/\* -\*- Mode: C; c-basic-offset:8 ; indent-tabs-mode:t -\*- \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* modified some function to avoid crash, support Android

\* Copyright (C) 2014-2016 saki@serenegiant All rights reserved.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*

\* Android usbfs backend for libusb

\* Copyright © 2007-2009 Daniel Drake <dsd@gentoo.org>

\* Copyright © 2001 Johannes Erdfelt <johannes@erdfelt.com>

\* Copyright © 2013 Nathan Hjelm <hjelmn@mac.com>

\* Copyright © 2012-2013 Hans de Goede <hdegoede@redhat.com>

\*

\* This library is free software; you can redistribute it and/or

\* modify it under the terms of the GNU Lesser General Public

\* License as published by the Free Software Foundation; either

\* version 2.1 of the License, or (at your option) any later version.

\*

\* This library is distributed in the hope that it will be useful,

\* but WITHOUT ANY WARRANTY; without even the implied warranty of

\* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU

\* Lesser General Public License for more details.

\*

\* You should have received a copy of the GNU Lesser General Public

\* License along with this library; if not, write to the Free Software

\* Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

\*/

#define LOCAL\_DEBUG 0

#define LOG\_TAG "libusb/usbfs"

#if 1 // デバッグ情報を出さない時1

#ifndef LOG\_NDEBUG

#define LOG\_NDEBUG // LOGV/LOGD/MARKを出力しない時

#endif

#undef USE\_LOGALL // 指定したLOGxだけを出力

#else

#define USE\_LOGALL

#undef LOG\_NDEBUG

#undef NDEBUG

#define GET\_RAW\_DESCRIPTOR

#endif

#include "config.h"

#include <assert.h>

#include <ctype.h>

#include <dirent.h>

#include <errno.h>

#include <fcntl.h>

#include <poll.h>

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/ioctl.h>

#include <sys/stat.h>

#include <sys/types.h>

#include <sys/utsname.h>

#include <unistd.h>

#include "libusb.h"

#include "libusbi.h"

#include "android\_usbfs.h"

/\* sysfs vs usbfs:

\* opening a usbfs node causes the device to be resumed, so we attempt to

\* avoid this during enumeration.

\*

\* sysfs allows us to read the kernel's in-memory copies of device descriptors

\* and so forth, avoiding the need to open the device:

\* - The binary "descriptors" file contains all config descriptors since

\* 2.6.26, commit 217a9081d8e69026186067711131b77f0ce219ed

\* - The binary "descriptors" file was added in 2.6.23, commit

\* 69d42a78f935d19384d1f6e4f94b65bb162b36df, but it only contains the

\* active config descriptors

\* - The "busnum" file was added in 2.6.22, commit

\* 83f7d958eab2fbc6b159ee92bf1493924e1d0f72

\* - The "devnum" file has been present since pre-2.6.18

\* - the "bConfigurationValue" file has been present since pre-2.6.18

\*

\* If we have bConfigurationValue, busnum, and devnum, then we can determine

\* the active configuration without having to open the usbfs node in RDWR mode.

\* The busnum file is important as that is the only way we can relate sysfs

\* devices to usbfs nodes.

\*

\* If we also have all descriptors, we can obtain the device descriptor and

\* configuration without touching usbfs at all.

\*/

/\* endianness for multi-byte fields:

\*

\* Descriptors exposed by usbfs have the multi-byte fields in the device

\* descriptor as host endian. Multi-byte fields in the other descriptors are

\* bus-endian. The kernel documentation says otherwise, but it is wrong.

\*

\* In sysfs all descriptors are bus-endian.

\*/

static const char \*usbfs\_path = NULL;

/\* use usbdev\*.\* device names in /dev instead of the usbfs bus directories \*/

static int usbdev\_names = 0;

/\* Linux 2.6.32 adds support for a bulk continuation URB flag. this basically

\* allows us to mark URBs as being part of a specific logical transfer when

\* we submit them to the kernel. then, on any error except a cancellation, all

\* URBs within that transfer will be cancelled and no more URBs will be

\* accepted for the transfer, meaning that no more data can creep in.

\*

\* The BULK\_CONTINUATION flag must be set on all URBs within a bulk transfer

\* (in either direction) except the first.

\* For IN transfers, we must also set SHORT\_NOT\_OK on all URBs except the

\* last; it means that the kernel should treat a short reply as an error.

\* For OUT transfers, SHORT\_NOT\_OK must not be set. it isn't needed (OUT

\* transfers can't be short unless there's already some sort of error), and

\* setting this flag is disallowed (a kernel with USB debugging enabled will

\* reject such URBs).

\*/

static int supports\_flag\_bulk\_continuation = -1;

/\* Linux 2.6.31 fixes support for the zero length packet URB flag. This

\* allows us to mark URBs that should be followed by a zero length data

\* packet, which can be required by device- or class-specific protocols.

\*/

static int supports\_flag\_zero\_packet = -1;

/\* clock ID for monotonic clock, as not all clock sources are available on all

\* systems. appropriate choice made at initialization time. \*/

static clockid\_t monotonic\_clkid = -1;

/\* Linux 2.6.22 (commit 83f7d958eab2fbc6b159ee92bf1493924e1d0f72) adds a busnum

\* to sysfs, so we can relate devices. This also implies that we can read

\* the active configuration through bConfigurationValue \*/

static int sysfs\_can\_relate\_devices = -1;

/\* Linux 2.6.26 (commit 217a9081d8e69026186067711131b77f0ce219ed) adds all

\* config descriptors (rather then just the active config) to the sysfs

\* descriptors file, so from then on we can use them. \*/

static int sysfs\_has\_descriptors = -1;

/\* how many times have we initted (and not exited) ? \*/

static int init\_count = 0;

/\* Serialize hotplug start/stop \*/

usbi\_mutex\_static\_t android\_hotplug\_startstop\_lock = USBI\_MUTEX\_INITIALIZER;

/\* Serialize scan-devices, event-thread, and poll \*/

usbi\_mutex\_static\_t android\_hotplug\_lock = USBI\_MUTEX\_INITIALIZER;

static int android\_start\_event\_monitor(void);

static int android\_stop\_event\_monitor(void);

static int android\_scan\_devices(struct libusb\_context \*ctx);

static int sysfs\_scan\_device(struct libusb\_context \*ctx, const char \*devname);

static int detach\_kernel\_driver\_and\_claim(struct libusb\_device\_handle \*, int);

#if !defined(USE\_UDEV)

static int android\_default\_scan\_devices(struct libusb\_context \*ctx);

#endif

struct android\_device\_priv {

char \*sysfs\_dir;

unsigned char \*descriptors;

int descriptors\_len;

int active\_config; /\* cache val for !sysfs\_can\_relate\_devices \*/

int fd;

};

struct android\_device\_handle\_priv {

int fd;

uint32\_t caps;

};

enum reap\_action {

NORMAL = 0,

/\* submission failed after the first URB, so await cancellation/completion

\* of all the others \*/

SUBMIT\_FAILED,

/\* cancelled by user or timeout \*/

CANCELLED,

/\* completed multi-URB transfer in non-final URB \*/

COMPLETED\_EARLY,

/\* one or more urbs encountered a low-level error \*/

ERROR,

};

struct android\_transfer\_priv {

union {

struct usbfs\_urb \*urbs;

struct usbfs\_urb \*\*iso\_urbs;

};

enum reap\_action reap\_action;

int num\_urbs;

int num\_retired;

enum libusb\_transfer\_status reap\_status;

/\* next iso packet in user-supplied transfer to be populated \*/

int iso\_packet\_offset;

};

#if LOCAL\_DEBUG

static void dump\_urb(int ix, int fd, struct usbfs\_urb \*urb) {

LOGI("%d:fd=%d", ix, fd);

int ret = fcntl(fd, F\_GETFL);

if (UNLIKELY(ret == -1)) {

LOGE("Failed to get fd flags: %d", errno);

}

LOGI("ファイフディスクリプタフラグ:%x", ret);

LOGI("O\_ACCMODE:%x", ret & O\_ACCMODE); // 0:読み込み専用, 1:書き込み専用, 2;読み書き可

LOGI("ノンブロッキングかどうか:%d", ret & O\_NONBLOCK); // 0:ブロッキング

LOGI("%d:type=%d,endpopint=0x%02x,status=%d,flag=%d", ix, urb->type, urb->endpoint, urb->status, urb->flags);

LOGI("%d:buffer=%p,buffer\_length=%d,actual\_length=%d,start\_frame=%d", ix, urb->buffer, urb->buffer\_length, urb->actual\_length, urb->start\_frame);

LOGI("%d:number\_of\_packets=%d,error\_count=%d,signr=%d", ix, urb->number\_of\_packets, urb->error\_count, urb->signr);

LOGI("%d:usercontext=%p,iso\_frame\_desc=%p", ix, urb->usercontext, urb->iso\_frame\_desc);

}

#endif

/\*\*

\* this is original \_get\_usbfs\_fd (name changed to \_\_get\_usbfs\_fd)

\*/

static int \_\_get\_usbfs\_fd(struct libusb\_device \*dev, mode\_t mode, int silent) {

struct libusb\_context \*ctx = DEVICE\_CTX(dev);

char path[PATH\_MAX];

int fd;

int delay = 10000;

if (usbdev\_names)

snprintf(path, PATH\_MAX, "%s/usbdev%d.%d",

usbfs\_path, dev->bus\_number, dev->device\_address);

else

snprintf(path, PATH\_MAX, "%s/%03d/%03d",

usbfs\_path, dev->bus\_number, dev->device\_address);

fd = open(path, mode);

if (LIKELY(fd != -1))

return fd; /\* Success \*/

if (errno == ENOENT) {

if (!silent)

usbi\_err(ctx, "File doesn't exist, wait %d ms and try again\n", delay / 1000);

/\* Wait 10ms for USB device path creation.\*/

usleep(delay);

fd = open(path, mode);

if (LIKELY(fd != -1))

return fd; /\* Success \*/

}

if (!silent) {

usbi\_err(ctx, "libusb couldn't open USB device %s: %s",

path, strerror(errno));

if (errno == EACCES && mode == O\_RDWR)

usbi\_err(ctx, "libusb requires write access to USB "

"device nodes.");

}

if (errno == EACCES)

return LIBUSB\_ERROR\_ACCESS;

if (errno == ENOENT)

return LIBUSB\_ERROR\_NO\_DEVICE;

return LIBUSB\_ERROR\_IO;

}

static struct android\_device\_priv \*\_device\_priv(struct libusb\_device \*device);

static int \_get\_usbfs\_fd(struct libusb\_device \*device, mode\_t mode, int silent) {

#ifdef \_\_ANDROID\_\_

struct android\_device\_priv \*dpriv = \_device\_priv(device);

if (LIKELY(dpriv->fd > 0))

return dpriv->fd;

else {

// fall back to original \_get\_usbfs\_fd function

// but this call will fail on Android devices without root

#if !defined(\_\_LP64\_\_)

usbi\_dbg("fd have not set yet. device=%x,fd=%d", (int )device, dpriv->fd);

#else

usbi\_dbg("fd have not set yet. device=%x,fd=%d", (long )device, dpriv->fd);

#endif

return \_\_get\_usbfs\_fd(device, mode, silent);

}

#else

return \_\_get\_usbfs\_fd(device, mode, silent);

#endif

}

static struct android\_device\_priv \*\_device\_priv(struct libusb\_device \*dev) {

return (struct android\_device\_priv \*) dev->os\_priv;

}

static struct android\_device\_handle\_priv \*\_device\_handle\_priv(

struct libusb\_device\_handle \*handle) {

return (struct android\_device\_handle\_priv \*) handle->os\_priv;

}

/\* check dirent for a /dev/usbdev%d.%d name

\* optionally return bus/device on success \*/

static int \_is\_usbdev\_entry(struct dirent \*entry, int \*bus\_p, int \*dev\_p) {

int busnum, devnum;

if (sscanf(entry->d\_name, "usbdev%d.%d", &busnum, &devnum) != 2)

return LIBUSB\_SUCCESS;

usbi\_dbg("found: %s", entry->d\_name);

if (bus\_p != NULL)

\*bus\_p = busnum;

if (dev\_p != NULL)

\*dev\_p = devnum;

return 1;

}

static int check\_usb\_vfs(const char \*dirname) {

DIR \*dir;

struct dirent \*entry;

int found = 0;

dir = opendir(dirname);

if (!dir)

return LIBUSB\_SUCCESS;

while ((entry = readdir(dir)) != NULL ) {

if (entry->d\_name[0] == '.')

continue;

/\* We assume if we find any files that it must be the right place \*/

found = 1;

break;

}

closedir(dir);

return found;

}

static const char \*find\_usbfs\_path(void) {

const char \*path = "/dev/bus/usb";

const char \*ret = NULL;

if (check\_usb\_vfs(path)) {

ret = path;

} else {

path = "/proc/bus/usb";

if (check\_usb\_vfs(path))

ret = path;

}

/\* look for /dev/usbdev\*.\* if the normal places fail \*/

if (ret == NULL) {

struct dirent \*entry;

DIR \*dir;

path = "/dev";

dir = opendir(path);

if (dir != NULL) {

while ((entry = readdir(dir)) != NULL ) {

if (\_is\_usbdev\_entry(entry, NULL, NULL)) {

/\* found one; that's enough \*/

ret = path;

usbdev\_names = 1;

break;

}

}

closedir(dir);

}

}

if (ret != NULL)

usbi\_dbg("found usbfs at %s", ret);

return ret;

}

/\* the monotonic clock is not usable on all systems (e.g. embedded ones often

\* seem to lack it). fall back to REALTIME if we have to. \*/

static clockid\_t find\_monotonic\_clock(void) {

#ifdef CLOCK\_MONOTONIC

struct timespec ts;

int r;

/\* Linux 2.6.28 adds CLOCK\_MONOTONIC\_RAW but we don't use it

\* because it's not available through timerfd \*/

r = clock\_gettime(CLOCK\_MONOTONIC, &ts);

if (r == 0)

return CLOCK\_MONOTONIC;

usbi\_dbg("monotonic clock doesn't work, errno %d", errno);

#endif

return CLOCK\_REALTIME;

}

static int kernel\_version\_ge(int major, int minor, int sublevel) {

struct utsname uts;

int atoms, kmajor, kminor, ksublevel;

if (uname(&uts) < 0)

return -1;

atoms = sscanf(uts.release, "%d.%d.%d", &kmajor, &kminor, &ksublevel);

if (UNLIKELY(atoms < 1))

return -1;

if (kmajor > major)

return 1;

if (kmajor < major)

return 0;

/\* kmajor == major \*/

if (atoms < 2)

return 0 == minor && 0 == sublevel;

if (kminor > minor)

return 1;

if (kminor < minor)

return 0;

/\* kminor == minor \*/

if (atoms < 3)

return 0 == sublevel;

return ksublevel >= sublevel;

}

static int op\_init2(struct libusb\_context \*ctx, const char \*usbfs) { // XXX

struct stat statbuf;

int r;

ENTER();

if (!usbfs || !strlen(usbfs)) {

usbfs\_path = find\_usbfs\_path();

} else {

usbfs\_path = usbfs;

}

if (UNLIKELY(!usbfs\_path)) {

LOGE("could not find usbfs");

usbi\_err(ctx, "could not find usbfs");

RETURN(LIBUSB\_ERROR\_OTHER, int);

}

if (monotonic\_clkid == -1)

monotonic\_clkid = find\_monotonic\_clock();

if (supports\_flag\_bulk\_continuation == -1) {

/\* bulk continuation URB flag available from Linux 2.6.32 \*/

supports\_flag\_bulk\_continuation = kernel\_version\_ge(2, 6, 32);

if (supports\_flag\_bulk\_continuation == -1) {

LOGE("error checking for bulk continuation support");

usbi\_err(ctx, "error checking for bulk continuation support");

RETURN(LIBUSB\_ERROR\_OTHER, int);

}

}

if (supports\_flag\_bulk\_continuation)

usbi\_dbg("bulk continuation flag supported");

if (-1 == supports\_flag\_zero\_packet) {

/\* zero length packet URB flag fixed since Linux 2.6.31 \*/

supports\_flag\_zero\_packet = kernel\_version\_ge(2, 6, 31);

if (-1 == supports\_flag\_zero\_packet) {

LOGE("error checking for zero length packet support");

usbi\_err(ctx, "error checking for zero length packet support");

RETURN(LIBUSB\_ERROR\_OTHER, int);

}

}

if (supports\_flag\_zero\_packet)

usbi\_dbg("zero length packet flag supported");

if (-1 == sysfs\_has\_descriptors) {

/\* sysfs descriptors has all descriptors since Linux 2.6.26 \*/

sysfs\_has\_descriptors = kernel\_version\_ge(2, 6, 26);

if (UNLIKELY(-1 == sysfs\_has\_descriptors)) {

LOGE("error checking for sysfs descriptors");

usbi\_err(ctx, "error checking for sysfs descriptors");

RETURN(LIBUSB\_ERROR\_OTHER, int);

}

}

if (-1 == sysfs\_can\_relate\_devices) {

/\* sysfs has busnum since Linux 2.6.22 \*/

sysfs\_can\_relate\_devices = kernel\_version\_ge(2, 6, 22);

if (UNLIKELY(-1 == sysfs\_can\_relate\_devices)) {

LOGE("error checking for sysfs busnum");

usbi\_err(ctx, "error checking for sysfs busnum");

RETURN(LIBUSB\_ERROR\_OTHER, int);

}

}

if (sysfs\_can\_relate\_devices || sysfs\_has\_descriptors) {

r = stat(SYSFS\_DEVICE\_PATH, &statbuf);

if (r != 0 || !S\_ISDIR(statbuf.st\_mode)) {

usbi\_warn(ctx, "sysfs not mounted");

sysfs\_can\_relate\_devices = 0;

sysfs\_has\_descriptors = 0;

}

}

if (sysfs\_can\_relate\_devices)

usbi\_dbg("sysfs can relate devices");

if (sysfs\_has\_descriptors)

usbi\_dbg("sysfs has complete descriptors");

usbi\_mutex\_static\_lock(&android\_hotplug\_startstop\_lock);

r = LIBUSB\_SUCCESS;

if (init\_count == 0) {

LOGI("start up hotplug event handler");

int r = android\_start\_event\_monitor();

if (r != LIBUSB\_SUCCESS) {

LOGE("warning: error starting hotplug event monitor");

usbi\_err(ctx, "warning: error starting hotplug event monitor");

}

}

if (r == LIBUSB\_SUCCESS) {

LOGI("call android\_scan\_devices");

r = android\_scan\_devices(ctx);

if (r == LIBUSB\_SUCCESS)

init\_count++;

else if (init\_count == 0)

android\_stop\_event\_monitor();

} else {

LOGE("error starting hotplug event monitor");

usbi\_err(ctx, "error starting hotplug event monitor");

}

usbi\_mutex\_static\_unlock(&android\_hotplug\_startstop\_lock);

RETURN(r, int);

}

static int op\_init(struct libusb\_context \*ctx) {

return op\_init2(ctx, NULL);

#if 0

struct stat statbuf;

int r;

usbfs\_path = find\_usbfs\_path();

if (UNLIKELY(!usbfs\_path)) {

usbi\_err(ctx, "could not find usbfs");

return LIBUSB\_ERROR\_OTHER;

}

if (monotonic\_clkid == -1)

monotonic\_clkid = find\_monotonic\_clock();

if (supports\_flag\_bulk\_continuation == -1) {

/\* bulk continuation URB flag available from Linux 2.6.32 \*/

supports\_flag\_bulk\_continuation = kernel\_version\_ge(2, 6, 32);

if (supports\_flag\_bulk\_continuation == -1) {

usbi\_err(ctx, "error checking for bulk continuation support");

return LIBUSB\_ERROR\_OTHER;

}

}

if (supports\_flag\_bulk\_continuation)

usbi\_dbg("bulk continuation flag supported");

if (-1 == supports\_flag\_zero\_packet) {

/\* zero length packet URB flag fixed since Linux 2.6.31 \*/

supports\_flag\_zero\_packet = kernel\_version\_ge(2, 6, 31);

if (-1 == supports\_flag\_zero\_packet) {

usbi\_err(ctx, "error checking for zero length packet support");

return LIBUSB\_ERROR\_OTHER;

}

}

if (supports\_flag\_zero\_packet)

usbi\_dbg("zero length packet flag supported");

if (-1 == sysfs\_has\_descriptors) {

/\* sysfs descriptors has all descriptors since Linux 2.6.26 \*/

sysfs\_has\_descriptors = kernel\_version\_ge(2, 6, 26);

if (UNLIKELY(-1 == sysfs\_has\_descriptors)) {

usbi\_err(ctx, "error checking for sysfs descriptors");

return LIBUSB\_ERROR\_OTHER;

}

}

if (-1 == sysfs\_can\_relate\_devices) {

/\* sysfs has busnum since Linux 2.6.22 \*/

sysfs\_can\_relate\_devices = kernel\_version\_ge(2, 6, 22);

if (UNLIKELY(-1 == sysfs\_can\_relate\_devices)) {

usbi\_err(ctx, "error checking for sysfs busnum");

return LIBUSB\_ERROR\_OTHER;

}

}

if (sysfs\_can\_relate\_devices || sysfs\_has\_descriptors) {

r = stat(SYSFS\_DEVICE\_PATH, &statbuf);

if (r != 0 || !S\_ISDIR(statbuf.st\_mode)) {

usbi\_warn(ctx, "sysfs not mounted");

sysfs\_can\_relate\_devices = 0;

sysfs\_has\_descriptors = 0;

}

}

if (sysfs\_can\_relate\_devices)

usbi\_dbg("sysfs can relate devices");

if (sysfs\_has\_descriptors)

usbi\_dbg("sysfs has complete descriptors");

usbi\_mutex\_static\_lock(&android\_hotplug\_startstop\_lock);

r = LIBUSB\_SUCCESS;

if (init\_count == 0) {

LOGI("start up hotplug event handler");

r = android\_start\_event\_monitor();

}

if (r == LIBUSB\_SUCCESS) {

r = android\_scan\_devices(ctx);

if (r == LIBUSB\_SUCCESS)

init\_count++;

else if (init\_count == 0)

android\_stop\_event\_monitor();

} else

usbi\_err(ctx, "error starting hotplug event monitor");

usbi\_mutex\_static\_unlock(&android\_hotplug\_startstop\_lock);

return r;

#endif

}

static void op\_exit(void) {

ENTER();

usbi\_mutex\_static\_lock(&android\_hotplug\_startstop\_lock);

assert(init\_count != 0);

if (!--init\_count) {

/\* tear down event handler \*/

(void) android\_stop\_event\_monitor();

}

usbi\_mutex\_static\_unlock(&android\_hotplug\_startstop\_lock);

EXIT();

}

static int android\_start\_event\_monitor(void) {

ENTER();

#ifdef \_\_ANDROID\_\_

// do nothing

RETURN(LIBUSB\_SUCCESS, int);

#else

#if defined(USE\_UDEV)

RETURN(android\_udev\_start\_event\_monitor(), int);

#else

RETURN(android\_netlink\_start\_event\_monitor(), int);

#endif

#endif

}

static int android\_stop\_event\_monitor(void) {

ENTER();

#ifdef \_\_ANDROID\_\_

RETURN(LIBUSB\_SUCCESS, int);

#else

#if defined(USE\_UDEV)

RETURN(android\_udev\_stop\_event\_monitor(), int);

#else

RETURN(android\_netlink\_stop\_event\_monitor(), int);

#endif

#endif

}

static int android\_scan\_devices(struct libusb\_context \*ctx) {

ENTER();

int ret = LIBUSB\_SUCCESS;

#ifdef \_\_ANDROID\_\_

// do nothing

#else

usbi\_mutex\_static\_lock(&android\_hotplug\_lock);

#if defined(USE\_UDEV)

ret = android\_udev\_scan\_devices(ctx);

#else

ret = android\_default\_scan\_devices(ctx);

#endif

usbi\_mutex\_static\_unlock(&android\_hotplug\_lock);

#endif

RETURN(ret, int);

}

static void op\_hotplug\_poll(void) {

ENTER();

#ifdef \_\_ANDROID\_\_

// do nothing

#else

#if defined(USE\_UDEV)

android\_udev\_hotplug\_poll();

#else

android\_netlink\_hotplug\_poll();

#endif

#endif

EXIT();

}

static int \_open\_sysfs\_attr(struct libusb\_device \*dev, const char \*attr) {

struct android\_device\_priv \*priv = \_device\_priv(dev);

char filename[PATH\_MAX];

int fd;

snprintf(filename, PATH\_MAX, "%s/%s/%s",

SYSFS\_DEVICE\_PATH, priv->sysfs\_dir, attr);

fd = open(filename, O\_RDONLY);

if (UNLIKELY(fd < 0)) {

usbi\_err(DEVICE\_CTX(dev),

"open %s failed ret=%d errno=%d", filename, fd, errno);

return LIBUSB\_ERROR\_IO;

}

return fd;

}

/\* Note only suitable for attributes which always read >= 0, < 0 is error \*/

static int \_\_read\_sysfs\_attr(struct libusb\_context \*ctx, const char \*devname,

const char \*attr) {

char filename[PATH\_MAX];

FILE \*f;

int r, value;

snprintf(filename, PATH\_MAX, "%s/%s/%s", SYSFS\_DEVICE\_PATH, devname, attr);

f = fopen(filename, "r");

if (UNLIKELY(f == NULL)) {

if (errno == ENOENT) {

/\* File doesn't exist. Assume the device has been

disconnected (see trac ticket #70). \*/

return LIBUSB\_ERROR\_NO\_DEVICE;

}

usbi\_err(ctx, "open %s failed errno=%d", filename, errno);

return LIBUSB\_ERROR\_IO;

}

r = fscanf(f, "%d", &value);

fclose(f);

if (UNLIKELY(r != 1)) {

usbi\_err(ctx, "fscanf %s returned %d, errno=%d", attr, r, errno);

return LIBUSB\_ERROR\_NO\_DEVICE; /\* For unplug race (trac #70) \*/

}

if (UNLIKELY(value < 0)) {

usbi\_err(ctx, "%s contains a negative value", filename);

return LIBUSB\_ERROR\_IO;

}

return value;

}

// XXX

static int op\_get\_raw\_descriptor(struct libusb\_device \*dev,

unsigned char \*buffer, int \*descriptors\_len, int \*host\_endian) {

struct android\_device\_priv \*priv = \_device\_priv(dev);

if (!descriptors\_len || !host\_endian)

return LIBUSB\_ERROR\_INVALID\_PARAM;

\*host\_endian = sysfs\_has\_descriptors ? 0 : 1;

if (buffer && (\*descriptors\_len >= priv->descriptors\_len)) {

memcpy(buffer, priv->descriptors, priv->descriptors\_len);

}

\*descriptors\_len = priv->descriptors\_len;

return LIBUSB\_SUCCESS;

}

static int op\_get\_device\_descriptor(struct libusb\_device \*dev,

unsigned char \*buffer, int \*host\_endian) {

struct android\_device\_priv \*priv = \_device\_priv(dev);

if (!host\_endian)

return LIBUSB\_ERROR\_INVALID\_PARAM;

\*host\_endian = sysfs\_has\_descriptors ? 0 : 1;

memcpy(buffer, priv->descriptors, DEVICE\_DESC\_LENGTH);

return LIBUSB\_SUCCESS;

}

/\* read the bConfigurationValue for a device \*/

static int sysfs\_get\_active\_config(struct libusb\_device \*dev, int \*config) {

char \*endptr;

char tmp[5] = { 0, 0, 0, 0, 0 };

long num;

int fd;

ssize\_t r;

fd = \_open\_sysfs\_attr(dev, "bConfigurationValue");

if (UNLIKELY(fd < 0))

return fd;

r = read(fd, tmp, sizeof(tmp));

close(fd);

if (UNLIKELY(r < 0)) {

usbi\_err(DEVICE\_CTX(dev),

"read bConfigurationValue failed ret=%d errno=%d", r, errno);

return LIBUSB\_ERROR\_IO;

} else if (r == 0) {

usbi\_dbg("device unconfigured");

\*config = -1;

return LIBUSB\_SUCCESS;

}

if (tmp[sizeof(tmp) - 1] != 0) {

usbi\_err(DEVICE\_CTX(dev), "not null-terminated?");

return LIBUSB\_ERROR\_IO;

} else if (tmp[0] == 0) {

usbi\_err(DEVICE\_CTX(dev), "no configuration value?");

return LIBUSB\_ERROR\_IO;

}

num = strtol(tmp, &endptr, 10);

if (endptr == tmp) {

usbi\_err(DEVICE\_CTX(dev), "error converting '%s' to integer", tmp);

return LIBUSB\_ERROR\_IO;

}

\*config = (int) num;

return LIBUSB\_SUCCESS;

}

int android\_get\_device\_address(struct libusb\_context \*ctx, int detached,

uint8\_t \*busnum, uint8\_t \*devaddr, const char \*dev\_node,

const char \*sys\_name) {

int sysfs\_attr;

usbi\_dbg("getting address for device: %s detached: %d", sys\_name, detached);

/\* can't use sysfs to read the bus and device number if the

\* device has been detached \*/

if (!sysfs\_can\_relate\_devices || detached || NULL == sys\_name) {

if (NULL == dev\_node) {

return LIBUSB\_ERROR\_OTHER;

}

/\* will this work with all supported kernel versions? \*/

if (!strncmp(dev\_node, "/dev/bus/usb", 12)) {

sscanf(dev\_node, "/dev/bus/usb/%hhd/%hhd", busnum, devaddr);

} else if (!strncmp(dev\_node, "/proc/bus/usb", 13)) {

sscanf(dev\_node, "/proc/bus/usb/%hhd/%hhd", busnum, devaddr);

}

return LIBUSB\_SUCCESS;

}

usbi\_dbg("scan %s", sys\_name);

sysfs\_attr = \_\_read\_sysfs\_attr(ctx, sys\_name, "busnum");

if (0 > sysfs\_attr)

return sysfs\_attr;

if (sysfs\_attr > 255)

return LIBUSB\_ERROR\_INVALID\_PARAM;

\*busnum = (uint8\_t) sysfs\_attr;

sysfs\_attr = \_\_read\_sysfs\_attr(ctx, sys\_name, "devnum");

if (0 > sysfs\_attr)

return sysfs\_attr;

if (sysfs\_attr > 255)

return LIBUSB\_ERROR\_INVALID\_PARAM;

\*devaddr = (uint8\_t) sysfs\_attr;

usbi\_dbg("bus=%d dev=%d", \*busnum, \*devaddr);

return LIBUSB\_SUCCESS;

}

/\*

\* Return offset of the first descriptor with the given type

\* return 0 if the buffer is already placed at the specific descriptor.

\* this is the difference from seek\_to\_next\_descriptor

\*/

static int seek\_to\_first\_descriptor(struct libusb\_context \*ctx,

uint8\_t descriptor\_type, unsigned char \*buffer, int size) {

struct usb\_descriptor\_header header;

int i;

for (i = 0; size >= 0; i += header.bLength, size -= header.bLength) {

if (size == 0)

return LIBUSB\_ERROR\_NOT\_FOUND;

if (size < LIBUSB\_DT\_HEADER\_SIZE) {

usbi\_err(ctx, "short descriptor read %d/2", size);

return LIBUSB\_ERROR\_IO;

}

usbi\_parse\_descriptor(buffer + i, "bb", &header, 0);

if (header.bDescriptorType == descriptor\_type) // XXX

return i;

}

usbi\_err(ctx, "bLength overflow by %d bytes", -size);

return LIBUSB\_ERROR\_IO;

}

/\* Return offset of the next descriptor with the given type \*/

static int seek\_to\_next\_descriptor(struct libusb\_context \*ctx,

uint8\_t descriptor\_type, unsigned char \*buffer, int size) {

struct usb\_descriptor\_header header;

int i;

for (i = 0; size >= 0; i += header.bLength, size -= header.bLength) {

if (size == 0)

return LIBUSB\_ERROR\_NOT\_FOUND;

if (size < LIBUSB\_DT\_HEADER\_SIZE) {

usbi\_err(ctx, "short descriptor read %d/2", size);

return LIBUSB\_ERROR\_IO;

}

usbi\_parse\_descriptor(buffer + i, "bb", &header, 0);

if (i && header.bDescriptorType == descriptor\_type)

return i;

}

usbi\_err(ctx, "bLength overflow by %d bytes", -size);

return LIBUSB\_ERROR\_IO;

}

/\* Return offset to next config \*/

static int seek\_to\_next\_config(struct libusb\_context \*ctx,

unsigned char \*buffer, int size) {

struct libusb\_config\_descriptor config;

struct usb\_descriptor\_header header;

if (size == 0)

return LIBUSB\_ERROR\_NOT\_FOUND;

if (size < LIBUSB\_DT\_HEADER\_SIZE) {

usbi\_err(ctx, "short descriptor read %d/%d",

size, LIBUSB\_DT\_CONFIG\_SIZE);

return LIBUSB\_ERROR\_IO;

}

if (size < LIBUSB\_DT\_CONFIG\_SIZE) {

usbi\_err(ctx, "short descriptor read %d/%d",

size, LIBUSB\_DT\_CONFIG\_SIZE);

return LIBUSB\_ERROR\_IO;

}

usbi\_parse\_descriptor(buffer, "bbwbbbbb", &config, 0);

if (config.bDescriptorType != LIBUSB\_DT\_CONFIG) {

usbi\_err(ctx, "descriptor is not a config desc (type 0x%02x)",

config.bDescriptorType);

return LIBUSB\_ERROR\_IO;

}

/\*

\* In usbfs the config descriptors are config.wTotalLength bytes apart,

\* with any short reads from the device appearing as holes in the file.

\*

\* In sysfs wTotalLength is ignored, instead the kernel returns a

\* config descriptor with verified bLength fields, with descriptors

\* with an invalid bLength removed.

\*/

if (sysfs\_has\_descriptors) {

int next = seek\_to\_next\_descriptor(ctx, LIBUSB\_DT\_CONFIG, buffer, size);

if (next == LIBUSB\_ERROR\_NOT\_FOUND)

next = size;

if (next < 0)

return next;

if (next != config.wTotalLength)

usbi\_warn(ctx, "config length mismatch wTotalLength "

"%d real %d", config.wTotalLength, next);

return next;

} else {

if (config.wTotalLength < LIBUSB\_DT\_CONFIG\_SIZE) {

usbi\_err(ctx, "invalid wTotalLength %d",

config.wTotalLength);

return LIBUSB\_ERROR\_IO;

} else if (config.wTotalLength > size) {

usbi\_warn(ctx, "short descriptor read %d/%d",

size, config.wTotalLength);

return size;

} else

return config.wTotalLength;

}

}

static int op\_get\_config\_descriptor\_by\_value(struct libusb\_device \*dev,

uint8\_t value, unsigned char \*\*buffer, int \*host\_endian) {

struct libusb\_context \*ctx = DEVICE\_CTX(dev);

struct android\_device\_priv \*priv = \_device\_priv(dev);

unsigned char \*descriptors = priv->descriptors;

int size = priv->descriptors\_len, r;

struct libusb\_config\_descriptor \*config;

\*buffer = NULL;

/\* Unlike the device desc. config descs. are always in raw format \*/

\*host\_endian = 0;

/\* Skip device header \*/

descriptors += DEVICE\_DESC\_LENGTH;

size -= DEVICE\_DESC\_LENGTH;

// XXX at this point, we skipped device descriptor only and the next one

// will not be a config descriptor. It may be a qualifer descriptor

// or other speed config descriptor on some device.

// Therefor we need to find the first config descriptor.

// FIXME On current implementation, any descriptor other than config descriptor

// are skipped if they placed before config descriptor.

r = seek\_to\_first\_descriptor(ctx, LIBUSB\_DT\_CONFIG, descriptors, size);

if UNLIKELY(r < 0) {

LOGE("could not find config descriptor:r=%d", r);

return r;

}

descriptors += r;

size -= r;

/\* Seek till the config is found, or till "EOF" \*/

for (; ;) {

register int next = seek\_to\_next\_config(ctx, descriptors, size);

if UNLIKELY(next < 0)

return next;

config = (struct libusb\_config\_descriptor \*) descriptors;

if (config->bConfigurationValue == value) {

\*buffer = descriptors;

return next;

}

size -= next;

descriptors += next;

}

}

static int op\_get\_active\_config\_descriptor(struct libusb\_device \*dev,

unsigned char \*buffer, size\_t len, int \*host\_endian) {

int r, config;

unsigned char \*config\_desc;

if (sysfs\_can\_relate\_devices) {

r = sysfs\_get\_active\_config(dev, &config);

if (UNLIKELY(r < 0))

return r;

} else {

/\* Use cached bConfigurationValue \*/

struct android\_device\_priv \*priv = \_device\_priv(dev);

config = priv->active\_config;

}

if (config == -1)

return LIBUSB\_ERROR\_NOT\_FOUND;

r = op\_get\_config\_descriptor\_by\_value(dev, config, &config\_desc,

host\_endian);

if (UNLIKELY(r < 0))

return r;

len = MIN(len, r);

memcpy(buffer, config\_desc, len);

return len;

}

static int op\_get\_config\_descriptor(struct libusb\_device \*dev,

uint8\_t config\_index, unsigned char \*buffer, size\_t len,

int \*host\_endian) {

struct libusb\_context \*ctx = DEVICE\_CTX(dev);

struct android\_device\_priv \*priv = \_device\_priv(dev);

unsigned char \*descriptors = priv->descriptors;

int i, r, size = priv->descriptors\_len;

/\* Unlike the device desc. config descs. are always in raw format \*/

\*host\_endian = 0;

/\* Skip device header (device descriptor) \*/

descriptors += DEVICE\_DESC\_LENGTH;

size -= DEVICE\_DESC\_LENGTH;

// XXX at this point, we skipped device descriptor only and the next one

// will not be a config descriptor. It may be a qualifer descriptor

// or other speed config descriptor on some device.

// Therefor we need to find the first config descriptor.

// FIXME On current implementation, any descriptor other than config descriptor

// are skipped if they placed before config descriptor.

r = seek\_to\_first\_descriptor(ctx, LIBUSB\_DT\_CONFIG, descriptors, size);

if UNLIKELY(r < 0) {

LOGE("could not find config descriptor:r=%d", r);

return r;

}

descriptors += r;

size -= r;

/\* Seek till the config is found, or till "EOF" \*/

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

r = seek\_to\_next\_config(ctx, descriptors, size);

if (UNLIKELY(r < 0)) // if error

return r;

if (i == config\_index)

break;

size -= r;

descriptors += r;

}

len = MIN(len, r);

memcpy(buffer, descriptors, len);

return len;

}

/\* send a control message to retrieve active configuration \*/

static int usbfs\_get\_active\_config(struct libusb\_device \*dev, int fd) {

unsigned char active\_config = 0;

int r;

struct usbfs\_ctrltransfer ctrl = {

.bmRequestType = LIBUSB\_ENDPOINT\_IN,

.bRequest = LIBUSB\_REQUEST\_GET\_CONFIGURATION,

.wValue = 0,

.wIndex = 0,

.wLength = 1,

.timeout = 1000,

.data = &active\_config

};

r = ioctl(fd, IOCTL\_USBFS\_CONTROL, &ctrl);

if (UNLIKELY(r < 0)) {

if (errno == ENODEV)

return LIBUSB\_ERROR\_NO\_DEVICE;

/\* we hit this error path frequently with buggy devices :( \*/

usbi\_warn(DEVICE\_CTX(dev),

"get\_configuration failed ret=%d errno=%d", r, errno);

return LIBUSB\_ERROR\_IO;

}

return active\_config;

}

static int initialize\_device(struct libusb\_device \*dev, uint8\_t busnum,

uint8\_t devaddr, const char \*sysfs\_dir) {

struct android\_device\_priv \*priv = \_device\_priv(dev);

struct libusb\_context \*ctx = DEVICE\_CTX(dev);

int descriptors\_size = 512; /\* Begin with a 1024 byte alloc \*/

int fd, speed;

ssize\_t r;

dev->bus\_number = busnum;

dev->device\_address = devaddr;

if (sysfs\_dir) {

priv->sysfs\_dir = malloc(strlen(sysfs\_dir) + 1);

if (!priv->sysfs\_dir)

return LIBUSB\_ERROR\_NO\_MEM;

strcpy(priv->sysfs\_dir, sysfs\_dir);

/\* Note speed can contain 1.5, in this case \_\_read\_sysfs\_attr

will stop parsing at the '.' and return 1 \*/

speed = \_\_read\_sysfs\_attr(DEVICE\_CTX(dev), sysfs\_dir, "speed");

if (speed >= 0) {

switch (speed) {

case 1: dev->speed = LIBUSB\_SPEED\_LOW; break;

case 12: dev->speed = LIBUSB\_SPEED\_FULL; break;

case 480: dev->speed = LIBUSB\_SPEED\_HIGH; break;

case 5000: dev->speed = LIBUSB\_SPEED\_SUPER; break;

default:

usbi\_warn(DEVICE\_CTX(dev), "Unknown device speed: %d Mbps", speed);

}

}

}

/\* cache descriptors in memory \*/

if (sysfs\_has\_descriptors) {

fd = \_open\_sysfs\_attr(dev, "descriptors");

} else {

fd = \_get\_usbfs\_fd(dev, O\_RDONLY, 0);

}

if (fd < 0)

return fd;

do {

descriptors\_size \*= 2;

priv->descriptors = usbi\_reallocf(priv->descriptors, descriptors\_size);

if (UNLIKELY(!priv->descriptors)) {

close(fd);

return LIBUSB\_ERROR\_NO\_MEM;

}

/\* usbfs has holes in the file \*/

if (!sysfs\_has\_descriptors) {

memset(priv->descriptors + priv->descriptors\_len, 0,

descriptors\_size - priv->descriptors\_len);

}

r = read(fd, priv->descriptors + priv->descriptors\_len,

descriptors\_size - priv->descriptors\_len);

if (UNLIKELY(r < 0)) {

usbi\_err(ctx, "read descriptor failed ret=%d errno=%d", fd, errno);

close(fd);

return LIBUSB\_ERROR\_IO;

}

priv->descriptors\_len += r;

} while (priv->descriptors\_len == descriptors\_size);

close(fd);

if (UNLIKELY(priv->descriptors\_len < DEVICE\_DESC\_LENGTH)) {

usbi\_err(ctx, "short descriptor read (%d)", priv->descriptors\_len);

return LIBUSB\_ERROR\_IO;

}

if (sysfs\_can\_relate\_devices)

return LIBUSB\_SUCCESS;

/\* cache active config \*/

fd = \_get\_usbfs\_fd(dev, O\_RDWR, 1);

if (fd < 0) { // if could not get fd of usbfs with read/write access

/\* cannot send a control message to determine the active

\* config. just assume the first one is active. \*/

usbi\_warn(ctx, "Missing rw usbfs access; cannot determine "

"active configuration descriptor");

if (priv->descriptors\_len

>= (DEVICE\_DESC\_LENGTH + LIBUSB\_DT\_CONFIG\_SIZE)) {

struct libusb\_config\_descriptor config;

usbi\_parse\_descriptor(priv->descriptors + DEVICE\_DESC\_LENGTH,

"bbwbbbbb", &config, 0);

priv->active\_config = config.bConfigurationValue;

} else

priv->active\_config = -1; /\* No config dt \*/

return LIBUSB\_SUCCESS;

}

// if we could get fd of usbfs with read/write access

r = usbfs\_get\_active\_config(dev, fd);

if (r > 0) {

priv->active\_config = r;

r = LIBUSB\_SUCCESS;

} else if (r == 0) {

/\* some buggy devices have a configuration 0, but we're

\* reaching into the corner of a corner case here, so let's

\* not support buggy devices in these circumstances.

\* stick to the specs: a configuration value of 0 means

\* unconfigured. \*/

usbi\_dbg("active cfg 0? assuming unconfigured device");

priv->active\_config = -1;

r = LIBUSB\_SUCCESS;

} else if (r == LIBUSB\_ERROR\_IO) {

/\* buggy devices sometimes fail to report their active config.

\* assume unconfigured and continue the probing \*/

usbi\_warn(ctx, "couldn't query active configuration, assuming"

" unconfigured");

priv->active\_config = -1;

r = LIBUSB\_SUCCESS;

} /\* else r < 0, just return the error code \*/

close(fd);

return r;

}

static int android\_get\_parent\_info(struct libusb\_device \*dev,

const char \*sysfs\_dir) {

struct libusb\_context \*ctx = DEVICE\_CTX(dev);

struct libusb\_device \*it;

char \*parent\_sysfs\_dir, \*tmp;

int ret, add\_parent = 1;

/\* XXX -- can we figure out the topology when using usbfs? \*/

if (NULL == sysfs\_dir || 0 == strncmp(sysfs\_dir, "usb", 3)) {

/\* either using usbfs or finding the parent of a root hub \*/

return LIBUSB\_SUCCESS;

}

parent\_sysfs\_dir = strdup(sysfs\_dir);

if (NULL != (tmp = strrchr(parent\_sysfs\_dir, '.')) ||

NULL != (tmp = strrchr(parent\_sysfs\_dir, '-'))) {

dev->port\_number = atoi(tmp + 1);

\*tmp = '\0';

} else {

usbi\_warn(ctx, "Can not parse sysfs\_dir: %s, no parent info",

parent\_sysfs\_dir);

free(parent\_sysfs\_dir);

return LIBUSB\_SUCCESS;

}

/\* is the parent a root hub? \*/

if (NULL == strchr(parent\_sysfs\_dir, '-')) {

tmp = parent\_sysfs\_dir;

ret = asprintf(&parent\_sysfs\_dir, "usb%s", tmp);

free(tmp);

if (0 > ret) {

return LIBUSB\_ERROR\_NO\_MEM;

}

}

retry:

/\* find the parent in the context \*/

usbi\_mutex\_lock(&ctx->usb\_devs\_lock);

list\_for\_each\_entry(it, &ctx->usb\_devs, list, struct libusb\_device)

{

struct android\_device\_priv \*priv = \_device\_priv(it);

if (0 == strcmp(priv->sysfs\_dir, parent\_sysfs\_dir)) {

dev->parent\_dev = libusb\_ref\_device(it);

break;

}

}

usbi\_mutex\_unlock(&ctx->usb\_devs\_lock);

if (!dev->parent\_dev && add\_parent) {

usbi\_dbg("parent\_dev %s not enumerated yet, enumerating now",

parent\_sysfs\_dir);

sysfs\_scan\_device(ctx, parent\_sysfs\_dir);

add\_parent = 0;

goto retry;

}

usbi\_dbg("Dev %p (%s) has parent %p (%s) port %d", dev, sysfs\_dir,

dev->parent\_dev, parent\_sysfs\_dir, dev->port\_number);

free(parent\_sysfs\_dir);

return LIBUSB\_SUCCESS;

}

static int android\_initialize\_device(struct libusb\_device \*dev,

uint8\_t busnum, uint8\_t devaddr, int fd) {

ENTER();

struct android\_device\_priv \*priv = \_device\_priv(dev);

struct libusb\_context \*ctx = DEVICE\_CTX(dev);

uint8\_t desc[4096]; // max descriptor size is 4096 bytes

int speed;

ssize\_t r;

dev->bus\_number = busnum;

dev->device\_address = devaddr;

LOGD("cache descriptors in memory");

priv->descriptors\_len = 0;

priv->fd = 0;

memset(desc, 0, sizeof(desc));

if (!lseek(fd, 0, SEEK\_SET)) {

// ディスクリプタを読み込んでローカルキャッシュする

int length = read(fd, desc, sizeof(desc));

LOGD("Device::init read returned %d errno %d\n", length, errno);

if (length > 0) {

priv->fd = fd;

priv->descriptors = usbi\_reallocf(priv->descriptors, length);

if (UNLIKELY(!priv->descriptors)) {

RETURN(LIBUSB\_ERROR\_NO\_MEM, int);

}

priv->descriptors\_len = length;

memcpy(priv->descriptors, desc, length);

}

}

if (UNLIKELY(priv->descriptors\_len < DEVICE\_DESC\_LENGTH)) {

usbi\_err(ctx, "short descriptor read (%d)", priv->descriptors\_len);

LOGE("short descriptor read (%d)", priv->descriptors\_len);

RETURN(LIBUSB\_ERROR\_IO, int);

}

if (fd < 0) { // if could not get fd of usbfs with read/write access

/\* cannot send a control message to determine the active

\* config. just assume the first one is active. \*/

usbi\_warn(ctx, "Missing rw usbfs access; cannot determine "

"active configuration descriptor");

if (priv->descriptors\_len

>= (DEVICE\_DESC\_LENGTH + LIBUSB\_DT\_CONFIG\_SIZE)) {

struct libusb\_config\_descriptor config;

usbi\_parse\_descriptor(priv->descriptors + DEVICE\_DESC\_LENGTH,

"bbwbbbbb", &config, 0);

priv->active\_config = config.bConfigurationValue;

} else

priv->active\_config = -1; /\* No config dt \*/

RETURN(LIBUSB\_SUCCESS, int);

}

// if we could get fd of usbfs with read/write access

r = usbfs\_get\_active\_config(dev, fd);

if (r > 0) {

priv->active\_config = r;

r = LIBUSB\_SUCCESS;

} else if (r == 0) {

/\* some buggy devices have a configuration 0, but we're

\* reaching into the corner of a corner case here, so let's

\* not support buggy devices in these circumstances.

\* stick to the specs: a configuration value of 0 means

\* unconfigured. \*/

usbi\_dbg("active cfg 0? assuming unconfigured device");

priv->active\_config = -1;

r = LIBUSB\_SUCCESS;

} else if (r == LIBUSB\_ERROR\_IO) {

/\* buggy devices sometimes fail to report their active config.

\* assume unconfigured and continue the probing \*/

usbi\_warn(ctx, "couldn't query active configuration, assuming"

" unconfigured");

priv->active\_config = -1;

r = LIBUSB\_SUCCESS;

} /\* else r < 0, just return the error code \*/

RETURN(r, int);

}

int android\_generate\_device(struct libusb\_context \*ctx, struct libusb\_device \*\*dev,

int vid, int pid, const char \*serial, int fd, int busnum, int devaddr) {

ENTER();

unsigned long session\_id;

int r = 0;

\*dev = NULL;

/\* FIXME: session ID is not guaranteed unique as addresses can wrap and

\* will be reused. instead we should add a simple sysfs attribute with

\* a session ID. \*/

session\_id = busnum << 8 | devaddr;

LOGD("allocating new device for %d/%d (session %ld)", busnum, devaddr, session\_id);

\*dev = usbi\_alloc\_device(ctx, session\_id); // この時点で参照カウンタ=1

if (UNLIKELY(!dev)) {

RETURN(LIBUSB\_ERROR\_NO\_MEM, int);

}

r = android\_initialize\_device(\*dev, busnum, devaddr, fd);

if (UNLIKELY(r < 0)) {

LOGE("initialize\_device failed: ret=%d", r);

goto out;

}

r = usbi\_sanitize\_device(\*dev);

if (UNLIKELY(r < 0)) {

LOGE("usbi\_sanitize\_device failed: ret=%d", r);

goto out;

}

out:

if (UNLIKELY(r < 0)) {

libusb\_unref\_device(\*dev); // ここで参照カウンタが0になって破棄される

\*dev = NULL;

} else {

usbi\_connect\_device(\*dev);

}

RETURN(r, int);

}

int android\_enumerate\_device(struct libusb\_context \*ctx, uint8\_t busnum,

uint8\_t devaddr, const char \*sysfs\_dir) {

unsigned long session\_id;

struct libusb\_device \*dev;

int r = 0;

/\* FIXME: session ID is not guaranteed unique as addresses can wrap and

\* will be reused. instead we should add a simple sysfs attribute with

\* a session ID. \*/

session\_id = busnum << 8 | devaddr;

usbi\_dbg("busnum %d devaddr %d session\_id %ld",

busnum, devaddr, session\_id);

dev = usbi\_get\_device\_by\_session\_id(ctx, session\_id);

if (dev) {

/\* device already exists in the context \*/

usbi\_dbg("session\_id %ld already exists", session\_id);

libusb\_unref\_device(dev);

return LIBUSB\_SUCCESS;

}

usbi\_dbg("allocating new device for %d/%d (session %ld)",

busnum, devaddr, session\_id);

dev = usbi\_alloc\_device(ctx, session\_id);

if (UNLIKELY(!dev))

return LIBUSB\_ERROR\_NO\_MEM;

r = initialize\_device(dev, busnum, devaddr, sysfs\_dir);

if (UNLIKELY(r < 0))

goto out;

r = usbi\_sanitize\_device(dev);

if (UNLIKELY(r < 0))

goto out;

r = android\_get\_parent\_info(dev, sysfs\_dir);

if (UNLIKELY(r < 0))

goto out;

out:

if (UNLIKELY(r < 0))

libusb\_unref\_device(dev);

else

usbi\_connect\_device(dev);

return r;

}

void android\_hotplug\_enumerate(uint8\_t busnum, uint8\_t devaddr,

const char \*sys\_name) {

struct libusb\_context \*ctx;

usbi\_mutex\_static\_lock(&active\_contexts\_lock);

list\_for\_each\_entry(ctx, &active\_contexts\_list, list, struct libusb\_context)

{

android\_enumerate\_device(ctx, busnum, devaddr, sys\_name);

}

usbi\_mutex\_static\_unlock(&active\_contexts\_lock);

}

void android\_device\_disconnected(uint8\_t busnum, uint8\_t devaddr,

const char \*sys\_name) {

struct libusb\_context \*ctx;

struct libusb\_device \*dev;

unsigned long session\_id = busnum << 8 | devaddr;

usbi\_mutex\_static\_lock(&active\_contexts\_lock);

list\_for\_each\_entry(ctx, &active\_contexts\_list, list, struct libusb\_context)

{

dev = usbi\_get\_device\_by\_session\_id(ctx, session\_id);

if (NULL != dev) {

usbi\_disconnect\_device(dev);

libusb\_unref\_device(dev);

} else {

usbi\_dbg("device not found for session %x", session\_id);

}

}

usbi\_mutex\_static\_unlock(&active\_contexts\_lock);

}

#if !defined(USE\_UDEV)

/\* open a bus directory and adds all discovered devices to the context \*/

static int usbfs\_scan\_busdir(struct libusb\_context \*ctx, uint8\_t busnum) {

DIR \*dir;

char dirpath[PATH\_MAX];

struct dirent \*entry;

int r = LIBUSB\_ERROR\_IO;

snprintf(dirpath, PATH\_MAX, "%s/%03d", usbfs\_path, busnum);

usbi\_dbg("%s", dirpath);

dir = opendir(dirpath);

if (UNLIKELY(!dir)) {

usbi\_err(ctx, "opendir '%s' failed, errno=%d", dirpath, errno);

/\* FIXME: should handle valid race conditions like hub unplugged

\* during directory iteration - this is not an error \*/

return r;

}

while ((entry = readdir(dir))) {

int devaddr;

if (entry->d\_name[0] == '.')

continue;

devaddr = atoi(entry->d\_name);

if (devaddr == 0) {

usbi\_dbg("unknown dir entry %s", entry->d\_name);

continue;

}

if (android\_enumerate\_device(ctx, busnum, (uint8\_t) devaddr, NULL)) {

usbi\_dbg("failed to enumerate dir entry %s", entry->d\_name);

continue;

}

r = 0;

}

closedir(dir);

return r;

}

static int usbfs\_get\_device\_list(struct libusb\_context \*ctx) {

struct dirent \*entry;

DIR \*buses = opendir(usbfs\_path);

int r = 0;

if (!buses) {

usbi\_err(ctx, "opendir buses failed errno=%d", errno);

return LIBUSB\_ERROR\_IO;

}

while ((entry = readdir(buses))) {

int busnum;

if (entry->d\_name[0] == '.')

continue;

if (usbdev\_names) {

int devaddr;

if (!\_is\_usbdev\_entry(entry, &busnum, &devaddr))

continue;

r = android\_enumerate\_device(ctx, busnum, (uint8\_t) devaddr, NULL);

if (UNLIKELY(r < 0)) {

usbi\_dbg("failed to enumerate dir entry %s", entry->d\_name);

continue;

}

} else {

busnum = atoi(entry->d\_name);

if (UNLIKELY(busnum == 0)) {

usbi\_dbg("unknown dir entry %s", entry->d\_name);

continue;

}

r = usbfs\_scan\_busdir(ctx, busnum);

if (UNLIKELY(r < 0))

break;

}

}

closedir(buses);

return r;

}

#endif

static int sysfs\_scan\_device(struct libusb\_context \*ctx, const char \*devname) {

uint8\_t busnum, devaddr;

int ret;

ret = android\_get\_device\_address(ctx, 0, &busnum, &devaddr, NULL, devname);

if (UNLIKELY(LIBUSB\_SUCCESS != ret)) {

return ret;

}

return android\_enumerate\_device(ctx, busnum & 0xff, devaddr & 0xff, devname);

}

#if !defined(USE\_UDEV)

static int sysfs\_get\_device\_list(struct libusb\_context \*ctx) {

DIR \*devices = opendir(SYSFS\_DEVICE\_PATH);

struct dirent \*entry;

int r = LIBUSB\_ERROR\_IO;

if (UNLIKELY(!devices)) {

usbi\_err(ctx, "opendir devices failed errno=%d", errno);

return r;

}

while ((entry = readdir(devices))) {

if ((!isdigit(entry->d\_name[0]) && strncmp(entry->d\_name, "usb", 3))

|| strchr(entry->d\_name, ':'))

continue;

if (sysfs\_scan\_device(ctx, entry->d\_name)) {

usbi\_dbg("failed to enumerate dir entry %s", entry->d\_name);

continue;

}

r = 0;

}

closedir(devices);

return r;

}

static int android\_default\_scan\_devices(struct libusb\_context \*ctx) {

/\* we can retrieve device list and descriptors from sysfs or usbfs.

\* sysfs is preferable, because if we use usbfs we end up resuming

\* any autosuspended USB devices. however, sysfs is not available

\* everywhere, so we need a usbfs fallback too.

\*

\* as described in the "sysfs vs usbfs" comment at the top of this

\* file, sometimes we have sysfs but not enough information to

\* relate sysfs devices to usbfs nodes. op\_init() determines the

\* adequacy of sysfs and sets sysfs\_can\_relate\_devices.

\*/

if (sysfs\_can\_relate\_devices != 0)

return sysfs\_get\_device\_list(ctx);

else

return usbfs\_get\_device\_list(ctx);

}

#endif

// this function is mainly for Android

// because native code can not open USB device on Android when without root

// so we need to defer real open/close operation to Java code

static int op\_set\_device\_fd(struct libusb\_device \*device, int fd) {

struct android\_device\_priv \*dpriv = \_device\_priv(device);

dpriv->fd = fd;

return LIBUSB\_SUCCESS;

}

static int op\_open(struct libusb\_device\_handle \*handle) {

struct android\_device\_handle\_priv \*hpriv = \_device\_handle\_priv(handle);

int r;

hpriv->fd = \_get\_usbfs\_fd(handle->dev, O\_RDWR, 0);

if (hpriv->fd < 0) {

if (hpriv->fd == LIBUSB\_ERROR\_NO\_DEVICE) {

/\* device will still be marked as attached if hotplug monitor thread

\* hasn't processed remove event yet \*/

usbi\_mutex\_static\_lock(&android\_hotplug\_lock);

if (handle->dev->attached) {

usbi\_dbg("open failed with no device, but device still attached");

android\_device\_disconnected(handle->dev->bus\_number,

handle->dev->device\_address, NULL);

}

usbi\_mutex\_static\_unlock(&android\_hotplug\_lock);

}

return hpriv->fd;

}

r = ioctl(hpriv->fd, IOCTL\_USBFS\_GET\_CAPABILITIES, &hpriv->caps);

if (UNLIKELY(r < 0)) {

if (errno == ENOTTY)

usbi\_dbg("getcap not available");

else

usbi\_err(HANDLE\_CTX(handle), "getcap failed (%d)", errno);

hpriv->caps = 0;

if (supports\_flag\_zero\_packet)

hpriv->caps |= USBFS\_CAP\_ZERO\_PACKET;

if (supports\_flag\_bulk\_continuation)

hpriv->caps |= USBFS\_CAP\_BULK\_CONTINUATION;

}

return usbi\_add\_pollfd(HANDLE\_CTX(handle), hpriv->fd, POLLOUT);

}

static void op\_close(struct libusb\_device\_handle \*dev\_handle) {

int fd = \_device\_handle\_priv(dev\_handle)->fd;

usbi\_remove\_pollfd(HANDLE\_CTX(dev\_handle), fd);

#ifndef \_\_ANDROID\_\_

// We can not (re)open USB device in the native code on no-rooted Android devices

// so keep open and defer real open/close operation on Java side

close(fd);

#endif

}

static int op\_get\_configuration(struct libusb\_device\_handle \*handle,

int \*config) {

int r;

if (sysfs\_can\_relate\_devices) {

r = sysfs\_get\_active\_config(handle->dev, config);

} else {

r = usbfs\_get\_active\_config(handle->dev,

\_device\_handle\_priv(handle)->fd);

}

if (UNLIKELY(r < 0))

return r;

if (\*config == -1) {

usbi\_err(HANDLE\_CTX(handle), "device unconfigured");

\*config = 0;

}

return LIBUSB\_SUCCESS;

}

static int op\_set\_configuration(struct libusb\_device\_handle \*handle, int config) {

struct android\_device\_priv \*priv = \_device\_priv(handle->dev);

const int fd = \_device\_handle\_priv(handle)->fd;

int r = ioctl(fd, IOCTL\_USBFS\_SETCONFIG, &config);

if (UNLIKELY(r)) {

if (errno == EINVAL) {

return LIBUSB\_ERROR\_NOT\_FOUND;

} else if (errno == EBUSY) {

return LIBUSB\_ERROR\_BUSY;

} else if (errno == ENODEV) {

return LIBUSB\_ERROR\_NO\_DEVICE;

}

usbi\_err(HANDLE\_CTX(handle), "failed, error %d errno %d", r, errno);

return LIBUSB\_ERROR\_OTHER;

}

/\* update our cached active config descriptor \*/

priv->active\_config = config;

return LIBUSB\_SUCCESS;

}

static int claim\_interface(struct libusb\_device\_handle \*handle, int iface) {

ENTER();

const int fd = \_device\_handle\_priv(handle)->fd;

LOGD("interface=%d, fd=%d", iface, fd);

int r = ioctl(fd, IOCTL\_USBFS\_CLAIMINTF, &iface);

if (UNLIKELY(r)) {

if (errno == ENOENT) {

RETURN(LIBUSB\_ERROR\_NOT\_FOUND, int);

} else if (errno == EBUSY) {

RETURN(LIBUSB\_ERROR\_BUSY, int);

} else if (errno == ENODEV) {

RETURN(LIBUSB\_ERROR\_NO\_DEVICE, int);

}

LOGE("claim interface failed, error %d errno %d", r, errno);

RETURN(LIBUSB\_ERROR\_OTHER, int);

}

RETURN(LIBUSB\_SUCCESS, int);

}

static int release\_interface(struct libusb\_device\_handle \*handle, int iface) {

ENTER();

const int fd = \_device\_handle\_priv(handle)->fd;

LOGD("interface=%d, fd=%d", iface, fd);

int r = ioctl(fd, IOCTL\_USBFS\_RELEASEINTF, &iface);

if (UNLIKELY(r)) {

if (errno == ENODEV) {

RETURN(LIBUSB\_ERROR\_NO\_DEVICE, int);

}

LOGE("release interface failed, error %d errno %d", r, errno);

RETURN(LIBUSB\_ERROR\_OTHER, int);

}

RETURN(LIBUSB\_SUCCESS, int);

}

static int op\_set\_interface(struct libusb\_device\_handle \*handle, int iface, int altsetting) {

ENTER();

const int fd = \_device\_handle\_priv(handle)->fd;

struct usbfs\_setinterface setintf;

int r;

setintf.interface = iface;

setintf.altsetting = altsetting;

r = ioctl(fd, IOCTL\_USBFS\_SETINTF, &setintf);

if (UNLIKELY(r)) {

if (errno == EINVAL) {

RETURN(LIBUSB\_ERROR\_NOT\_FOUND, int);

} else if (errno == ENODEV) {

RETURN(LIBUSB\_ERROR\_NO\_DEVICE, int);

}

usbi\_err(HANDLE\_CTX(handle),

"setintf failed error %d errno %d", r, errno);

RETURN(LIBUSB\_ERROR\_OTHER, int);

}

RETURN(LIBUSB\_SUCCESS, int);

}

static int op\_clear\_halt(struct libusb\_device\_handle \*handle,

unsigned char endpoint) {

const int fd = \_device\_handle\_priv(handle)->fd;

unsigned int \_endpoint = endpoint;

int r = ioctl(fd, IOCTL\_USBFS\_CLEAR\_HALT, &\_endpoint);

if (UNLIKELY(r)) {

if (errno == ENOENT)

return LIBUSB\_ERROR\_NOT\_FOUND;

else if (errno == ENODEV)

return LIBUSB\_ERROR\_NO\_DEVICE;

usbi\_err(HANDLE\_CTX(handle),

"clear\_halt failed error %d errno %d", r, errno);

return LIBUSB\_ERROR\_OTHER;

}

return LIBUSB\_SUCCESS;

}

static int op\_reset\_device(struct libusb\_device\_handle \*handle) {

const int fd = \_device\_handle\_priv(handle)->fd;

int i, r, ret = 0;

/\* Doing a device reset will cause the usbfs driver to get unbound

from any interfaces it is bound to. By voluntarily unbinding

the usbfs driver ourself, we stop the kernel from rebinding

the interface after reset (which would end up with the interface

getting bound to the in kernel driver if any). \*/

for (i = 0; i < USB\_MAXINTERFACES; i++) {

if (handle->claimed\_interfaces & (1L << i)) {

release\_interface(handle, i);

}

}

usbi\_mutex\_lock(&handle->lock);

r = ioctl(fd, IOCTL\_USBFS\_RESET, NULL);

if (UNLIKELY(r)) {

if (errno == ENODEV) {

ret = LIBUSB\_ERROR\_NOT\_FOUND;

goto out;

}

usbi\_err(HANDLE\_CTX(handle),

"reset failed error %d errno %d", r, errno);

ret = LIBUSB\_ERROR\_OTHER;

goto out;

}

/\* And re-claim any interfaces which were claimed before the reset \*/

for (i = 0; i < USB\_MAXINTERFACES; i++) {

if (handle->claimed\_interfaces & (1L << i)) {

/\*

\* A driver may have completed modprobing during

\* IOCTL\_USBFS\_RESET, and bound itself as soon as

\* IOCTL\_USBFS\_RESET released the device lock

\*/

r = detach\_kernel\_driver\_and\_claim(handle, i);

if (UNLIKELY(r)) {

usbi\_warn(HANDLE\_CTX(handle),

"failed to re-claim interface %d after reset: %s",

i, libusb\_error\_name(r));

handle->claimed\_interfaces &= ~(1L << i);

ret = LIBUSB\_ERROR\_NOT\_FOUND;

}

}

}

out:

usbi\_mutex\_unlock(&handle->lock);

return ret;

}

static int do\_streams\_ioctl(struct libusb\_device\_handle \*handle, long req,

uint32\_t num\_streams, unsigned char \*endpoints, int num\_endpoints) {

const int fd = \_device\_handle\_priv(handle)->fd;

int r;

struct usbfs\_streams \*streams;

if (num\_endpoints > 30) /\* Max 15 in + 15 out eps \*/

return LIBUSB\_ERROR\_INVALID\_PARAM;

streams = malloc(sizeof(struct usbfs\_streams) + num\_endpoints);

if (!streams)

return LIBUSB\_ERROR\_NO\_MEM;

streams->num\_streams = num\_streams;

streams->num\_eps = num\_endpoints;

memcpy(streams->eps, endpoints, num\_endpoints);

r = ioctl(fd, req, streams);

free(streams);

if (r < 0) {

if (errno == ENOTTY)

return LIBUSB\_ERROR\_NOT\_SUPPORTED;

else if (errno == EINVAL)

return LIBUSB\_ERROR\_INVALID\_PARAM;

else if (errno == ENODEV)

return LIBUSB\_ERROR\_NO\_DEVICE;

usbi\_err(HANDLE\_CTX(handle),

"streams-ioctl failed error %d errno %d", r, errno);

return LIBUSB\_ERROR\_OTHER;

}

return r;

}

static int op\_alloc\_streams(struct libusb\_device\_handle \*handle,

uint32\_t num\_streams, unsigned char \*endpoints, int num\_endpoints)

{

return do\_streams\_ioctl(handle, IOCTL\_USBFS\_ALLOC\_STREAMS,

num\_streams, endpoints, num\_endpoints);

}

static int op\_free\_streams(struct libusb\_device\_handle \*handle,

unsigned char \*endpoints, int num\_endpoints)

{

return do\_streams\_ioctl(handle, IOCTL\_USBFS\_FREE\_STREAMS, 0,

endpoints, num\_endpoints);

}

static int op\_kernel\_driver\_active(struct libusb\_device\_handle \*handle, int interface) {

const int fd = \_device\_handle\_priv(handle)->fd;

struct usbfs\_getdriver getdrv;

int r;

getdrv.interface = interface;

r = ioctl(fd, IOCTL\_USBFS\_GETDRIVER, &getdrv);

if (UNLIKELY(r)) {

if (errno == ENODATA)

return LIBUSB\_SUCCESS;

else if (errno == ENODEV)

return LIBUSB\_ERROR\_NO\_DEVICE;

usbi\_err(HANDLE\_CTX(handle),

"get driver failed error %d errno %d", r, errno);

return LIBUSB\_ERROR\_OTHER;

}

return (strcmp(getdrv.driver, "usbfs") == 0) ? 0 : 1;

}

static int op\_detach\_kernel\_driver(struct libusb\_device\_handle \*handle, int interface) {

const int fd = \_device\_handle\_priv(handle)->fd;

struct usbfs\_ioctl command;

struct usbfs\_getdriver getdrv;

int r;

command.ifno = interface;

command.ioctl\_code = IOCTL\_USBFS\_DISCONNECT;

command.data = NULL;

getdrv.interface = interface;

r = ioctl(fd, IOCTL\_USBFS\_GETDRIVER, &getdrv);

if (r == 0 && strcmp(getdrv.driver, "usbfs") == 0)

return LIBUSB\_ERROR\_NOT\_FOUND;

r = ioctl(fd, IOCTL\_USBFS\_IOCTL, &command);

if (UNLIKELY(r)) {

if (errno == ENODATA)

return LIBUSB\_ERROR\_NOT\_FOUND;

else if (errno == EINVAL)

return LIBUSB\_ERROR\_INVALID\_PARAM;

else if (errno == ENODEV)

return LIBUSB\_ERROR\_NO\_DEVICE;

usbi\_err(HANDLE\_CTX(handle),

"detach failed error %d errno %d", r, errno);

return LIBUSB\_ERROR\_OTHER;

}

return LIBUSB\_SUCCESS;

}

static int op\_attach\_kernel\_driver(struct libusb\_device\_handle \*handle, int interface) {

const int fd = \_device\_handle\_priv(handle)->fd;

struct usbfs\_ioctl command;

int r;

command.ifno = interface;

command.ioctl\_code = IOCTL\_USBFS\_CONNECT;

command.data = NULL;

r = ioctl(fd, IOCTL\_USBFS\_IOCTL, &command);

if (UNLIKELY(r < 0)) {

if (errno == ENODATA)

return LIBUSB\_ERROR\_NOT\_FOUND;

else if (errno == EINVAL)

return LIBUSB\_ERROR\_INVALID\_PARAM;

else if (errno == ENODEV)

return LIBUSB\_ERROR\_NO\_DEVICE;

else if (errno == EBUSY)

return LIBUSB\_ERROR\_BUSY;

usbi\_err(HANDLE\_CTX(handle),

"attach failed error %d errno %d", r, errno);

return LIBUSB\_ERROR\_OTHER;

} else if (UNLIKELY(r == 0)) {

return LIBUSB\_ERROR\_NOT\_FOUND;

}

return LIBUSB\_SUCCESS;

}

static int detach\_kernel\_driver\_and\_claim(struct libusb\_device\_handle \*handle, int interface) {

ENTER();

const int fd = \_device\_handle\_priv(handle)->fd;

struct usbfs\_disconnect\_claim dc;

int r;

dc.interface = interface;

strcpy(dc.driver, "usbfs");

dc.flags = USBFS\_DISCONNECT\_CLAIM\_EXCEPT\_DRIVER;

r = ioctl(fd, IOCTL\_USBFS\_DISCONNECT\_CLAIM, &dc);

if (r == 0 || (r != 0 && errno != ENOTTY)) {

if (r == 0) {

RETURN(LIBUSB\_SUCCESS, int);

}

switch (errno) {

case EBUSY:

RETURN(LIBUSB\_ERROR\_BUSY, int);

case EINVAL:

RETURN(LIBUSB\_ERROR\_INVALID\_PARAM, int);

case ENODEV:

RETURN(LIBUSB\_ERROR\_NO\_DEVICE, int);

}

usbi\_err(HANDLE\_CTX(handle),

"disconnect-and-claim failed errno %d", errno);

RETURN(LIBUSB\_ERROR\_OTHER, int);

}

/\* Fallback code for kernels which don't support the

disconnect-and-claim ioctl \*/

r = op\_detach\_kernel\_driver(handle, interface);

if (r != 0 && r != LIBUSB\_ERROR\_NOT\_FOUND) {

RETURN(r, int);

}

r = claim\_interface(handle, interface);

RETURN(r, int);

}

static int op\_claim\_interface(struct libusb\_device\_handle \*handle, int iface) {

if (handle->auto\_detach\_kernel\_driver)

return detach\_kernel\_driver\_and\_claim(handle, iface);

else

return claim\_interface(handle, iface);

}

static int op\_release\_interface(struct libusb\_device\_handle \*handle, int iface) {

int r;

r = release\_interface(handle, iface);

if (UNLIKELY(r))

return r;

if (handle->auto\_detach\_kernel\_driver)

op\_attach\_kernel\_driver(handle, iface);

return LIBUSB\_SUCCESS;

}

static void op\_destroy\_device(struct libusb\_device \*dev) {

struct android\_device\_priv \*priv = \_device\_priv(dev);

if (priv->descriptors)

free(priv->descriptors);

if (priv->sysfs\_dir)

free(priv->sysfs\_dir);

}

/\* URBs are discarded in reverse order of submission to avoid races. \*/

static int discard\_urbs(struct usbi\_transfer \*itransfer, int first, int last\_plus\_one) {

ENTER();

struct libusb\_transfer \*transfer = USBI\_TRANSFER\_TO\_LIBUSB\_TRANSFER(itransfer);

struct android\_transfer\_priv \*tpriv = usbi\_transfer\_get\_os\_priv(itransfer);

struct android\_device\_handle\_priv \*dpriv = \_device\_handle\_priv(transfer->dev\_handle);

int i, ret = 0;

struct usbfs\_urb \*urb;

for (i = last\_plus\_one - 1; i >= first; i--) {

if (LIBUSB\_TRANSFER\_TYPE\_ISOCHRONOUS == transfer->type)

urb = tpriv->iso\_urbs[i];

else

urb = &tpriv->urbs[i];

// XXX this function call may always fail on non-rooted Android devices with errno=22(EINVAL)...

if (0 == ioctl(dpriv->fd, IOCTL\_USBFS\_DISCARDURB, urb))

continue;

if (EINVAL == errno) {

usbi\_dbg("URB not found --> assuming ready to be reaped");

if (i == (last\_plus\_one - 1))

ret = LIBUSB\_ERROR\_NOT\_FOUND;

} else if (ENODEV == errno) {

usbi\_dbg("Device not found for URB --> assuming ready to be reaped");

ret = LIBUSB\_ERROR\_NO\_DEVICE;

} else {

usbi\_warn(TRANSFER\_CTX(transfer),

"unrecognised discard errno %d", errno);

ret = LIBUSB\_ERROR\_OTHER;

}

}

RETURN(ret, int);

}

static void free\_iso\_urbs(struct android\_transfer\_priv \*tpriv) {

int i;

for (i = 0; i < tpriv->num\_urbs; i++) {

struct usbfs\_urb \*urb = tpriv->iso\_urbs[i];

if (UNLIKELY(!urb))

break;

free(urb);

}

free(tpriv->iso\_urbs);

tpriv->iso\_urbs = NULL;

}

static int submit\_bulk\_transfer(struct usbi\_transfer \*itransfer) {

struct libusb\_transfer \*transfer

= USBI\_TRANSFER\_TO\_LIBUSB\_TRANSFER(itransfer);

struct android\_transfer\_priv \*tpriv = usbi\_transfer\_get\_os\_priv(itransfer);

struct android\_device\_handle\_priv \*dpriv

= \_device\_handle\_priv(transfer->dev\_handle);

struct usbfs\_urb \*urbs;

int is\_out = (transfer->endpoint & LIBUSB\_ENDPOINT\_DIR\_MASK)

== LIBUSB\_ENDPOINT\_OUT;

int bulk\_buffer\_len, use\_bulk\_continuation;

int r;

int i;

size\_t alloc\_size;

if (UNLIKELY(tpriv->urbs))

return LIBUSB\_ERROR\_BUSY;

if (UNLIKELY(is\_out && (transfer->flags & LIBUSB\_TRANSFER\_ADD\_ZERO\_PACKET))

&& !(dpriv->caps & USBFS\_CAP\_ZERO\_PACKET))

return LIBUSB\_ERROR\_NOT\_SUPPORTED;

/\*

\* Older versions of usbfs place a 16kb limit on bulk URBs. We work

\* around this by splitting large transfers into 16k blocks, and then

\* submit all urbs at once. it would be simpler to submit one urb at

\* a time, but there is a big performance gain doing it this way.

\*

\* Newer versions lift the 16k limit (USBFS\_CAP\_NO\_PACKET\_SIZE\_LIM),

\* using arbritary large transfers can still be a bad idea though, as

\* the kernel needs to allocate physical contiguous memory for this,

\* which may fail for large buffers.

\*

\* The kernel solves this problem by splitting the transfer into

\* blocks itself when the host-controller is scatter-gather capable

\* (USBFS\_CAP\_BULK\_SCATTER\_GATHER), which most controllers are.

\*

\* Last, there is the issue of short-transfers when splitting, for

\* short split-transfers to work reliable USBFS\_CAP\_BULK\_CONTINUATION

\* is needed, but this is not always available.

\*/

if (dpriv->caps & USBFS\_CAP\_BULK\_SCATTER\_GATHER) {

/\* Good! Just submit everything in one go \*/

bulk\_buffer\_len = transfer->length ? transfer->length : 1;

use\_bulk\_continuation = 0;

} else if (dpriv->caps & USBFS\_CAP\_BULK\_CONTINUATION) {

/\* Split the transfers and use bulk-continuation to

avoid issues with short-transfers \*/

bulk\_buffer\_len = MAX\_BULK\_BUFFER\_LENGTH;

use\_bulk\_continuation = 1;

} else if (dpriv->caps & USBFS\_CAP\_NO\_PACKET\_SIZE\_LIM) {

/\* Don't split, assume the kernel can alloc the buffer

(otherwise the submit will fail with -ENOMEM) \*/

bulk\_buffer\_len = transfer->length ? transfer->length : 1;

use\_bulk\_continuation = 0;

} else {

/\* Bad, splitting without bulk-continuation, short transfers

which end before the last urb will not work reliable! \*/

/\* Note we don't warn here as this is "normal" on kernels <

2.6.32 and not a problem for most applications \*/

bulk\_buffer\_len = MAX\_BULK\_BUFFER\_LENGTH;

use\_bulk\_continuation = 0;

}

int num\_urbs = transfer->length / bulk\_buffer\_len;

int last\_urb\_partial = 0;

if (transfer->length == 0) {

num\_urbs = 1;

} else if ((transfer->length % bulk\_buffer\_len) > 0) {

last\_urb\_partial = 1;

num\_urbs++;

}

usbi\_dbg("need %d urbs for new transfer with length %d", num\_urbs, transfer->length);

alloc\_size = num\_urbs \* sizeof(struct usbfs\_urb);

urbs = calloc(1, alloc\_size);

if (UNLIKELY(!urbs))

return LIBUSB\_ERROR\_NO\_MEM;

tpriv->urbs = urbs;

tpriv->num\_urbs = num\_urbs;

tpriv->num\_retired = 0;

tpriv->reap\_action = NORMAL;

tpriv->reap\_status = LIBUSB\_TRANSFER\_COMPLETED;

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

struct usbfs\_urb \*urb = &urbs[i];

urb->usercontext = itransfer;

switch (transfer->type) {

case LIBUSB\_TRANSFER\_TYPE\_BULK:

urb->type = USBFS\_URB\_TYPE\_BULK;

urb->stream\_id = 0;

break;

case LIBUSB\_TRANSFER\_TYPE\_BULK\_STREAM:

urb->type = USBFS\_URB\_TYPE\_BULK;

urb->stream\_id = itransfer->stream\_id;

break;

case LIBUSB\_TRANSFER\_TYPE\_INTERRUPT:

urb->type = USBFS\_URB\_TYPE\_INTERRUPT;

break;

}

urb->endpoint = transfer->endpoint;

urb->buffer = transfer->buffer + (i \* bulk\_buffer\_len);

/\* don't set the short not ok flag for the last URB \*/

if (use\_bulk\_continuation && !is\_out && (i < num\_urbs - 1))

urb->flags = USBFS\_URB\_SHORT\_NOT\_OK;

if (i == num\_urbs - 1 && last\_urb\_partial)

urb->buffer\_length = transfer->length % bulk\_buffer\_len;

else if (transfer->length == 0)

urb->buffer\_length = 0;

else

urb->buffer\_length = bulk\_buffer\_len;

if (i > 0 && use\_bulk\_continuation)

urb->flags |= USBFS\_URB\_BULK\_CONTINUATION;

/\* we have already checked that the flag is supported \*/

if (is\_out && i == num\_urbs - 1

&& transfer->flags & LIBUSB\_TRANSFER\_ADD\_ZERO\_PACKET)

urb->flags |= USBFS\_URB\_ZERO\_PACKET;

#if LOCAL\_DEBUG

dump\_urb(i, dpriv->fd, urb);

#endif

r = ioctl(dpriv->fd, IOCTL\_USBFS\_SUBMITURB, urb);

if (UNLIKELY(r < 0)) {

if (errno == ENODEV) {

r = LIBUSB\_ERROR\_NO\_DEVICE;

} else {

usbi\_err(TRANSFER\_CTX(transfer),

"submiturb failed error %d errno=%d", r, errno);

r = LIBUSB\_ERROR\_IO;

}

/\* if the first URB submission fails, we can simply free up and

\* return failure immediately. \*/

if (UNLIKELY(i == 0)) {

usbi\_dbg("first URB failed, easy peasy");

free(urbs);

tpriv->urbs = NULL;

return r;

}

/\* if it's not the first URB that failed, the situation is a bit

\* tricky. we may need to discard all previous URBs. there are

\* complications:

\* - discarding is asynchronous - discarded urbs will be reaped

\* later. the user must not have freed the transfer when the

\* discarded URBs are reaped, otherwise libusb will be using

\* freed memory.

\* - the earlier URBs may have completed successfully and we do

\* not want to throw away any data.

\* - this URB failing may be no error; EREMOTEIO means that

\* this transfer simply didn't need all the URBs we submitted

\* so, we report that the transfer was submitted successfully and

\* in case of error we discard all previous URBs. later when

\* the final reap completes we can report error to the user,

\* or success if an earlier URB was completed successfully.

\*/

tpriv->reap\_action =

EREMOTEIO == errno ? COMPLETED\_EARLY : SUBMIT\_FAILED;

/\* The URBs we haven't submitted yet we count as already

\* retired. \*/

tpriv->num\_retired += num\_urbs - i;

/\* If we completed short then don't try to discard. \*/

if (COMPLETED\_EARLY == tpriv->reap\_action)

return LIBUSB\_SUCCESS;

discard\_urbs(itransfer, 0, i);

usbi\_dbg("reporting successful submission but waiting for %d "

"discards before reporting error", i);

return LIBUSB\_SUCCESS;

}

}

return LIBUSB\_SUCCESS;

}

static int submit\_iso\_transfer(struct usbi\_transfer \*itransfer) {

struct libusb\_transfer \*transfer = USBI\_TRANSFER\_TO\_LIBUSB\_TRANSFER(itransfer);

struct android\_transfer\_priv \*tpriv = usbi\_transfer\_get\_os\_priv(itransfer);

struct android\_device\_handle\_priv \*dpriv = \_device\_handle\_priv(transfer->dev\_handle);

struct usbfs\_urb \*\*urbs;

size\_t alloc\_size;

const int num\_packets = transfer->num\_iso\_packets;

int i;

int this\_urb\_len = 0;

int num\_urbs = 1;

int packet\_offset = 0;

unsigned int packet\_len;

unsigned char \*urb\_buffer = transfer->buffer;

if (UNLIKELY(tpriv->iso\_urbs))

return LIBUSB\_ERROR\_BUSY;

/\* usbfs places a 32kb limit on iso URBs. we divide up larger requests

\* into smaller units to meet such restriction, then fire off all the

\* units at once. it would be simpler if we just fired one unit at a time,

\* but there is a big performance gain through doing it this way.

\*

\* Newer kernels lift the 32k limit (USBFS\_CAP\_NO\_PACKET\_SIZE\_LIM),

\* using arbritary large transfers is still be a bad idea though, as

\* the kernel needs to allocate physical contiguous memory for this,

\* which may fail for large buffers.

\*/

/\* calculate how many URBs we need \*/

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

unsigned int space\_remaining = MAX\_ISO\_BUFFER\_LENGTH - this\_urb\_len;

packet\_len = transfer->iso\_packet\_desc[i].length;

if (packet\_len > space\_remaining) {

num\_urbs++;

this\_urb\_len = packet\_len;

} else {

this\_urb\_len += packet\_len;

}

}

usbi\_dbg("need %d of 32k URBs for transfer", num\_urbs);

alloc\_size = num\_urbs \* sizeof(\*urbs);

urbs = calloc(1, alloc\_size);

if (UNLIKELY(!urbs))

return LIBUSB\_ERROR\_NO\_MEM;

tpriv->iso\_urbs = urbs;

tpriv->num\_urbs = num\_urbs;

tpriv->num\_retired = 0;

tpriv->reap\_action = NORMAL;

tpriv->iso\_packet\_offset = 0;

/\* allocate + initialize each URB with the correct number of packets \*/

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

struct usbfs\_urb \*urb;

unsigned int space\_remaining\_in\_urb = MAX\_ISO\_BUFFER\_LENGTH;

int urb\_packet\_offset = 0;

unsigned char \*urb\_buffer\_orig = urb\_buffer;

int j;

int k;

/\* swallow up all the packets we can fit into this URB \*/

while (packet\_offset < num\_packets) {

packet\_len = transfer->iso\_packet\_desc[packet\_offset].length;

if (packet\_len <= space\_remaining\_in\_urb) {

/\* throw it in \*/

urb\_packet\_offset++;

packet\_offset++;

space\_remaining\_in\_urb -= packet\_len;

urb\_buffer += packet\_len;

} else {

/\* it can't fit, save it for the next URB \*/

break;

}

}

alloc\_size = sizeof(\*urb)

+ (urb\_packet\_offset \* sizeof(struct usbfs\_iso\_packet\_desc));

urb = calloc(1, alloc\_size);

if (UNLIKELY(!urb)) {

free\_iso\_urbs(tpriv);

return LIBUSB\_ERROR\_NO\_MEM;

}

urbs[i] = urb;

/\* populate packet lengths \*/

for (j = 0, k = packet\_offset - urb\_packet\_offset;

k < packet\_offset; k++, j++) {

packet\_len = transfer->iso\_packet\_desc[k].length;

urb->iso\_frame\_desc[j].length = packet\_len;

}

urb->usercontext = itransfer;

urb->type = USBFS\_URB\_TYPE\_ISO;

/\* FIXME: interface for non-ASAP data? \*/

urb->flags = USBFS\_URB\_ISO\_ASAP;

urb->endpoint = transfer->endpoint;

urb->number\_of\_packets = urb\_packet\_offset;

urb->buffer = urb\_buffer\_orig;

}

/\* submit URBs \*/

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

int r = ioctl(dpriv->fd, IOCTL\_USBFS\_SUBMITURB, urbs[i]);

if (UNLIKELY(r < 0)) {

if (errno == ENODEV) {

r = LIBUSB\_ERROR\_NO\_DEVICE;

} else {

usbi\_err(TRANSFER\_CTX(transfer),

"submiturb failed error %d errno=%d", r, errno);

r = LIBUSB\_ERROR\_IO;

}

/\* if the first URB submission fails, we can simply free up and

\* return failure immediately. \*/

if (UNLIKELY(i == 0)) {

usbi\_dbg("first URB failed, easy peasy");

free\_iso\_urbs(tpriv);

return r;

}

/\* if it's not the first URB that failed, the situation is a bit

\* tricky. we must discard all previous URBs. there are

\* complications:

\* - discarding is asynchronous - discarded urbs will be reaped

\* later. the user must not have freed the transfer when the

\* discarded URBs are reaped, otherwise libusb will be using

\* freed memory.

\* - the earlier URBs may have completed successfully and we do

\* not want to throw away any data.

\* so, in this case we discard all the previous URBs BUT we report

\* that the transfer was submitted successfully. then later when

\* the final discard completes we can report error to the user.

\*/

tpriv->reap\_action = SUBMIT\_FAILED;

/\* The URBs we haven't submitted yet we count as already

\* retired. \*/

tpriv->num\_retired = num\_urbs - i;

discard\_urbs(itransfer, 0, i);

usbi\_dbg("reporting successful submission but waiting for %d "

"discards before reporting error", i);

return LIBUSB\_SUCCESS;

}

}

return LIBUSB\_SUCCESS;

}

static int submit\_control\_transfer(struct usbi\_transfer \*itransfer) {

struct android\_transfer\_priv \*tpriv = usbi\_transfer\_get\_os\_priv(itransfer);

struct libusb\_transfer \*transfer

= USBI\_TRANSFER\_TO\_LIBUSB\_TRANSFER(itransfer);

struct android\_device\_handle\_priv \*dpriv

= \_device\_handle\_priv(transfer->dev\_handle);

struct usbfs\_urb \*urb;

int r;

if (UNLIKELY(tpriv->urbs))

return LIBUSB\_ERROR\_BUSY;

if (UNLIKELY(transfer->length - LIBUSB\_CONTROL\_SETUP\_SIZE > MAX\_CTRL\_BUFFER\_LENGTH))

return LIBUSB\_ERROR\_INVALID\_PARAM;

urb = calloc(1, sizeof(struct usbfs\_urb));

if (UNLIKELY(!urb))

return LIBUSB\_ERROR\_NO\_MEM;

tpriv->urbs = urb;

tpriv->num\_urbs = 1;

tpriv->reap\_action = NORMAL;

urb->usercontext = itransfer;

urb->type = USBFS\_URB\_TYPE\_CONTROL;

urb->endpoint = transfer->endpoint;

urb->buffer = transfer->buffer;

urb->buffer\_length = transfer->length;

r = ioctl(dpriv->fd, IOCTL\_USBFS\_SUBMITURB, urb);

if (UNLIKELY(r < 0)) {

free(urb);

tpriv->urbs = NULL;

if (errno == ENODEV)

return LIBUSB\_ERROR\_NO\_DEVICE;

usbi\_err(TRANSFER\_CTX(transfer),

"submiturb failed error %d errno=%d", r, errno);

return LIBUSB\_ERROR\_IO;

}

return LIBUSB\_SUCCESS;

}

static int op\_submit\_transfer(struct usbi\_transfer \*itransfer) {

struct libusb\_transfer \*transfer

= USBI\_TRANSFER\_TO\_LIBUSB\_TRANSFER(itransfer);

switch (transfer->type) {

case LIBUSB\_TRANSFER\_TYPE\_CONTROL:

return submit\_control\_transfer(itransfer);

case LIBUSB\_TRANSFER\_TYPE\_BULK:

case LIBUSB\_TRANSFER\_TYPE\_BULK\_STREAM:

return submit\_bulk\_transfer(itransfer);

case LIBUSB\_TRANSFER\_TYPE\_INTERRUPT:

return submit\_bulk\_transfer(itransfer);

case LIBUSB\_TRANSFER\_TYPE\_ISOCHRONOUS:

return submit\_iso\_transfer(itransfer);

default:

usbi\_err(TRANSFER\_CTX(transfer),

"unknown endpoint type %d", transfer->type);

return LIBUSB\_ERROR\_INVALID\_PARAM;

}

}

static int op\_cancel\_transfer(struct usbi\_transfer \*itransfer) {

ENTER();

struct android\_transfer\_priv \*tpriv = usbi\_transfer\_get\_os\_priv(itransfer);

struct libusb\_transfer \*transfer = USBI\_TRANSFER\_TO\_LIBUSB\_TRANSFER(itransfer);

switch (transfer->type) {

case LIBUSB\_TRANSFER\_TYPE\_BULK:

case LIBUSB\_TRANSFER\_TYPE\_BULK\_STREAM:

if (tpriv->reap\_action == ERROR)

break;

/\* else, fall through \*/

case LIBUSB\_TRANSFER\_TYPE\_CONTROL:

case LIBUSB\_TRANSFER\_TYPE\_INTERRUPT:

case LIBUSB\_TRANSFER\_TYPE\_ISOCHRONOUS:

tpriv->reap\_action = CANCELLED;

break;

default:

usbi\_err(TRANSFER\_CTX(transfer),

"unknown endpoint type %d", transfer->type);

RETURN(LIBUSB\_ERROR\_INVALID\_PARAM, int);

}

if (UNLIKELY(!tpriv->urbs))

RETURN(LIBUSB\_ERROR\_NOT\_FOUND, int);

RETURN(discard\_urbs(itransfer, 0, tpriv->num\_urbs), int);

}

static void op\_clear\_transfer\_priv(struct usbi\_transfer \*itransfer) {

struct libusb\_transfer \*transfer = USBI\_TRANSFER\_TO\_LIBUSB\_TRANSFER(itransfer);

struct android\_transfer\_priv \*tpriv = usbi\_transfer\_get\_os\_priv(itransfer);

/\* urbs can be freed also in submit\_transfer so lock mutex first \*/

switch (transfer->type) {

case LIBUSB\_TRANSFER\_TYPE\_CONTROL:

case LIBUSB\_TRANSFER\_TYPE\_BULK:

case LIBUSB\_TRANSFER\_TYPE\_BULK\_STREAM:

case LIBUSB\_TRANSFER\_TYPE\_INTERRUPT:

usbi\_mutex\_lock(&itransfer->lock);

if (tpriv->urbs)

free(tpriv->urbs);

tpriv->urbs = NULL;

usbi\_mutex\_unlock(&itransfer->lock);

break;

case LIBUSB\_TRANSFER\_TYPE\_ISOCHRONOUS:

usbi\_mutex\_lock(&itransfer->lock);

if (tpriv->iso\_urbs)

free\_iso\_urbs(tpriv);

usbi\_mutex\_unlock(&itransfer->lock);

break;

default:

usbi\_err(TRANSFER\_CTX(transfer),

"unknown endpoint type %d", transfer->type);

}

}

static int handle\_bulk\_completion(struct libusb\_device\_handle \*handle, // XXX added saki

struct usbi\_transfer \*itransfer,

struct usbfs\_urb \*urb) {

struct android\_transfer\_priv \*tpriv = usbi\_transfer\_get\_os\_priv(itransfer);

struct libusb\_transfer \*transfer = USBI\_TRANSFER\_TO\_LIBUSB\_TRANSFER(itransfer);

int urb\_idx = urb - tpriv->urbs;

usbi\_mutex\_lock(&itransfer->lock);

usbi\_dbg("handling completion status %d of bulk urb %d/%d",

urb->status, urb\_idx + 1, tpriv->num\_urbs);

tpriv->num\_retired++;

if (UNLIKELY(tpriv->reap\_action != NORMAL)) {

/\* cancelled, submit\_fail, or completed early \*/

usbi\_dbg("abnormal reap: urb status %d", urb->status);

/\* even though we're in the process of cancelling, it's possible that

\* we may receive some data in these URBs that we don't want to lose.

\* examples:

\* 1. while the kernel is cancelling all the packets that make up an

\* URB, a few of them might complete. so we get back a successful

\* cancellation \*and\* some data.

\* 2. we receive a short URB which marks the early completion condition,

\* so we start cancelling the remaining URBs. however, we're too

\* slow and another URB completes (or at least completes partially).

\* (this can't happen since we always use BULK\_CONTINUATION.)

\*

\* When this happens, our objectives are not to lose any "surplus" data,

\* and also to stick it at the end of the previously-received data

\* (closing any holes), so that libusb reports the total amount of

\* transferred data and presents it in a contiguous chunk.

\*/

if (urb->actual\_length > 0) {

unsigned char \*target = transfer->buffer + itransfer->transferred;

usbi\_dbg("received %d bytes of surplus data", urb->actual\_length);

if (urb->buffer != target) {

usbi\_dbg("moving surplus data from offset %d to offset %d",

(unsigned char \*) urb->buffer - transfer->buffer,

target - transfer->buffer);

memmove(target, urb->buffer, urb->actual\_length);

}

itransfer->transferred += urb->actual\_length;

}

if (tpriv->num\_retired == tpriv->num\_urbs) {

usbi\_dbg("abnormal reap: last URB handled, reporting");

if (tpriv->reap\_action != COMPLETED\_EARLY

&& tpriv->reap\_status == LIBUSB\_TRANSFER\_COMPLETED)

tpriv->reap\_status = LIBUSB\_TRANSFER\_ERROR;

goto completed;

}

goto out\_unlock;

}

itransfer->transferred += urb->actual\_length;

/\* Many of these errors can occur on \*any\* urb of a multi-urb

\* transfer. When they do, we tear down the rest of the transfer.

\*/

switch (urb->status) {

case 0:

break;

case -EREMOTEIO: /\* short transfer \*/

break;

case -ENOENT: /\* cancelled \*/

case -ECONNRESET:

break;

case -ENODEV:

case -ESHUTDOWN:

usbi\_dbg("device removed");

tpriv->reap\_status = LIBUSB\_TRANSFER\_NO\_DEVICE;

goto cancel\_remaining;

case -EPIPE:

usbi\_dbg("detected endpoint stall");

if (tpriv->reap\_status == LIBUSB\_TRANSFER\_COMPLETED)

tpriv->reap\_status = LIBUSB\_TRANSFER\_STALL;

LOGE("LIBUSB\_TRANSFER\_STALL");

op\_clear\_halt(handle, urb->endpoint); // XXX added saki

goto cancel\_remaining;

case -EOVERFLOW:

/\* overflow can only ever occur in the last urb \*/

usbi\_dbg("overflow, actual\_length=%d", urb->actual\_length);

if (tpriv->reap\_status == LIBUSB\_TRANSFER\_COMPLETED)

tpriv->reap\_status = LIBUSB\_TRANSFER\_OVERFLOW;

goto completed;

case -ETIME:

case -EPROTO:

case -EILSEQ:

case -ECOMM:

case -ENOSR:

usbi\_dbg("low level error %d", urb->status);

tpriv->reap\_action = ERROR;

goto cancel\_remaining;

default:

usbi\_warn(ITRANSFER\_CTX(itransfer),

"unrecognised urb status %d", urb->status);

tpriv->reap\_action = ERROR;

goto cancel\_remaining;

}

/\* if we're the last urb or we got less data than requested then we're done \*/

if (urb\_idx == tpriv->num\_urbs - 1) {

usbi\_dbg("last URB in transfer --> complete!");

goto completed;

} else if (urb->actual\_length < urb->buffer\_length) {

usbi\_dbg("short transfer %d/%d --> complete!",

urb->actual\_length, urb->buffer\_length);

if (tpriv->reap\_action == NORMAL)

tpriv->reap\_action = COMPLETED\_EARLY;

} else

goto out\_unlock;

cancel\_remaining:

if (ERROR == tpriv->reap\_action

&& LIBUSB\_TRANSFER\_COMPLETED == tpriv->reap\_status)

tpriv->reap\_status = LIBUSB\_TRANSFER\_ERROR;

if (tpriv->num\_retired == tpriv->num\_urbs) /\* nothing to cancel \*/

goto completed;

/\* cancel remaining urbs and wait for their completion before reporting results \*/

discard\_urbs(itransfer, urb\_idx + 1, tpriv->num\_urbs);

out\_unlock:

usbi\_mutex\_unlock(&itransfer->lock);

return LIBUSB\_SUCCESS;

completed:

if (tpriv->urbs)

free(tpriv->urbs);

tpriv->urbs = NULL;

usbi\_mutex\_unlock(&itransfer->lock);

return CANCELLED == tpriv->reap\_action ?

usbi\_handle\_transfer\_cancellation(itransfer) :

usbi\_handle\_transfer\_completion(itransfer, tpriv->reap\_status);

}

static int handle\_iso\_completion(struct libusb\_device\_handle \*handle, // XXX added saki

struct usbi\_transfer \*itransfer,

struct usbfs\_urb \*urb) {

struct libusb\_transfer \*transfer = USBI\_TRANSFER\_TO\_LIBUSB\_TRANSFER(itransfer);

struct android\_transfer\_priv \*tpriv = usbi\_transfer\_get\_os\_priv(itransfer);

int num\_urbs = tpriv->num\_urbs;

int urb\_idx = 0;

int i;

enum libusb\_transfer\_status status = LIBUSB\_TRANSFER\_COMPLETED;

usbi\_mutex\_lock(&itransfer->lock);

if(tpriv->iso\_urbs == NULL){

return LIBUSB\_SUCCESS;

}

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

if (urb == tpriv->iso\_urbs[i]) {

urb\_idx = i + 1;

break;

}

}

if (UNLIKELY(urb\_idx == 0)) {

usbi\_err(TRANSFER\_CTX(transfer), "could not locate urb!"); // crash 2014/09/29 SIGSEGV/SEGV\_MAPERR

usbi\_mutex\_unlock(&itransfer->lock);

return LIBUSB\_ERROR\_NOT\_FOUND;

}

usbi\_dbg("handling completion status %d of iso urb %d/%d",

urb->status, urb\_idx, num\_urbs);

/\* copy isochronous results back in \*/

for (i = 0; i < urb->number\_of\_packets; i++) {

struct usbfs\_iso\_packet\_desc \*urb\_desc = &urb->iso\_frame\_desc[i];

struct libusb\_iso\_packet\_descriptor \*lib\_desc =

&transfer->iso\_packet\_desc[tpriv->iso\_packet\_offset++];

lib\_desc->status = LIBUSB\_TRANSFER\_COMPLETED;

switch (urb\_desc->status) {

case 0:

break;

case -ENOENT: /\* cancelled \*/

case -ECONNRESET:

break;

case -ENODEV:

case -ESHUTDOWN:

usbi\_dbg("device removed");

lib\_desc->status = LIBUSB\_TRANSFER\_NO\_DEVICE;

break;

case -EPIPE:

usbi\_dbg("detected endpoint stall");

lib\_desc->status = LIBUSB\_TRANSFER\_STALL;

LOGE("LIBUSB\_TRANSFER\_STALL");

op\_clear\_halt(handle, urb->endpoint); // XXX added saki

break;

case -EOVERFLOW:

usbi\_dbg("overflow error");

lib\_desc->status = LIBUSB\_TRANSFER\_OVERFLOW;

break;

case -ETIME:

case -EPROTO:

case -EILSEQ:

case -ECOMM:

case -ENOSR:

case -EXDEV:

usbi\_dbg("low-level USB error %d", urb\_desc->status);

lib\_desc->status = LIBUSB\_TRANSFER\_ERROR;

break;

default:

usbi\_warn(TRANSFER\_CTX(transfer),

"unrecognised urb status %d", urb\_desc->status);

lib\_desc->status = LIBUSB\_TRANSFER\_ERROR;

break;

}

lib\_desc->actual\_length = urb\_desc->actual\_length;

}

tpriv->num\_retired++;

if (UNLIKELY(tpriv->reap\_action != NORMAL)) { /\* cancelled or submit\_fail \*/

usbi\_dbg("CANCEL: urb status %d", urb->status);

if (tpriv->num\_retired == num\_urbs) {

usbi\_dbg("CANCEL: last URB handled, reporting");

free\_iso\_urbs(tpriv);

if (tpriv->reap\_action == CANCELLED) {

usbi\_mutex\_unlock(&itransfer->lock);

return usbi\_handle\_transfer\_cancellation(itransfer);

} else {

usbi\_mutex\_unlock(&itransfer->lock);

return usbi\_handle\_transfer\_completion(itransfer, LIBUSB\_TRANSFER\_ERROR);

}

}

goto out;

}

switch (urb->status) {

case 0:

break;

case -ENOENT: /\* cancelled \*/

case -ECONNRESET:

break;

case -ESHUTDOWN:

usbi\_dbg("device removed");

status = LIBUSB\_TRANSFER\_NO\_DEVICE;

break;

default:

usbi\_warn(TRANSFER\_CTX(transfer),

"unrecognised urb status %d", urb->status);

status = LIBUSB\_TRANSFER\_ERROR;

break;

}

/\* if we're the last urb then we're done \*/

if (urb\_idx == num\_urbs) {

usbi\_dbg("last URB in transfer --> complete!");

free\_iso\_urbs(tpriv);

usbi\_mutex\_unlock(&itransfer->lock);

return usbi\_handle\_transfer\_completion(itransfer, status);

}

out:

usbi\_mutex\_unlock(&itransfer->lock);

return LIBUSB\_SUCCESS;

}

static int handle\_control\_completion(struct libusb\_device\_handle \*handle, // XXX added saki

struct usbi\_transfer \*itransfer,

struct usbfs\_urb \*urb) {

struct android\_transfer\_priv \*tpriv = usbi\_transfer\_get\_os\_priv(itransfer);

int status;

usbi\_mutex\_lock(&itransfer->lock);

usbi\_dbg("handling completion status %d", urb->status);

itransfer->transferred += urb->actual\_length;

if (UNLIKELY(tpriv->reap\_action == CANCELLED)) {

if (urb->status != 0 && urb->status != -ENOENT)

usbi\_warn(ITRANSFER\_CTX(itransfer),

"cancel: unrecognised urb status %d", urb->status);

if (tpriv->urbs) {

free(tpriv->urbs);

tpriv->urbs = NULL;

}

usbi\_mutex\_unlock(&itransfer->lock);

return usbi\_handle\_transfer\_cancellation(itransfer);

}

switch (urb->status) {

case 0:

status = LIBUSB\_TRANSFER\_COMPLETED;

break;

case -ENOENT: /\* cancelled \*/

status = LIBUSB\_TRANSFER\_CANCELLED;

break;

case -ENODEV:

case -ESHUTDOWN:

usbi\_dbg("device removed");

status = LIBUSB\_TRANSFER\_NO\_DEVICE;

break;

case -EPIPE:

usbi\_dbg("unsupported control request");

status = LIBUSB\_TRANSFER\_STALL;

LOGE("LIBUSB\_TRANSFER\_STALL");

op\_clear\_halt(handle, urb->endpoint); // XXX added saki

break;

case -EOVERFLOW:

usbi\_dbg("control overflow error");

status = LIBUSB\_TRANSFER\_OVERFLOW;

break;

case -ETIME:

case -EPROTO:

case -EILSEQ:

case -ECOMM:

case -ENOSR:

usbi\_dbg("low-level bus error occurred");

status = LIBUSB\_TRANSFER\_ERROR;

break;

default:

usbi\_warn(ITRANSFER\_CTX(itransfer),

"unrecognised urb status %d", urb->status);

status = LIBUSB\_TRANSFER\_ERROR;

break;

}

if (tpriv->urbs) {

free(tpriv->urbs); // crash

tpriv->urbs = NULL;

}

usbi\_mutex\_unlock(&itransfer->lock);

return usbi\_handle\_transfer\_completion(itransfer, status);

}

static int reap\_for\_handle(struct libusb\_device\_handle \*handle) {

struct android\_device\_handle\_priv \*hpriv = \_device\_handle\_priv(handle);

int r;

struct usbfs\_urb \*urb;

struct usbi\_transfer \*itransfer;

struct libusb\_transfer \*transfer;

r = ioctl(hpriv->fd, IOCTL\_USBFS\_REAPURBNDELAY, &urb);

if (r == -1 && errno == EAGAIN)

return 1;

if (UNLIKELY(r < 0)) {

if (errno == ENODEV)

return LIBUSB\_ERROR\_NO\_DEVICE;

usbi\_err(HANDLE\_CTX(handle), "reap failed error %d errno=%d", r, errno);

return LIBUSB\_ERROR\_IO;

}

itransfer = urb->usercontext;

transfer = USBI\_TRANSFER\_TO\_LIBUSB\_TRANSFER(itransfer);

usbi\_dbg("urb type=%d status=%d transferred=%d",

urb->type, urb->status, urb->actual\_length);

switch (transfer->type) {

case LIBUSB\_TRANSFER\_TYPE\_ISOCHRONOUS:

return handle\_iso\_completion(handle, itransfer, urb);

case LIBUSB\_TRANSFER\_TYPE\_BULK:

case LIBUSB\_TRANSFER\_TYPE\_BULK\_STREAM:

case LIBUSB\_TRANSFER\_TYPE\_INTERRUPT:

return handle\_bulk\_completion(handle, itransfer, urb);

case LIBUSB\_TRANSFER\_TYPE\_CONTROL:

return handle\_control\_completion(handle, itransfer, urb);

default:

usbi\_err(HANDLE\_CTX(handle),

"unrecognised endpoint type %x", transfer->type);

return LIBUSB\_ERROR\_OTHER;

}

}

static int op\_handle\_events(struct libusb\_context \*ctx, struct pollfd \*fds,

POLL\_NFDS\_TYPE nfds, int num\_ready) {

int r;

unsigned int i = 0;

usbi\_mutex\_lock(&ctx->open\_devs\_lock);

for (i = 0; i < nfds && num\_ready > 0; i++) {

struct pollfd \*pollfd = &fds[i];

struct libusb\_device\_handle \*handle;

struct android\_device\_handle\_priv \*hpriv = NULL;

if (!pollfd->revents)

continue;

num\_ready--;

list\_for\_each\_entry(handle, &ctx->open\_devs, list, struct libusb\_device\_handle)

{

hpriv = \_device\_handle\_priv(handle);

if (hpriv->fd == pollfd->fd)

break;

}

if (!hpriv || hpriv->fd != pollfd->fd) {

usbi\_err(ctx, "cannot find handle for fd %d\n",

pollfd->fd);

continue;

}

if (pollfd->revents & POLLERR) {

usbi\_remove\_pollfd(HANDLE\_CTX(handle), hpriv->fd);

usbi\_mutex\_lock(&ctx->events\_lock); // XXX as a note of usbi\_handle\_disconnect shows that need event\_lock locked

usbi\_handle\_disconnect(handle);

usbi\_mutex\_unlock(&ctx->events\_lock); // XXX

/\* device will still be marked as attached if hotplug monitor thread

\* hasn't processed remove event yet \*/

usbi\_mutex\_static\_lock(&android\_hotplug\_lock);

if (handle->dev->attached)

android\_device\_disconnected(handle->dev->bus\_number,

handle->dev->device\_address, NULL);

usbi\_mutex\_static\_unlock(&android\_hotplug\_lock);

continue;

}

do {

r = reap\_for\_handle(handle);

} while (r == 0);

if (r == 1 || r == LIBUSB\_ERROR\_NO\_DEVICE)

continue;

else if (r < 0)

goto out;

}

r = 0;

out:

usbi\_mutex\_unlock(&ctx->open\_devs\_lock);

return r;

}

static int op\_clock\_gettime(int clk\_id, struct timespec \*tp) {

switch (clk\_id) {

case USBI\_CLOCK\_MONOTONIC:

return clock\_gettime(monotonic\_clkid, tp);

case USBI\_CLOCK\_REALTIME:

return clock\_gettime(CLOCK\_REALTIME, tp);

default:

return LIBUSB\_ERROR\_INVALID\_PARAM;

}

}

#ifdef USBI\_TIMERFD\_AVAILABLE

static clockid\_t op\_get\_timerfd\_clockid(void)

{

return monotonic\_clkid;

}

#endif

const struct usbi\_os\_backend android\_usbfs\_backend = {

.name = "Android usbfs",

.caps = USBI\_CAP\_HAS\_HID\_ACCESS | USBI\_CAP\_SUPPORTS\_DETACH\_KERNEL\_DRIVER,

.init = op\_init,

.init2 = op\_init2, // XXX

.exit = op\_exit,

.get\_device\_list = NULL,

.hotplug\_poll = op\_hotplug\_poll,

.get\_raw\_descriptor = op\_get\_raw\_descriptor, // XXX

.get\_device\_descriptor = op\_get\_device\_descriptor,

.get\_active\_config\_descriptor = op\_get\_active\_config\_descriptor,

.get\_config\_descriptor = op\_get\_config\_descriptor,

.get\_config\_descriptor\_by\_value = op\_get\_config\_descriptor\_by\_value,

.set\_device\_fd = op\_set\_device\_fd, // XXX add for no-rooted Android devices

.open = op\_open,

.close = op\_close,

.get\_configuration = op\_get\_configuration,

.set\_configuration = op\_set\_configuration,

.claim\_interface = op\_claim\_interface,

.release\_interface = op\_release\_interface,

.set\_interface\_altsetting = op\_set\_interface,

.clear\_halt = op\_clear\_halt,

.reset\_device = op\_reset\_device,

.alloc\_streams = op\_alloc\_streams,

.free\_streams = op\_free\_streams,

.kernel\_driver\_active = op\_kernel\_driver\_active,

.detach\_kernel\_driver = op\_detach\_kernel\_driver,

.attach\_kernel\_driver = op\_attach\_kernel\_driver,

.destroy\_device = op\_destroy\_device,

.submit\_transfer = op\_submit\_transfer,

.cancel\_transfer = op\_cancel\_transfer,

.clear\_transfer\_priv = op\_clear\_transfer\_priv,

.handle\_events = op\_handle\_events,

.clock\_gettime = op\_clock\_gettime,

#ifdef USBI\_TIMERFD\_AVAILABLE

.get\_timerfd\_clockid = op\_get\_timerfd\_clockid,

#endif

.device\_priv\_size = sizeof(struct android\_device\_priv),

.device\_handle\_priv\_size = sizeof(struct android\_device\_handle\_priv),

.transfer\_priv\_size = sizeof(struct android\_transfer\_priv),

.add\_iso\_packet\_size = 0,

};

