

for (i = 0; i < ACM\_NW; i++)

usb\_free\_urb(acm->wb[i].urb);

alloc\_fail6:

for (i = 0; i < num\_rx\_buf; i++)

usb\_free\_urb(acm->read\_urbs[i]);

acm\_read\_buffers\_free(acm);

usb\_free\_urb(acm->ctrlurb);

alloc\_fail5:

acm\_write\_buffers\_free(acm);

alloc\_fail4:

usb\_free\_coherent(usb\_dev, ctrlsize, acm->ctrl\_buffer, acm->ctrl\_dma);

alloc\_fail2:

acm\_release\_minor(acm);

alloc\_fail1:

kfree(acm);

alloc\_fail:

return rv;

}

static void acm\_disconnect(struct usb\_interface \*intf)

{

struct acm \*acm = usb\_get\_intfdata(intf);

struct tty\_struct \*tty;

/\* sibling interface is already cleaning up \*/

if (!acm)

return;

mutex\_lock(&acm->mutex);

acm->disconnected = true;

if (acm->country\_codes) {

device\_remove\_file(&acm->control->dev,

&dev\_attr\_wCountryCodes);

device\_remove\_file(&acm->control->dev,

&dev\_attr\_iCountryCodeRelDate);

}

wake\_up\_all(&acm->wioctl);

device\_remove\_file(&acm->control->dev, &dev\_attr\_bmCapabilities);

usb\_set\_intfdata(acm->control, NULL);

usb\_set\_intfdata(acm->data, NULL);

mutex\_unlock(&acm->mutex);

tty = tty\_port\_tty\_get(&acm->port);

if (tty) {

tty\_vhangup(tty);

tty\_kref\_put(tty);

}

acm\_kill\_urbs(acm);

cancel\_work\_sync(&acm->work);

tty\_unregister\_device(acm\_tty\_driver, acm->minor);

acm\_write\_buffers\_free(acm);

usb\_free\_coherent(acm->dev, acm->ctrlsize, acm->ctrl\_buffer, acm->ctrl\_dma);

acm\_read\_buffers\_free(acm);

if (!acm->combined\_interfaces)

usb\_driver\_release\_interface(&acm\_driver, intf == acm->control ?

acm->data : acm->control);

tty\_port\_put(&acm->port);

}

11.